UNIVERSITY OF SURREY

Meta-Compilation for C++

A dissertation for the degree of Doctor of Philosophy

Edward D. Willink

29-June-2001
DEFENDED POSITION

- Meta-programming is essential for significant applications.
- Meta-programming is best achieved in C++ by elevating existing language constructs to the meta-level.
- The FOG solution for meta-programming is logical, systematic and adequate for typical applications.
- Compilation problems of both FOG and plain C++ are resolvable by introduction of a clear partitioning between syntactic and semantic analyses.
- FOG renders the C preprocessor redundant.
RELEVANT PUBLICATIONS BY THE AUTHOR


Page iv 29-June-2001
Software Engineering progresses as improvements are made in languages and methodologies. Significant advances have been made through the use of Object-Oriented Programming, exploiting the effective support available in C++. Further evolution of OOP involving the use of design patterns and aspects requires additional language support.

Increased flexibility in the declaration of objects is proposed in the form of the FOG (Flexible Object Generator) language, which is a superset of C++ implemented by a translator to C++. FOG generalises C++ syntax and supports compile-time meta-programming and reflection.

The syntax generalisations provide the freedom for programmers to organise code to suit programming concerns and eliminate the need for duplication between interfaces and implementations. Further generalisations define composition policies for repeated declarations so that classes, arrays, enumerations and functions may all be extended. These composition policies support the weaving necessary for re-useable implementation of design patterns and for Aspect Oriented Programming.

A declarative form of meta-programming is supported by derivation rules, which specify how a declaration specified in a base class is to be reinterpreted in derived classes. Automated generation of derived functionality is important for a number of design patterns.

More general meta-programming is provided by elevating most run-time concepts to the meta-level, allowing conditional and iterated manipulation of declarations at compile-time. Compile-time execution enables subsequent run-time code to be optimised to suit application requirements.

The use of meta-variables and meta-functions together with a well-disciplined lexical context for meta-programming and meta-level execution provide a complete replacement for the traditional C preprocessor functionality, satisfying Stroustrup's goal of making Cpp redundant [Stroustrup97]. The new functionality is integrated with the language, fits within an Object-Oriented framework and provides adequate support for modern Software Engineering practices.

The C++ grammar is known to pose a significant parsing challenge and to require context dependent type and template knowledge. This creates considerable difficulties when meta-programming occurs in unresolved contexts. A new approach to parsing C++ has therefore been developed that defers the use of type and template information. This approach leads to a simpler grammar implementation. An extended form of regular expression is presented and used to predict known ambiguities and then show that this simpler grammar covers the C++ grammar.
ACKNOWLEDGEMENTS

My various academic supervisors gave invaluable support, while they remained at Surrey. Chris Jesshope got me started and prompted me to broaden my programming outlook. Sasha Shafarenko never allowed me to become complacent. Slava Muchnick explained some of the intricacies of data parallelism and then encouraged me as the research direction changed. Slava read drafts enthusiastically, challenging my English from a uniquely Russian perspective.


Eventually George Pavlou picked up the reins and ensured that a suitable portion of these suggestions were incorporated to produce this much improved revision.

My industrial supervisor, Edward Stansfield, ensured everything ran smoothly from the industrial angle.

Brian Hillam offered encouragement and a suitably objective critique of associated papers. Lui Melechi provided reassurance that the direction was sensible.

And finally, of course, I am extremely grateful to my employer, Thales Research (then called Racal Research), for supplying the motivation, time and resources, and to Peter Blair for presenting the opportunity and then allowing the research to take its course.
# TABLE OF CONTENTS

**DEFENDED POSITION** ................................................................. iii  
**RELEVANT PUBLICATIONS BY THE AUTHOR** .............................. iv  
**ABSTRACT** .................................................................................... v  
**ACKNOWLEDGEMENTS** ............................................................... vi  
**TABLE OF CONTENTS** ............................................................... vii  
**LIST OF FIGURES** ....................................................................... xvii  
**LIST OF TABLES** .......................................................................... xvii

## 1 Introduction  ................................................................................. 1  
1.1 Language Limitations ............................................................... 5  
1.2 Organisation ............................................................................... 6  
1.3 Conventions ............................................................................... 7  
1.4 Traditional Preprocessing ......................................................... 8  
1.4.1 Lexical substitution ............................................................... 9  
1.4.2 Name concatenation ............................................................ 9  
1.4.3 String conversion ............................................................... 9  
1.4.4 Text replacement ............................................................... 9  
1.4.5 Conditional compilation ..................................................... 10  
1.5 Object-Oriented preprocessing ............................................... 11  
1.5.1 Scopes ............................................................................... 11  
1.5.2 Joint interface and implementation ..................................... 12  
1.5.3 Composition ....................................................................... 13  
1.5.4 Derivation rules .................................................................. 15  
1.5.5 Compilation model ............................................................ 17  
1.5.6 Meta-concepts .................................................................... 18  
1.5.7 Meta-Programming .......................................................... 18  
1.6 FOG versions ........................................................................... 18  

## 2 Related Work ............................................................................. 21  
2.1 Language Constructs ............................................................... 21  
2.1.1 Syntax ............................................................................... 21  
2.1.2 Macros ............................................................................... 21  
2.1.3 Joins .................................................................................. 22  
2.1.4 Meta-classes ...................................................................... 23  
2.2 Meta-level and Reflection ....................................................... 23  
2.2.1 The tower .......................................................................... 23  
2.2.2 Metaobject Protocols ......................................................... 24  
2.2.3 Languages ......................................................................... 24  
2.2.4 Applications ....................................................................... 25  
2.3 Programming Styles ............................................................... 26  
2.3.1 Patterns ............................................................................ 26  
2.3.2 Aspects ............................................................................ 27  
2.3.3 Generative Programming .................................................. 29  
2.4 Applications ............................................................................ 30  
2.4.1 Design by contract .......................................................... 30  
2.4.2 Persistence and Marshalling .............................................. 30  
2.4.3 Synchronisation ............................................................... 31  
2.5 Summary ................................................................................ 31  

## 3 FOG Grammar ............................................................................ 33  
3.1 Grammar Extensions ............................................................... 33  
3.1.1 Substitution, Concatenation and Tokenization .................... 33  
3.1.1.1 C++ Phase 6 Concatenation Grammar ............................ 34  
3.1.1.2 C++ Phase 7 Tokenization Grammar ............................ 34
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1.3 FOG Phase 6 Concatenation Grammar</td>
<td>35</td>
</tr>
<tr>
<td>3.1.1.4 FOG Phase 7 Tokenization Grammar</td>
<td>36</td>
</tr>
<tr>
<td>3.1.1.5 character-literal</td>
<td>38</td>
</tr>
<tr>
<td>3.1.1.6 FOG tree-literals</td>
<td>38</td>
</tr>
<tr>
<td>3.1.1.7 Design Rationale</td>
<td>42</td>
</tr>
<tr>
<td>3.1.2 Names</td>
<td>43</td>
</tr>
<tr>
<td>3.1.3 Syntax Generalisation</td>
<td>45</td>
</tr>
<tr>
<td>3.1.3.1 Forward declaration for namespace</td>
<td>45</td>
</tr>
<tr>
<td>3.1.3.2 access-specifiers as decl-specifiers</td>
<td>46</td>
</tr>
<tr>
<td>3.1.3.3 Pure-virtual</td>
<td>46</td>
</tr>
<tr>
<td>3.1.3.4 !static</td>
<td>47</td>
</tr>
<tr>
<td>3.1.3.5 !inline</td>
<td>47</td>
</tr>
<tr>
<td>3.1.3.6 !virtual</td>
<td>48</td>
</tr>
<tr>
<td>3.1.4 Syntax Enhancements</td>
<td>49</td>
</tr>
<tr>
<td>3.1.4.1 Default member initializer</td>
<td>49</td>
</tr>
<tr>
<td>3.1.4.2 gcc indexed array initializer</td>
<td>50</td>
</tr>
<tr>
<td>3.1.4.3 compound-declaration</td>
<td>51</td>
</tr>
<tr>
<td>3.1.4.4 using</td>
<td>51</td>
</tr>
<tr>
<td>3.1.4.5 using template</td>
<td>54</td>
</tr>
<tr>
<td>3.1.4.6 Object-statement-scopes</td>
<td>55</td>
</tr>
<tr>
<td>3.1.4.7 Function-statement-scopes</td>
<td>56</td>
</tr>
<tr>
<td>3.1.4.8 Derivation Rules</td>
<td>57</td>
</tr>
<tr>
<td>3.1.5 Meta-Programming</td>
<td>59</td>
</tr>
<tr>
<td>3.1.5.1 Meta-names</td>
<td>63</td>
</tr>
<tr>
<td>3.1.5.2 Meta-classes</td>
<td>63</td>
</tr>
<tr>
<td>3.1.5.3 Base Meta-classes</td>
<td>64</td>
</tr>
<tr>
<td>3.1.5.4 Meta-types</td>
<td>65</td>
</tr>
<tr>
<td>3.1.5.5 Meta-variables</td>
<td>66</td>
</tr>
<tr>
<td>3.1.5.6 Meta-functions</td>
<td>67</td>
</tr>
<tr>
<td>3.1.5.7 Meta-constructor and Meta-destructor</td>
<td>68</td>
</tr>
<tr>
<td>3.1.5.8 Meta-control-statements and meta-control-declarations</td>
<td>68</td>
</tr>
<tr>
<td>3.1.5.9 Meta-typedef</td>
<td>70</td>
</tr>
<tr>
<td>3.1.5.10 Meta-expression-statement</td>
<td>70</td>
</tr>
<tr>
<td>3.2 Built-In Functionality</td>
<td>73</td>
</tr>
<tr>
<td>3.2.1 Built-in Root Meta-class</td>
<td>73</td>
</tr>
<tr>
<td>3.2.2 Built-in Meta-variables</td>
<td>73</td>
</tr>
<tr>
<td>3.2.3 Built-in Meta-functions</td>
<td>73</td>
</tr>
<tr>
<td>3.2.4 std meta-namespace</td>
<td>73</td>
</tr>
<tr>
<td>3.3 Incompatibilities</td>
<td>73</td>
</tr>
<tr>
<td>3.3.1 Semantic Errors</td>
<td>74</td>
</tr>
<tr>
<td>3.3.2 Transparency</td>
<td>74</td>
</tr>
<tr>
<td>3.3.3 auto</td>
<td>74</td>
</tr>
<tr>
<td>3.3.4 Incompatible concatenation</td>
<td>74</td>
</tr>
<tr>
<td>3.4 Cpp Replacement</td>
<td>75</td>
</tr>
<tr>
<td>3.4.1 Cpp limitations</td>
<td>75</td>
</tr>
<tr>
<td>3.4.1.1 Unwanted substitution</td>
<td>76</td>
</tr>
<tr>
<td>3.4.1.2 Language independence</td>
<td>76</td>
</tr>
<tr>
<td>3.4.1.3 Side-effects</td>
<td>76</td>
</tr>
<tr>
<td>3.4.1.4 Substitution level</td>
<td>77</td>
</tr>
<tr>
<td>3.4.1.5 Backslash continuations</td>
<td>78</td>
</tr>
<tr>
<td>3.4.2 Concatenation and Stringizing</td>
<td>78</td>
</tr>
<tr>
<td>3.4.3 #define directive</td>
<td>79</td>
</tr>
<tr>
<td>3.4.4 #include directive</td>
<td>79</td>
</tr>
<tr>
<td>3.4.5 #if, #ifdef, #ifndef, #else, #elif, #endif directives</td>
<td>79</td>
</tr>
<tr>
<td>3.4.6 #line directive</td>
<td>80</td>
</tr>
<tr>
<td>3.4.7 #error directive</td>
<td>80</td>
</tr>
<tr>
<td>3.4.8 #pragma directive</td>
<td>80</td>
</tr>
</tbody>
</table>
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Alternatives</td>
<td>148</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Roskind grammar</td>
<td>148</td>
</tr>
<tr>
<td>5.3.2</td>
<td>gcc</td>
<td>149</td>
</tr>
<tr>
<td>5.3.3</td>
<td>CPPP</td>
<td>149</td>
</tr>
<tr>
<td>5.3.4</td>
<td>PCCTS</td>
<td>150</td>
</tr>
<tr>
<td>5.3.5</td>
<td>C++ to F-code translator</td>
<td>150</td>
</tr>
<tr>
<td>5.4</td>
<td>FOG parsing</td>
<td>150</td>
</tr>
<tr>
<td>5.5</td>
<td>Analysis of the C++ Grammar</td>
<td>153</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Notation</td>
<td>153</td>
</tr>
<tr>
<td>5.5.2</td>
<td>C++ Grammar Properties</td>
<td>156</td>
</tr>
<tr>
<td>5.5.2.1</td>
<td>Names</td>
<td>156</td>
</tr>
<tr>
<td>5.5.2.2</td>
<td>Declarators, Declarations and Type Identifiers</td>
<td>160</td>
</tr>
<tr>
<td>5.5.2.3</td>
<td>Expressions</td>
<td>162</td>
</tr>
<tr>
<td>5.5.3</td>
<td>C++ Ambiguities (using type information)</td>
<td>164</td>
</tr>
<tr>
<td>5.5.3.1</td>
<td>Declaration / Declaration Ambiguity</td>
<td>164</td>
</tr>
<tr>
<td>5.5.3.2</td>
<td>Declaration / Expression ambiguity</td>
<td>167</td>
</tr>
<tr>
<td>5.5.3.3</td>
<td>Expression / Expression ambiguities</td>
<td>170</td>
</tr>
<tr>
<td>5.5.3.4</td>
<td>type-id / expression-list ambiguity</td>
<td>171</td>
</tr>
<tr>
<td>5.5.3.5</td>
<td>call / functional-cast ambiguity</td>
<td>172</td>
</tr>
<tr>
<td>5.6</td>
<td>Parsing the ambiguities</td>
<td>172</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Parsing against an ambiguity (the traditional approach)</td>
<td>173</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Parsing without an ambiguity (the multi-pass approach)</td>
<td>174</td>
</tr>
<tr>
<td>5.6.3</td>
<td>Parsing with the ambiguity (the superset approach)</td>
<td>175</td>
</tr>
<tr>
<td>5.7</td>
<td>The Superset Grammar Approach</td>
<td>175</td>
</tr>
<tr>
<td>5.7.1</td>
<td>C++ Ambiguities (without type information)</td>
<td>176</td>
</tr>
<tr>
<td>5.7.1.1</td>
<td>Declaration / Expression ambiguity</td>
<td>176</td>
</tr>
<tr>
<td>5.7.1.2</td>
<td>Expression / Expression ambiguity</td>
<td>177</td>
</tr>
<tr>
<td>5.7.1.3</td>
<td>type-id / expression-list ambiguity</td>
<td>177</td>
</tr>
<tr>
<td>5.7.1.4</td>
<td>Call / functional-cast ambiguity</td>
<td>178</td>
</tr>
<tr>
<td>5.7.2</td>
<td>A naive Assignment-Expression / Parameter-Declaration superset</td>
<td>178</td>
</tr>
<tr>
<td>5.7.3</td>
<td>The Superset</td>
<td>179</td>
</tr>
<tr>
<td>5.7.3.1</td>
<td>Generalised Name</td>
<td>179</td>
</tr>
<tr>
<td>5.7.3.2</td>
<td>Generalised Array</td>
<td>181</td>
</tr>
<tr>
<td>5.7.3.3</td>
<td>Generalised Parentheses</td>
<td>181</td>
</tr>
<tr>
<td>5.7.3.4</td>
<td>Generalised pointers</td>
<td>182</td>
</tr>
<tr>
<td>5.7.3.5</td>
<td>Generalised primary-expression</td>
<td>182</td>
</tr>
<tr>
<td>5.7.3.6</td>
<td>Generalised assignment-expression</td>
<td>182</td>
</tr>
<tr>
<td>5.7.3.7</td>
<td>Generalised parameter-declaration</td>
<td>183</td>
</tr>
<tr>
<td>5.7.3.8</td>
<td>Coverage of the generalised parameter-declaration</td>
<td>183</td>
</tr>
<tr>
<td>5.7.4</td>
<td>Ambiguities in the superset grammar</td>
<td>186</td>
</tr>
<tr>
<td>5.8</td>
<td>Back-tracking</td>
<td>187</td>
</tr>
<tr>
<td>5.8.1</td>
<td>Linear search in yacc</td>
<td>187</td>
</tr>
<tr>
<td>5.8.2</td>
<td>Binary tree search in yacc</td>
<td>189</td>
</tr>
<tr>
<td>5.9</td>
<td>FOG grammar</td>
<td>193</td>
</tr>
<tr>
<td>5.10</td>
<td>Code Structure</td>
<td>193</td>
</tr>
<tr>
<td>5.11</td>
<td>Grammar Metrics</td>
<td>195</td>
</tr>
<tr>
<td>5.12</td>
<td>Summary</td>
<td>199</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Files</td>
<td>201</td>
</tr>
<tr>
<td>6.1</td>
<td>Practical problems</td>
<td>201</td>
</tr>
<tr>
<td>6.2</td>
<td>File disposition</td>
<td>201</td>
</tr>
<tr>
<td>6.3</td>
<td>Utility</td>
<td>203</td>
</tr>
<tr>
<td>6.4</td>
<td>Dependency Analysis</td>
<td>204</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Create Usages</td>
<td>207</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Usage ordering</td>
<td>207</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Usage closure</td>
<td>208</td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.4</td>
<td>File usage ordering</td>
<td>208</td>
</tr>
<tr>
<td>6.4.5</td>
<td>File pre-ordering</td>
<td>208</td>
</tr>
<tr>
<td>6.4.6</td>
<td>File post-ordering</td>
<td>209</td>
</tr>
<tr>
<td>6.4.7</td>
<td>Emission</td>
<td>209</td>
</tr>
<tr>
<td>6.5</td>
<td>Target File Generation Policies</td>
<td>209</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Global namespace</td>
<td>209</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Friend functions</td>
<td>210</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Source File Protection</td>
<td>210</td>
</tr>
<tr>
<td>6.5.4</td>
<td>Suppressed Non-changes</td>
<td>210</td>
</tr>
<tr>
<td>6.5.5</td>
<td>Net dependencies</td>
<td>210</td>
</tr>
<tr>
<td>6.5.6</td>
<td>Pretty Printing</td>
<td>210</td>
</tr>
<tr>
<td>6.5.7</td>
<td>#line</td>
<td>211</td>
</tr>
<tr>
<td>6.5.8</td>
<td>Integrity</td>
<td>211</td>
</tr>
<tr>
<td>6.5.8.1</td>
<td>Checksum</td>
<td>212</td>
</tr>
<tr>
<td>6.5.8.2</td>
<td>Database</td>
<td>212</td>
</tr>
<tr>
<td>6.6</td>
<td>Summary</td>
<td>213</td>
</tr>
<tr>
<td>7</td>
<td>Examples</td>
<td>215</td>
</tr>
<tr>
<td>7.1</td>
<td>Idioms</td>
<td>215</td>
</tr>
<tr>
<td>7.1.1</td>
<td>InheritedTypedef</td>
<td>215</td>
</tr>
<tr>
<td>7.1.2</td>
<td>NoAssign, NoCopy</td>
<td>217</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Mutate</td>
<td>219</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Clone, Prototype</td>
<td>220</td>
</tr>
<tr>
<td>7.2</td>
<td>Patterns</td>
<td>221</td>
</tr>
<tr>
<td>7.2.1</td>
<td>StaticFlyweight</td>
<td>221</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Member</td>
<td>224</td>
</tr>
<tr>
<td>7.2.3</td>
<td>ReferenceCount</td>
<td>226</td>
</tr>
<tr>
<td>7.2.4</td>
<td>WholePart</td>
<td>230</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Visitor</td>
<td>233</td>
</tr>
<tr>
<td>7.3</td>
<td>Meta-Programming</td>
<td>235</td>
</tr>
<tr>
<td>7.3.1</td>
<td>OS Traits</td>
<td>235</td>
</tr>
<tr>
<td>7.3.2</td>
<td>SynchronisedEnum</td>
<td>236</td>
</tr>
<tr>
<td>7.3.3</td>
<td>RTTI</td>
<td>237</td>
</tr>
<tr>
<td>7.4</td>
<td>Aspects</td>
<td>239</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Marshalling</td>
<td>240</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Monitor</td>
<td>246</td>
</tr>
<tr>
<td>7.5</td>
<td>A Real Example - BURG</td>
<td>251</td>
</tr>
<tr>
<td>7.6</td>
<td>Summary</td>
<td>255</td>
</tr>
<tr>
<td>8</td>
<td>Summary</td>
<td>257</td>
</tr>
<tr>
<td>8.1</td>
<td>Parsing</td>
<td>257</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Context-free syntactical C++ parsing</td>
<td>257</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Back-tracking in yacc</td>
<td>257</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Superset grammar</td>
<td>258</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Semantic analysis restricted to semantics</td>
<td>258</td>
</tr>
<tr>
<td>8.1.5</td>
<td>Extended regular expressions</td>
<td>258</td>
</tr>
<tr>
<td>8.2</td>
<td>C++ Extensions</td>
<td>258</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Meta-programming</td>
<td>258</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Composition rather than One Definition Rule</td>
<td>259</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Minor extensions</td>
<td>259</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Derivation rules</td>
<td>259</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Syntax macros</td>
<td>259</td>
</tr>
<tr>
<td>8.3</td>
<td>Detailed Language Issues</td>
<td>259</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Scoped preprocessing</td>
<td>259</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Deferred substitution</td>
<td>259</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Polymorphic syntax</td>
<td>259</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Literal source</td>
<td>260</td>
</tr>
<tr>
<td>8.3.5</td>
<td>Potential and Actual</td>
<td>260</td>
</tr>
<tr>
<td>8.4</td>
<td>Further Work</td>
<td>260</td>
</tr>
<tr>
<td>8.5</td>
<td>Limitations</td>
<td>261</td>
</tr>
<tr>
<td>8.6</td>
<td>Other Languages</td>
<td>262</td>
</tr>
<tr>
<td>8.7</td>
<td>Resolution of Goals</td>
<td>262</td>
</tr>
<tr>
<td>9</td>
<td>Glossary</td>
<td>265</td>
</tr>
<tr>
<td>9.1</td>
<td>Acronyms</td>
<td>265</td>
</tr>
<tr>
<td>9.2</td>
<td>Terms</td>
<td>266</td>
</tr>
<tr>
<td>10</td>
<td>References</td>
<td>267</td>
</tr>
<tr>
<td>A</td>
<td>FOG Grammar changes</td>
<td>279</td>
</tr>
<tr>
<td>A.1</td>
<td>Keywords</td>
<td>279</td>
</tr>
<tr>
<td>A.2</td>
<td>Lexical conventions</td>
<td>280</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Phase 6 Concatenation Grammar</td>
<td>280</td>
</tr>
<tr>
<td>A.2.2</td>
<td>Phase 7 Tokenization Grammar</td>
<td>281</td>
</tr>
<tr>
<td>A.3</td>
<td>Basic concepts</td>
<td>282</td>
</tr>
<tr>
<td>A.4</td>
<td>Expressions</td>
<td>282</td>
</tr>
<tr>
<td>A.5</td>
<td>Statements</td>
<td>282</td>
</tr>
<tr>
<td>A.6</td>
<td>Declarations</td>
<td>283</td>
</tr>
<tr>
<td>A.7</td>
<td>Declarators</td>
<td>284</td>
</tr>
<tr>
<td>A.8</td>
<td>Classes</td>
<td>285</td>
</tr>
<tr>
<td>A.9</td>
<td>Derived Classes</td>
<td>286</td>
</tr>
<tr>
<td>A.10</td>
<td>Special member functions</td>
<td>286</td>
</tr>
<tr>
<td>A.11</td>
<td>Overloading</td>
<td>286</td>
</tr>
<tr>
<td>A.12</td>
<td>Templates</td>
<td>286</td>
</tr>
<tr>
<td>A.13</td>
<td>Exception Handling</td>
<td>286</td>
</tr>
<tr>
<td>A.14</td>
<td>Tree Literals</td>
<td>286</td>
</tr>
<tr>
<td>A.15</td>
<td>Object statements</td>
<td>287</td>
</tr>
<tr>
<td>A.16</td>
<td>Meta-Programming</td>
<td>288</td>
</tr>
<tr>
<td>A.16.1</td>
<td>Meta-names</td>
<td>288</td>
</tr>
<tr>
<td>A.16.2</td>
<td>Meta-classes</td>
<td>288</td>
</tr>
<tr>
<td>A.16.3</td>
<td>Meta-types</td>
<td>289</td>
</tr>
<tr>
<td>A.16.4</td>
<td>Meta-variables</td>
<td>289</td>
</tr>
<tr>
<td>A.16.5</td>
<td>Meta-functions, Meta-constructors and Meta-destructors</td>
<td>289</td>
</tr>
<tr>
<td>A.16.6</td>
<td>Meta-statements</td>
<td>289</td>
</tr>
<tr>
<td>A.16.7</td>
<td>Meta-expressions</td>
<td>290</td>
</tr>
<tr>
<td>A.17</td>
<td>Syntax macros</td>
<td>291</td>
</tr>
<tr>
<td>A.18</td>
<td>Files</td>
<td>291</td>
</tr>
<tr>
<td>B</td>
<td>C++ Grammar</td>
<td>293</td>
</tr>
<tr>
<td>C</td>
<td>FOG Grammar</td>
<td>313</td>
</tr>
<tr>
<td>D</td>
<td>Command Line</td>
<td>343</td>
</tr>
<tr>
<td>D.1</td>
<td>Miscellaneous options</td>
<td>343</td>
</tr>
<tr>
<td>D.2</td>
<td>Preprocessor options</td>
<td>343</td>
</tr>
<tr>
<td>D.3</td>
<td>Variant C++ options</td>
<td>343</td>
</tr>
<tr>
<td>D.4</td>
<td>Generated C++ options</td>
<td>343</td>
</tr>
<tr>
<td>D.5</td>
<td>Output file options</td>
<td>343</td>
</tr>
<tr>
<td>D.6</td>
<td>Diagnostic options</td>
<td>344</td>
</tr>
<tr>
<td>D.7</td>
<td>Predefined macros</td>
<td>344</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>E.1.1 auto</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>E.2.1 std</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>E.3.1 builtjn and built_in_type_specifier</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.3.12 enum and enumSpecifier</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.1 array_modifier</td>
<td>349</td>
<td></td>
</tr>
<tr>
<td>E.4.2 base and base_specifier</td>
<td>349</td>
<td></td>
</tr>
<tr>
<td>E.4.3 built_in and built_in_type_specifier</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.4 character</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.5 class and class_specifier</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.6 class_key</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.7 cv_qualifier</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.8 decl_specifier</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.9 declaration</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.10 elaborated_type_specifier</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.11 entity and specifier</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.12 enum and enum_specifier</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.13 enumerator and enumerator_definition</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.14 exception and exception_declaration</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.15 exception_specification</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.16 expression</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>E.4.17 filespace and filespace_specifier</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>E.4.18 function, function_modifier and function_specifier</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>E.4.19 handler</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>E.4.20 identifier</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>E.4.21 iterator</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>E.4.22 keyword</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.23 linkage and linkage_specification</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.24 meta_class and meta_class_specifier</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.25 meta_function and meta_function_specifier</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.26 meta_parameter and meta_parameter_specifier</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.27 meta_type</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.28 meta_variable and meta_variable_specifier</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.29 modifier</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.30 name</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.31 namespace and namespace_definition</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.32 namespace_alias and namespace_alias_definition</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>E.4.33 number</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.34 object and object_specifier</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.35 object_statement</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.36 parameter and parameter_specifier</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.37 pointer_modifier</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.38 punctuation</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.39 reference_modifier</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.40 reserved</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.41 scope and scope_specifier</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.42 scoped_modifier</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>E.4.43 specifier</td>
<td>354</td>
<td></td>
</tr>
<tr>
<td>E.4.44 statement</td>
<td>354</td>
<td></td>
</tr>
<tr>
<td>E.4.45 string</td>
<td>354</td>
<td></td>
</tr>
<tr>
<td>E.4.46 template_parameter and template_parameter_specifier</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>E.4.47 token</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>E.4.48 type and type_specifier</td>
<td>356</td>
<td></td>
</tr>
<tr>
<td>E.4.49 typedef and typedef_specifier</td>
<td>356</td>
<td></td>
</tr>
<tr>
<td>E.4.50 typename</td>
<td>356</td>
<td></td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

E.4.51 using and using_declaration ......................................................... 356
E.4.52 using_directive................................................................. 356
E.4.53 variable and variable_specifier ..................................................... 356
E.4.54 void .............................................................................. 356

F
Implementation ................................................................. 357
F.1 Syntax Implementation.......................................................... 357
F.1.1 !const and !volatile ...................................................... 357
F.1.2 Member variable delegation ............................................... 358
F.1.3 Pattern names .................................................................. 359
F.1.4 #line directive .................................................................. 360
F.2 Resolution of parsing difficulties ............................................... 360
F.2.1 Context-free problems .......................................................... 360
F.2.1.1 #include anomaly .......................................................... 361
F.2.1.2 Type information .......................................................... 362
F.2.1.3 < as template-start or greater than (§14.2-3) ......................... 362
F.2.1.4 > as template-end or greater than (§14.2-3) ......................... 362
F.2.1.5 Meta-types .................................................................. 363
F.2.2 Trivial Ambiguities ............................................................. 363
F.2.2.1 Empty statement .......................................................... 363
F.2.2.2 Template parameter ...................................................... 363
F.2.3 Syntactic Ambiguities .......................................................... 364
F.2.3.1 Dangling else (§6.4.1) ...................................................... 364
F.2.3.2 < as template-start or less than (§14.2-3) ......................... 364
F.2.3.3 Multiply nested scope (§7.1-2) ........................................ 364
F.2.3.4 new-type-id (§5.3.4-2), conversion-function-id (§12.3.2-4) ....... 365
F.2.3.5 Array of operator ambiguity ............................................ 365
F.2.4 Deep Ambiguities ................................................................. 366
F.2.4.1 Bit-field or inheritance ...................................................... 366
F.2.4.2 Type I functions ............................................................. 367
F.2.5 Semantic Ambiguities ............................................................. 367
F.2.5.1 Declaration/Expression ambiguity (§6.8) ......................... 367
F.2.5.2 Parenthesised-call / cast-parenthesis ................................... 368
F.2.5.3 Call/functional-cast ambiguity ........................................ 369
F.2.5.4 Destructor name/one's complement ambiguity (§5.3.1-9) ........ 369
F.2.5.5 new-placement/new-initializer ambiguity ......................... 369
F.2.5.6 sizeof ambiguity ............................................................ 369
F.2.5.7 typeid ambiguity ............................................................ 370
F.2.5.8 Template argument type/value ambiguity ......................... 370
F.2.6 New C++ ambiguities .......................................................... 370
F.2.6.1 ctor-initializer or Named Bit-field ....................................... 371
F.2.6.2 class-specifier or enum-specifier as function-definition name .......................................................... 371
F.2.6.3 delete[] ambiguity .......................................................... 371
F.2.6.4 linkage-specification ambiguity ......................................... 372
F.2.7 Extra FOG ambiguities ......................................................... 372
F.2.7.1 built-in-type-id maximised ............................................. 372
F.2.7.2 label preferred to anonymous bit-field ................................ 373
F.2.7.3 handler-seq maximised ...................................................... 373
F.2.7.4 access-specifier ............................................................. 373
F.2.7.5 inline/ and virtual/ cannot be expressions ......................... 373
F.2.7.6 using string-literal is not an expression ......................... 374
F.3 Semantic checks ................................................................. 374
F.3.1 Resolution ................................................................. 374
F.3.1.1 Declaration/Expression (§6.8) ................................................... 375
F.3.1.2 new-placement/new-initializer ambiguity ......................... 376
F.3.1.3 sizeof and typeid ambiguity ........................................... 376
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>F.3.1.4</td>
<td>pure-specifier</td>
<td>376</td>
</tr>
<tr>
<td>F.3.1.5</td>
<td>explicit-instantiation</td>
<td>376</td>
</tr>
<tr>
<td>F.3.1.6</td>
<td>Implicit int</td>
<td>376</td>
</tr>
<tr>
<td>F.3.2</td>
<td>Correction</td>
<td>377</td>
</tr>
<tr>
<td>F.3.2.1</td>
<td>Template/arithmetric ambiguity</td>
<td>377</td>
</tr>
<tr>
<td>F.3.2.2</td>
<td>Parenthesised-call / cast-parenthesis</td>
<td>377</td>
</tr>
<tr>
<td></td>
<td>parenthesised-binary / cast-unary ambiguity</td>
<td>377</td>
</tr>
<tr>
<td>F.3.2.3</td>
<td>Call_functional-cast ambiguity</td>
<td>377</td>
</tr>
<tr>
<td>F.3.2.4</td>
<td>Destructor name/one's complement ambiguity (§5.3.1-9)</td>
<td>377</td>
</tr>
<tr>
<td>F.3.2.5</td>
<td>Array of operator ambiguity</td>
<td>377</td>
</tr>
<tr>
<td>F.3.2.6</td>
<td>delete[] ambiguity</td>
<td>377</td>
</tr>
<tr>
<td>F.3.3</td>
<td>Validation</td>
<td>378</td>
</tr>
<tr>
<td>F.3.4</td>
<td>Implementation</td>
<td>378</td>
</tr>
<tr>
<td>F.4</td>
<td>File Syntaxes</td>
<td>378</td>
</tr>
<tr>
<td>F.4.1</td>
<td>Target File Names</td>
<td>378</td>
</tr>
<tr>
<td>F.4.2</td>
<td>Target File Identities</td>
<td>380</td>
</tr>
<tr>
<td>F.4.3</td>
<td>Target File Placement</td>
<td>380</td>
</tr>
<tr>
<td>F.4.4</td>
<td>File-spaces</td>
<td>381</td>
</tr>
<tr>
<td>F.4.5</td>
<td>Target File Dependencies</td>
<td>382</td>
</tr>
<tr>
<td>F.4.6</td>
<td>File inclusion</td>
<td>383</td>
</tr>
</tbody>
</table>

INDEX ...........................................................................................................385
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Meta-compilation model</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>C++ translation</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>FOG Meta-compilation Stages</td>
<td>84</td>
</tr>
<tr>
<td>4.2</td>
<td>Intrinsic Meta-type inheritance</td>
<td>88</td>
</tr>
<tr>
<td>4.3</td>
<td>Potential and Actual Meta-type inheritance</td>
<td>89</td>
</tr>
<tr>
<td>4.4</td>
<td>Example Potential Declaration Tree</td>
<td>90</td>
</tr>
<tr>
<td>4.5</td>
<td>Example Actual Declaration Tree</td>
<td>90</td>
</tr>
<tr>
<td>4.6</td>
<td>Example Tree initializer</td>
<td>93</td>
</tr>
<tr>
<td>4.7</td>
<td>int i = k</td>
<td>100</td>
</tr>
<tr>
<td>4.8</td>
<td>Meta-function and function meta-object distinction</td>
<td>104</td>
</tr>
<tr>
<td>4.9</td>
<td>Pure OO object model</td>
<td>129</td>
</tr>
<tr>
<td>4.10</td>
<td>C++ object model (Memory Perspective)</td>
<td>131</td>
</tr>
<tr>
<td>4.11</td>
<td>C++ object model (Naming Perspective)</td>
<td>132</td>
</tr>
<tr>
<td>4.12</td>
<td>C++ Run-time object model</td>
<td>133</td>
</tr>
<tr>
<td>4.13</td>
<td>FOG Compile-time object model</td>
<td>134</td>
</tr>
<tr>
<td>4.14</td>
<td>Inheritance and Meta-inheritance</td>
<td>136</td>
</tr>
<tr>
<td>5.1</td>
<td>Compiler Translation Stages</td>
<td>144</td>
</tr>
<tr>
<td>5.2</td>
<td>Abstract Syntax Tree</td>
<td>146</td>
</tr>
<tr>
<td>5.3</td>
<td>C++ Names</td>
<td>158</td>
</tr>
<tr>
<td>5.4</td>
<td>FOG analysis phases</td>
<td>194</td>
</tr>
<tr>
<td>6.1</td>
<td>Usage Dependency Graph</td>
<td>206</td>
</tr>
<tr>
<td>7.1</td>
<td>Pattern Class Friendships</td>
<td>231</td>
</tr>
<tr>
<td>7.2</td>
<td>Visitor classes</td>
<td>233</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Statement and Declaration Grammar</td>
<td>61</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Intrinsic Meta-types</td>
<td>91</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Potential and Actual Meta-types</td>
<td>92</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Terminals and Non-Terminals</td>
<td>155</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Mathematical Operators</td>
<td>156</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Grammatical Operators</td>
<td>157</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Properties</td>
<td>159</td>
</tr>
<tr>
<td>Table 5.5</td>
<td>Back-tracking costs</td>
<td>192</td>
</tr>
<tr>
<td>Table 5.6</td>
<td>Grammar Statistics</td>
<td>196</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Token size reduction through use of FOG</td>
<td>252</td>
</tr>
<tr>
<td>Table F.1</td>
<td>File Name Component Contributions</td>
<td>379</td>
</tr>
</tbody>
</table>
1 Introduction

Improved languages and improved methodologies jointly contribute to improved programming practice and performance. However, the two contributions are not balanced. Whereas a language extension supports a better methodology, a novel methodology may demand a language change.

The advent of practical support for Object Oriented Programming in C++ has made OO extremely popular, the most forward looking, although not yet the dominant industry programming style.

Extensive use of an OO style in C++ reveals its limitations and demands improvements. An OO style results in a large number of classes whose structure and relationships are in some way predictable, often conforming to a number of design patterns. It can be difficult to capture the predictability, so that the programming intent is expressed compactly, reliably and re-usably, indeed it is sometimes difficult to express the intent in more than comments.

An OO style is well suited to a program organisation based on a data-centric file structure, but some problems are better modularised from an algorithmic or Aspect Oriented perspective. C++ lacks facilities to weave a variety of contributions together to produce the required composite program.

We address these limitations and resolve them by introducing the run-time meta-programming capabilities of languages such as CLOS and Smalltalk as compile-time capabilities in C++, without losing the fundamental run-time efficiency or deviating too far from the language. At the same time, we replace the historical anomaly that is the C preprocessor to create a more powerful and integrated programming language.

The solutions, which are embodied in an enhanced C++ language and compiler called FOG (Flexible Object Generator), form the topic for this thesis. Implementation of FOG required the development of a new approach to parsing C++ without the use of semantic information. This work is presented in Chapter 5 so that it may read in isolation by those only interested in C++.

Background

C++ [C++98] is a popular, very widely used and successful industrial strength language with support for Object Orientation. The popularity of C++ is in part attributable to a very high degree of compatibility with C [C89]. Portability and run-time efficiency are some of C's and consequently C++'s attractions to programmers.

Efficiency is achieved in C by providing programming constructs that are relatively low level. Efficiency is preserved in C++ by using a restrictive form of object model that enables C++ to resolve at compile-time what many other Object-Oriented languages resolve at run-time.

Programmers as well as compiler writers seek to trade run-time for compile-time activity. A programmer may improve run-time efficiency by identifying better algorithms, selecting a more efficient compiler, or structuring code to exploit the good, or avoid the poor, characteristics of a compiler. In order to improve compile-time, there appear to be few approaches, although different coding styles and appropriate management of include file dependencies and compilation unit sizes can show surprising benefits.

The inadequacies of C support for compile-time programming were recognised from the outset by the provision of Cpp (the C preprocessor). C++ has introduced more powerful language constructs but provides no assistance for programmers who need to capture predictable programming structure that does not correspond to language constructs.
Introduction

Programming style, patterns and aspects

Dramatic improvements in programming time (and maintainability and reliability) may be achieved when an automatic code generator such as lex or yacc is applicable, or when an application generator such as a GUI builder is suitable. For many more mundane programming applications, the structure of the code is in some way predictable, but not of sufficient size or complexity to justify the development of a custom code generator. For these applications, the programmer is forced into workarounds, exploiting whatever tools are available. These workarounds often require indirect or repeated expression of the programming intent, introducing:

- maintenance problems through lack of clarity
- inefficiencies through the need for repeated editing
- errors through inconsistent repeated editing.

C++ was for a long time the main tool available to C and C++ programmers. And, prior to the introduction of templates, it was standard practice to use some very large preprocessor macros to define generic classes for containers. Templates now provide a powerful solution to problems that can be characterised by the requirement to define a family of types or functions. However, for many other problems, C++ remains the only alternative. Lexical pasting using the preprocessor is inelegant and not without its problems, but it is less error prone than manual approaches.

Compatibility with C required C++ to preserve C++, although its limitations as a programming tool have long been recognised (and are summarised in Section 3.4). While C++ introduces a number of new constructs that eliminate some traditional uses of C++, other uses remained. The power and complexity of Object Orientation and the increasing use of simple patterns [Gamma95], clichés [Gill97] or idioms [Coplien92] considerably increases the need for programmers to program at compile-time and as a result C++ is perhaps more, rather than less, important to C++ than to C.

C++ should be replaced rather than eliminated.

The Object Oriented Programming community has recognised that groups of objects with a shared behaviour can be found within apparently dissimilar applications. These shared behaviours are classified by patterns. There is no precise definition of what constitutes a pattern, but it is generally agreed that a pattern is a solution to a recurring problem in a context. Recurrence is an important discriminant between generic patterns and candidate patterns, which may be just special purpose tricks. A pattern is used in a specific context which imposes constraints (or forces) that influence the way in which the pattern solves the problem. The description of a pattern provides a range of solutions and a discussion of how the differing forces from the application context may influence the usage. The generality of the pattern concept allows the pattern form to be very widely applied: from project management through to idiomatic coding. This generality is a little frustrating since the literature lacks focus.

Analysis, design and coding patterns are of significance to this work. Coding patterns tend to be simple program idioms that implement standard coding practices that are not directly supported by the target language. Design patterns capture the collaborations between implemented objects. Analysis patterns similarly capture collaborations, but at a more abstract level reflecting the higher level analysis perspective. The extent to which patterns used during analysis, design or coding appear in the implementation is a rather contentious issue. Automated implementation of patterns would seem attractive but Misconception 4 of [Vissides98] strongly cautions against this.

From the purist perspective, compromising between the generally conflicting forces unique to each application requires careful selection between a wide range
of possible solutions. It is inappropriate and impossible to offer a cookbook solution to a pattern, partly because such a solution cannot offer sufficient flexibility. In many applications, more than one pattern is employed and where the patterns overlap, the solution must be adapted to share rather than duplicate the overlapping functionality.

From the practical perspective, an inferior set of proven cookbook solutions is often preferable to optimal handcrafted solutions. Programmers tend to implement solutions they are familiar with, rather than those that could be more optimal. If patterns have been used as part of the analysis and design phases, it seems appropriate for these patterns to find some form of expression in the code. Otherwise, if the patterns are not expressed at all, the patterns are lost [Soukup94] and subsequent code maintenance is hampered by greater barriers to comprehension. When patterns are expressed only in the form of comments, compliance with design principles and constraints is informal; no enforcement occurs during implementation or subsequent maintenance.

The challenge is to provide cookbook solutions with sufficient parameterisation to satisfy the purists, while offering adequate efficiency and utility for practical requirements. Enthusiastic use of templates, as practised in Generative Programming [Czarnecki97] can result in highly configurable functions and to some extent types and components. However much of the required parameterisation involves appropriate selection and configuration of declarations in ways not amenable to template programming. Program configuration in C++ should occur at compile-time, but the facilities of the C preprocessor are inadequate for the task and a generally inappropriate foundation for this new programming paradigm.

**Compile-time programming is necessary to configure declarations.**

A problem with implementing patterns is that a pattern tends to involve more than one class, and so use of a pattern requires code to be added to more than one class. A similar but larger scale problem arises in Aspect Oriented Programming [Mens97], in which an aspect is a programming concern (such as error recovery) that cuts across more than one class. AO programs organise source code according to the programming concerns and then use a weaver to combine the disparate contributions into a complete program.

**Patterns and AOP require weaving.**

The C++ One Definition Rule ([C++98] §3.2) mandates a single definition of each declaration, with the result that with a few exceptions (forward references, externs, typedefs, and namespaces) C++ source code must be organised to satisfy the constraint of complete declarations. It is not possible to interleave class declarations. This might seems like a desirable restriction, until it is appreciated that this prevents weaving in support of AOP or even support for a multi-class pattern solution.

**The One Definition Rule must be circumvented.**

**Meta-programming**

Prior to C++, Object Orientation, as then exemplified by Smalltalk, was perceived to be inherently inefficient because of the run-time costs associated with message dispatch. C++ introduced a more restrictive object model that enabled most of the run-time costs to be replaced by compile-time computation. As a result Object Orientation in C++ is efficient and widely used. (More efficient implementation approaches developed to make Smalltalk costs more acceptable are now being exploited by Java).

C++ requires that the layout of objects be frozen at compile-time, and that the (base) type of the recipient of any message is known. The layout constraint implies a single contiguous memory allocation for each object, simplifying memory
management and providing member variable access by a simple indexing operation. The messaging constraint enables static and some dynamic methods to be implemented as simple function calls. The remaining dynamic methods require a virtual function that is implemented by a single indirection from a known index into a relatively small dispatch table. These are pragmatic constraints on the object model. Elimination of run-time object flexibility removed the need for run-time code to manipulate object structure, and for run-time objects to describe it. The meta-classes that are essential for languages such as Smalltalk were therefore not necessary for C++, and so they are not part of the C++ language.

It has been found that some degree of self-awareness is useful to an Object Oriented program. This may involve

- a knowledge of class names for diagnostic purposes
- availability of inheritance information as Run-Time Type Information to validate dynamic casts
- object layout information to support marshalling for communication
- object layout information for persistent storage in data bases
- full class descriptions for browsers or debuggers

The first two of these needs have been addressed as C++ has evolved from ARM [Ellis90] to ISO standard [C++98]. Applications requiring more substantial information must resort to special purpose pre- or post-processing. Reflection supports this extra processing directly as part of the compilation process.

Introspection is useful for simple applications.
Reflection is almost essential for sophisticated applications.

When a Smalltalk or CLOS program reflects upon itself, this necessarily happens at run-time, since this is when object structure is defined. Support for reflection is relatively easily provided by formalising the interface to the underlying run-time language support.

In C++, objects are defined at compile-time, and so an opportunity exists for a program to reflect upon itself at compile-time as well as, or instead of, at run-time. If the purpose of that reflection is just to extract some information or perform some checking in a one-off fashion, it is clearly preferable for such code to execute at compile-time. This is very much in the C++ spirit of maximising run-time performance by resolving as much program structure as possible at compile-time. C++ only optimises those constructs that form part of C++. Reflection supports optimisation of user defined concepts.

If reflection is to happen continuously, then it must occur at run-time. The C++ philosophy dictates that unwanted language functionality should not impose run-time costs, so provision of run-time reflection must be cost free, when unused. The amount of run-time reflection may vary between applications, with the majority not using it at all. Some may wish to just browse data structures describing declarations. Very sophisticated applications may seek to reify the different stages of message dispatch to validate argument lists or call-from access rights. The required support for run-time reflection can be achieved by using compile-time reflection to create data structures and modify code to collaborate with a run-time support environment. The degree of support can be tailored to match the requirements.

It is not entirely clear that compile-time modification of executable code is necessary for many practical applications, and so we consider only perusal and modification of declarations. Modification of code is nevertheless possible, provided the intended target for modification is encapsulated within a template or inline function, which can then be modified as a declaration.

1. To make a thing, typically by creating an object to represent an abstract concept.
1.1 Language Limitations

The work described in this thesis grew out of a recognition that the programming effort for C++ programs increased disproportionately with the size of the program. This is not particularly unexpected. Large programs have been a consistent problem for software developers. Solutions have been found through improved languages and programming methodologies, with C++ and Object Orientation making their contributions. The concepts of patterns and Aspect Orientation offer further improvements to programming methodologies but highlight constraints imposed by C++. We therefore assess and resolve these limitations.

Declarative Redundancy

C++ exhibits significant lexical redundancy.

There is duplication between many interface and implementation declarations. This requires a duplication of editing effort and provides a limited opportunity for inconsistencies to introduce errors.

In a deep inheritance hierarchy, the same virtual function may have many implementations, again requiring repeated editing effort. This effort is most noticeable when it is necessary to change the function signature: two additional edits may be needed for each derived class to realise what is a single conceptual change. More seriously, an inconsistency between declarations within the hierarchy is not necessarily an error, although a helpful compiler may choose to flag the hazard.

*Lexical redundancy should be eliminated.*

Algorithmic Redundancy

Implementation of many idioms imposes a well-defined protocol that must be observed by derived classes. C++ provides no mechanism for implementation of this protocol, although in some cases use of a pure virtual function may diagnose a non-implementation. Correct observance of the protocol requires implementors of derived classes to add the code manually, sometimes making use of preprocessor macros. This practice is at best tedious. It makes derivation from third party libraries unnecessarily difficult because application writers have to supply extra library support code.

*Predictable code should be provided automatically.*

Organisational Restrictions

Implementation of more interesting idioms and patterns requires code injection into multiple classes. This is not supported in C++. A particular solution to a pattern requires an interface and an implementation preprocessor macro for each collaborator. These macros are typically invoked from the interface and implementation files of each class. As a result a pattern involving 3 classes may require 6 macros. These 6 macro invocations that instantiate a single concept are dispersed throughout the source code.

*A concept should be instantiated by a single invocation.*

For some applications, it is appropriate to partition the source code according to the data structures: algorithmic code is then naturally assigned to classes. For other applications, in particular Aspect Oriented applications, the algorithmic perspective may be more important, and so all the code for one algorithm or aspect should be kept in one source module, while that for another should be in another module. In this situation the aspect cuts across the class structure. C++
requires sequential declaration of complete classes. It is not possible for partial declarations to be interleaved.

*Interleaved declarations should be allowed.*

**Summary of Problems**

*Cpp should be replaced rather than eliminated.*

*Compile-time programming is necessary to configure declarations.*

*Patterns and AOP require weaving.*

*The One Definition Rule must be circumvented.*

*Introspection is useful for simple applications.*

*Reflection is almost essential for sophisticated applications.*

*Lexical redundancy should be eliminated.*

*Predictable code should be provided automatically.*

*A concept should be instantiated by a single invocation.*

*Interleaved declarations should be allowed.*

**The FOG Solution**

These problems require revision of C++, and so in order to avoid development of a new language, or a new compiler for a modified language, revisions were implemented within a translator from extended C++ to standard C++.

The experimental translator is called the Flexible Object Generator (FOG) and the extended language is FOG. The translator

- revises C++ (upward compatibly, with no new reserved words)
- rearranges source code
- synthesizes declarations
- interprets meta-programs

The most significant revision is the relaxation of the One Definition Rule to support composition of multiple declarations. This is a major semantic enhancement, but it is almost invisible syntactically. This revision opens the door to weaving and pattern implementation.

The facilities of the C preprocessor are essential for practical programming, but integrate very poorly with C or C++. Many of the FOG extensions provide replacements for Cpp functionality, thus meta-functions replace function-like macros, tree-literals invite a replacement avoiding the accidental substitutions characteristic of Cpp, and meta-statements support conditional compilation. Consistent generalisation of each of these concepts results in a compile-time environment in which meta-programs can be interpreted.

**1.2 Organisation**

An overview of FOG is provided in this chapter, so that the later full exposition of the FOG grammar can provide examples less constrained by forward referencing.

Some related work has already been mentioned as part of the motivation for this work. Chapter 2 contains a more extensive review and comparison.

The main description of FOG is provided with Chapter 3 covering the lexical and syntactical enhancements as the foundation for the more substantial discussions of the semantics of substitution, composition and meta-classes in Chapter 4. The changes are described one at a time in Chapter 3 interspersed with discussion. The modified grammar is therefore repeated in Appendix A for ease of comparison with Annex A of [C++98].
A context-free grammar is important for flexible meta-programming, and it is well-known that the C++ grammar is not context-free. This would appear to preclude context-free meta-programming. Chapter 5 examines existing parsing approaches, and introduces a new approach that supports context-free syntactical analysis of C++. The validity of the new approach is shown by using an extended form of regular expression to analyze the C++ grammar and deduce the ambiguities. A working yacc-able implementation of the C++ grammar alone is presented in Appendix B, with the full FOG version in Appendix C.

Operation of FOG as a translator to C++ involves a number of practical concerns regarding file locations, partitioning of declarations into files and generation of appropriate include file dependencies. These issues and a description of the algorithms required for code emission are described in Chapter 6.

A number of small examples are provided as part of the overview and enlarged upon together with a few slightly larger ones in Chapter 7.

The achievements of FOG are summarised in Chapter 8, followed by a glossary of significant terms and acronyms and a list of all references.

The FOG command line is described in Appendix D.

The preliminary catalogue of built-in meta-functions may be found in Appendix E.

Detailed descriptions of
- discarded syntax
- parsing ambiguity resolution
- C++ ambiguities introduced by the superset grammar approach
- FOG ambiguities introduced by the superset grammar approach
- file placement and dependency syntax
are presented in Appendix F.

1.3 Conventions

Before we provide an overview of the FOG functionality, we must define the typographical conventions used throughout the rest of the text.

The FOG grammar is a superset of C++ and so it is necessary to make occasional reference to the C++ grammar as defined by [C++98] and then summarised in its Annex A, which is very similar to Appendix A of [Stroustrup97]. In order to save the reader having to keep a copy to hand, relevant sections are included in the text and in Appendix A. Specific paragraphs are referenced as §11.4, or more commonly as §11.4-5 where the 5 denotes the numbered paragraph within the section numbered 11.4 in [C++98].

The BNF-like (Backus Naur Form) language of the C++ standard is used in this document. Terminals (such as static) are distinguished by the use of a typewriter font. Non-terminals (such as parameter-declaration-clause) are in an italicised serifed font. A production (rule) comprises a non-terminal on its left-hand side followed by a colon followed by the right-hand side. Productions that share a common left-hand side are grouped together with one right-hand side per line. Optional elements are denoted by an opt suffix. Comments may be supplied following a //. Thus

```plaintext
base-specifier:
  : :opt nested-name-specifier_opt opt class-name  // defaults to private
virtual access-specifier_opt ::opt nested-name-specifier_opt opt class-name
access-specifier virtual ::opt nested-name-specifier_opt opt class-name
```

comprises the 3 rules that define the syntax of a base-specifier. (The standard is a little lax in its formatting of these productions, neglecting to use typewriter font for the virtual keyword or the :: punctuation.)
In order to ease comparison of similar FOG and C++ grammar, the two are combined with a strike-through to denote C++ constructs removed in FOG, and an underline to denote FOG constructs added to C++. Thus the C++ rules

`primary-expression:`
- `literal`
- `this`
- `: : ` identifier
- `: : ` operator-function-id
- `: : ` qualified-id
- `{ expression }
- `id-expression`

and the replacement FOG rules

`primary-expression:`
- `literal`
- `this`
- `{ expression }
- `declarator-id`

may be shown in combination as

`primary-expression:`
- `literal`
- `this`
- `++ ` identifier
- `++ ` operator-function-id
- `++ ` qualified-id
- `{ expression }
- `id-expression`
- `declarator-id`

When a strike-through or underline is applied to a shared left-hand side rather than a rule, the strike-through or underline applies to all rules, but is omitted in the interests of readability.

`using-declaration:`
- `using typename_opt : : opt nested-name-specifier unqualified-id ;`
- `using : : unqualified-id ;`

Application of a strike-through to the left-hand side (definition) non-terminal implies application of a strike-through to all references of the non-terminal as well.

C++ grammar productions are generalised in FOG, but are not given different meanings. There is therefore no ambiguity in referring to the grammar production for a declaration as `declaration`. Italics in normal text denote a non-terminal.

When it is necessary to show examples of grammar implementation rather than specification, typewriter font is used throughout. Non-parametric terminals are spelled out in single or double quotes (" *" or "class"). Parametric terminals are shown in mixed-case (StringLiteral). Non-terminals are shown in lower case (base_specifier).

Multi-character terminals such as "::" or "virtual" are a non-standard extension and are not supported by yacc or bison. They are used in the main text for clarity. They are not used in the grammars in Appendix B or Appendix C where an upper-case lexical token such as SCOPE or VIRTUAL is used.

### 1.4 Traditional Preprocessing

This overview of FOG starts by showing how the basic facilities of Cpp are replaced, using very simple examples, that are gradually reworked as more powerful facilities are described and exploited.
1.4.1 Lexical substitution

Lexical substitution enables common definitions to be shared, given sensible names, and factored out if alternative definitions are needed in different contexts. When used responsibly, this leads to a considerable improvement in code quality, and is one of the main reasons for the widespread use of the preprocessor. However it is very easy for unfortunate substitutions to occur, and the presence of all names from all header files in a single name space is a source of many problems.

C++ has removed the need for many substitutions by the introduction of initialised consts and scoped enumerations. However, even where these are appropriate, the need for a non-integral type may defeat C++ enhancements.

Problems with Cpp substitution stem from the single namespace and from forceful substitution irrespective of context. Resolution of the namespace problem in FOG will be dealt with later. The problem of over-enthusiastic substitution is resolved by changing to a policy of substitution by invitation, rather than substitution by imposition. In FOG, instantiation of the definition of NAME is invited by $NAME, with the fallback of $$(NAME) when subsequent characters could cause an unwanted meaning. The increased safety incurs the cost of the trigger character(s) to invite the substitution. These characters are not too out of place in a cryptic language such as C. The syntax should be familiar to Unix shell or make programmers.

1.4.2 Name concatenation

Name concatenation is useful for generating a new name derived from some stem. Thus an implementation of the NullObject pattern [Martin97] may automatically define a Null class derived from its AbstractObject by suffixing Null to the class name of the AbstractObject.

class AbstractObjectNull : public AbstractObject
{ /*...*/ };

This can be realised directly in FOG, where unseparated identifiers and literals (numbers, strings and characters) are concatenated

class ${ABSTRACTOBJECT}Null : public ${ABSTRACTOBJECT} { /*...*/ };

Cpp provides the ## concatenation operator that can only be used within a macro:

#define NULLOBJECT_INTERFACE(ABSTRACTOBJECT) \
    class ABSTRACTOBJECT ## Null : public ABSTRACTOBJECT { /*...*/ };

1.4.3 String conversion

It is sometimes necessary, particularly for diagnostic purposes, to use a name as both an identifier and a string.

const char *Class::class_name() const { return "Class"; }

This may be expressed directly in FOG, exploiting concatenation of an empty string to perform a lexical cast, since the result of a concatenation is of the same kind as the first contribution.

const char *${CLASS}::class_name() const { return "${CLASS}"; }

Cpp provides the # operator for use within macros.

#define CLASS_NAME_IMPLEMENTATION(CLASS) \
    const char *CLASS::class_name() const { return #CLASS; }

1.4.4 Text replacement

The preprocessor #define directive is used to define object-like macros
#define PI 3.14159
and function-like macros
#define max(a,b) ((a) > (b) ? (a) : (b))
supporting usage as
a = max(sin(2*PI*f), 0.5)

The replacement text is an arbitrary sequence of preprocessor tokens that are
substituted without regard to context. Errors, particularly in nested definitions, are
difficult to diagnose, because substitution occurs before any language
interpretation is applied; few compilers or debuggers support tracing back to the
source once substitution has occurred. Long definitions require the use of
backslashed continuation lines, which are inconvenient and unreliable to edit or
read. Readability is further impaired by the need to use parentheses to guard
against the possibility of accidental association problems.

FOG provides a meta-level where conventional run-time concepts can be used at
(meta-)compile time. Meta-variables replace object-like macros and meta-
functions replace function-like macros. Meta-variables and meta-functions are
declared and typed in a very similar way to normal C++ variables and functions,
save for the new use of the auto keyword and the introduction of meta-types:

```
auto double PI = 3.14159;          // Meta-variable
auto expression max(expression a, expression b)   // Meta-function
{
    $a > $b ? $a : $b;
}
```

for use as
```
a = $max(sin(2*$PI*f), 0.5)
```

The auto keyword is almost totally obsolete in C++, where auto is only permitted
within functions. auto is reused outside of functions in FOG to declare meta-
functionality. Readers may choose to pronounce auto as meta, throughout this
thesis.

The meta-types correspond to the basic kinds of token (identifier, string and
character), the numeric types (bool, double, int and etc.) and also to
productions such as declaration and expression from the C++ grammar.

Use of meta-types enables the parser to ensure that arguments are passed and
returned compatibly, and to diagnose errors more helpfully. When appropriate,
conversions between the basic kinds are performed automatically.

Substitution within the meta-function replaces each invocation by its
corresponding argument expression.

The simple meta-function implementation of max solves the parenthesis problem,
works for arbitrary types but remains prone to side effects. The invocation
```
$max(a++, b++)
```
will result in one argument receiving a double increment just as in Cpp.

### 1.4.5 Conditional compilation

Conditional compilation is essential to support a variety of configuration options,
often to resolve distinctions between different operating systems. It may be
appropriate to define
```
static const char *temp_path = "~/tmp/";
```
for use under Unix whereas NT might require
```
static const char *temp_path = "C:\\Temp\"
```
FOG elevates C++ run-time statements such as if ... else ... for use at the meta-level, so that the selection may be made using an apparently conventional test:

```cpp
auto if ($UNIX)
    static const char *temp_path = "/tmp/";
else
    static const char *temp_path = "C:\\Temp\";
```

C++ provides line-oriented conditional directives that mark-up rather than form part of the source text:

```cpp
#if defined(UNIX)
    static const char *temp_path = "/tmp/";
#else
    static const char *temp_path = "C:\\Temp\";
#endif
```

C++ statements occur only within functions. The use of the `auto` prefix in FOG is therefore redundant in the above example. However since the prefix makes meta-code easier to distinguish, the prefix will be used throughout this thesis.

1.5 Object-Oriented preprocessing

The facilities described above provide consistent replacement for Cpp behaviour. Most of the extensions could be regarded as extensions to C rather than C++. Reviewing and generalising the facilities within the context of C++ leads to a much more powerful programming environment in which predictable program structures can be coded effectively.

1.5.1 Scopes

Meta-variables and meta-functions may be scoped and inherited, and meta-statements may occur within declaration scopes.

Revisiting the conditional compilation example of Section 1.4.5 from an Object-Oriented perspective, we find no need for conditional compilation. The characteristics of each configuration option may be packaged as meta-variables (and meta-functions) of a (meta-)class.

```cpp
class OsTraits_Abstract
{
    auto bool NT = false; // default value
    auto bool UNIX = false;
    //...
};
class OsTraits_NT : public OsTraits_Abstract // derived class
{
    auto bool NT = true;
    auto string temp_path = "C:\\Temp\";
    //...
};
class OsTraits_U X i n : public OsTraits_Abstract
{
    auto bool UNIX = true;
    auto string temp_path = "/tmp/";
    //...
};
```

OsTraits_NT may be configured as the implementation of OsTraits, by specifying the value of `OS` on the FOG command line

```
fog ... -D OS=NT ...
```

and using the built-in meta-function
auto string std::get_cpp(string macroName)
to access it from
class OsTraits : public OsTraits_$std::get_cpp("OS") {};
thereby creating the equivalent declaration
class OsTraits : public OsTraits_NT {};
This maps the required configuration to OsTraits, so that an operating system
specific file may be defined using the temporary path by
const char *fileName = $OsTraits::temp_path "results.dat";
This is then resolved at compile-time to
const char *fileName = "C:\Temp\results.dat";
Having isolated the configuration in separate classes and an associated header
file, a new operating system can be supported by providing a prefix file
characterising the new system and invoking it with an appropriate command line.
Existing source files need no change. This could be achieved directly using
multiple layers of name substitutions with C preprocessor, but it never is.
Modularization is much easier when supported by the programming environment.
This cannot be achieved using C++ templates, which lack the ability to perform
string manipulations.

1.5.2 Joint Interface and implementation

Introduction of a meta-compiler that synthesises interface and implementation
files eliminates the need for independent interface and implementation
declarations. It is appropriate to generalise C++ declarations to remove the
distinction between interface-specific and implementation-specific declarations.
This generalisation turns out to be almost entirely semantic, since the C++
grammar already permits an interface-specific keyword such as virtual to
accompany a function-definition. It is only necessary to allow an access-specifier (e.g.
protected) as part of a decl-specifier (the type part of a declaration), and to permit
a full id-expression (e.g. Scope::name) where previously only an identifier was
allowed.
Programmers may then use an implementation style of declaration for parts of
interfaces
public typedef size_t Class::SizeType;
or provide complete implementations in interfaces:
class Class
{
    protected virtual void f(int x = 0) = 0 { std::cout << x; }
public:
    static double y = 0;
};
A complete solution to the class_name() example from Section 1.4.3 may now be
captured by the single meta-function
auto declaration ClassName()
{
    public virtual !inline const char *class_name() const
    { return ""$Scope; }
};
which can be invoked as
class NamedClass
{
    $ClassName();
};
The reserved meta-variable Scope refers to the prevailing scope, avoiding the need to pass it as a parameter.
The negated keyword !inline ensures that the function body is not inlined. Similarly !static would provide for explicit rather than default programming intent.
The single meta-function invocation generates the equivalent C++ interface
class NamedClass
{ /* ... */
    public:
        virtual const char *class_name() const;
};
and implementation
const char *NamedClass::class_name() const { return "NamedClass"; }

This requires a pair of macros when implemented using Cpp.
#define CLASS_NAME_INTERFACE()
    virtual const char *class_name() const;
#define CLASS_NAME_IMPLEMENTATION(CLASS)
    const char *CLASS::class_name() const { return # CLASS; }
and a corresponding pair of invocations one from the interface
CLASS_NAME_INTERFACE()
and one from the implementation
CLASS_NAME_IMPLEMENTATION(NamedClass)

1.5.3 Composition

In C++, multiple declarations are an error. In FOG, multiple compatible declarations are composed; only incompatible declarations are an error. A brief summary of the composition rule is given here. A full exposition is provided in Section 4.4.

Composed declarations merge their components, and so a variable qualified with static carries the static with it when merged with another variable that has no static specification, but provokes an error message if merged with a !static.

Overloaded function declarations compose independently. Default values may be repeated but may not conflict.

Arrays and enumerations extend to accommodate all contributions. Duplicate initialisations must match. Holes in arrays are zero filled. The GNU C [Stallman98] extension is supported so that sparse arrays can be defined and composed.
The constructor initialisers for a particular constructor are composed and must not conflict. Unspecified initialisers for non-copy constructors are obtained from member variable initialisers. For example, code to support an error handling aspect may add a member variable with a default initializer:
public bool Class::_error_generated = false;
A constructor independently added in support of some other aspect
Class::Class(PersistenceManager&) /* ... */;
then provides the requisite initialisation.
Classes expand to encompass all distinct member declarations, with repeated declarations composed recursively.

Function (and constructor) bodies are composed by concatenating code contributions within named regions, which are in turn concatenated to form the overall function body. The regions named entry and exit typically provide for variable declaration and initialisation and a return statement, ensuring a predictable structure. Regions named pre and post provide code to operate before or after the default body region of the function. Function definitions are extended to support a declarative scope within which regions are prefixed by their name.

```cpp
class Manager
{
private:
    Aspect _aspect;
public:
    bool do_it();
};

bool Manager::do_it()
{
    bool exitStatus = true;
    if (!_aspect.do_it())
        exitStatus = false;
    return exitStatus;
}
```

FOG weaves the contributions together to produce the equivalent C++ declarations:

```cpp
class Manager
{
private:
    Aspect _aspect;
public:
    bool do_it();
};

bool Manager::do_it()
{
    bool exitStatus = true;
    if (!_aspect.do_it())
        exitStatus = false;
    return exitStatus;
}
```

Readers who have programmed extensively with a macro assembler may recognise that the ability to extend classes, function code regions, enumerations and arrays at will gives each declaration space the attributes of a program section. It is possible to define a meta-function that performs extension of an enumeration and a text array so that numeric and text declarations are automatically synchronised.

```cpp
auto declaration NamedEnum(identifier aName) {
    public enum Enum { $aName };
    public static const char *names[] = { "$aName" };
}
```

Invocation as
class Colours
{
$NamedEnum(RED);
$NamedEnum(GREEN);
$NamedEnum(BLUE);
};

provides successive entries for Colours::Enum and corresponding entries for Colours::names[], as if the user had typed:

class Colours
{
public:
    enum Enum { RED, GREEN, BLUE };
    static const char *names[];
};

and

const char *Colours::names[] = { "RED", "GREEN", "BLUE" }; 

The conversion of a single name such as RED into multiple interleaved declarations cannot generally be achieved using the preprocessor or C++ templates.

1.5.4 Derivation rules

There are many idioms that require entirely predictable code to be provided by derived classes in order to comply with a protocol defined by a base class. The class_name() method of Section 1.5.2 provides one example. In C++, a declaration applies to the scope for which it is specified. In FOG, this scope is referred to as the root scope for that potential declaration. An optional derivation rule specifies how that potential declaration may be automatically redefined in the inheritance tree of scopes that derive from the root scope to contribute to a number of actual declarations. Refining the example from Section 1.5.2:

auto declaration ClassName()
{
    public virtual !inline const char *class_name() const
    :
    {
        derived(true) { return "@Scope; }; 
    }
};

A declarative scope has been introduced by the :{ ... } to prefix a derivation rule to the function body. The predicate of derived(true) is always true and so the potential declaration is applied throughout the entire inheritance tree, that is at the root scope and all derived scopes.

The change of substitution operator from $ to changes the evaluation time. $ is an early substitution operator, evaluated when source tokens are first parsed to create a potential declaration in its associated root scope, at which point Scope resolves to the root scope. @ is a late substitution operator, evaluated when a potential declaration becomes an actual declaration in its eventual scope, at which point Scope resolves to the actual scope. (If the $ operator were used in the example, all derived scopes would return the name of the root scope.)

Derivation rules can apply to the declaration of any entity. Michael Tiemann provided a solution [Stroustrup94] to the problem of providing a mnemonic name for the primary base class

class foreman : public employee {
    typedef employee inherited;
    //...
    void print();
};
class manager : public foreman {
    typedef foreman inherited;
    //...
    void print();
};

enabling a derived class to refer to its base class mnemonically as inherited
rather than explicitly.

void manager::print()
{
    inherited::print();
    //...
}

In FOG, the entire hierarchy of typedefs can be expressed by a single declaration.

private typedef @Super employee::inherited
    :( derived(!is_root()); );

This provides a typedef for all derived classes. The derivation predicate inhibits
the declaration at the root, where the built-in meta-variable Super may have no
valid resolution for the primary base class.

The Prototype pattern [Gamma95], virtual constructor, or cloning idiom
[Stroustrup97] is also provided very easily using a derivation rule. The
conventional approach requires that a clone method be defined for every non­
abstract class in an inheritance hierarchy

class ConcreteClass /* ... */
{
    /* ... */
    virtual RootClass *clone() const;
};

RootClass *ConcreteClass::clone() const
    { return new ConcreteClass(*this); }

This requires the programmer to manually weave the code in to every class. This
is potentially error prone and costs at least one line per interface and one line per
implementation file of every class. Using FOG, the idiom can be fully defined by a
meta-function:

auto declaration Prototype()
{
    public virtual $Scope2 *clone() const = 0
    {
        derived(!Scope.is_pure()) { return new @{Scope}(*this); };
    }
}

The !Scope.is_pure() derivation predicate specifies that the declaration
contributes code to all derived classes that have no pure virtual functions.
Instantiation requires a single line in the base class that defines the protocol. No
code is required in derived classes.

class Base
{
    $Prototype();
};

These two examples demonstrate FOG at its most advantageous: one line in the
base class guarantees protocol observance and replaces one or more lines in
each derived class.

2. @{$Scope} may be changed to @Scope in the above example to use the derived type
as the return type.
1.5.5 Compilation model

C++ supports a two stage translation process involving multiple independent compilations followed by a link editing stage to produce a final executable. The independent compilations are consistent provided the One Definition Rule (§3.2) is observed. Simply stated, this rule requires that a declaration in one compilation must not have a different meaning in any other. In practice, placing declarations in header (interface) files, which are included by each compilation session that requires them, usually satisfies the One Definition Rule.

From the perspective of a compiler writer, the One Definition Rule is very useful, if not essential. From the perspective of the programmer, the One Definition Rule is very inconvenient. Declarations must be provided twice, once in the interface file and again in the implementation file. Declarations cannot be freely interleaved.

In more serious applications, a conflict arises between language constraints and the programmer's need to organise code to suit algorithmic or functional perspectives. Code has to be organised to suit the compiler. Patterns cannot be preserved in the code [Soukup94] and Aspect-Oriented Programming [Kiczales97] is not readily supported.

A preprocessor for C++, that performs its processing prior to compilation, can bridge the gap between the organisational requirements of the programmer and the integrity requirements of the compiler. FOG operates in this way using an augmented compilation model as shown in Figure 1.1.

![Figure 1.1 Meta-compilation model](image)

The centre and right hand sides show the conventional C++ compilation model. Interface files provide the declarations to be shared by independent compilations, which produce object files to be linked together with libraries to produce an executable. (The complexities of static construction and template instantiation are conveniently hidden by the 'Linker2' activity.) Meta-compilation adds the extra stages on the left hand side. The conventional C++ interface and implementation files are generated by one or more meta-compilations from source files (the forward arrows) and from frozen interfaces (the reverse arrows). Sources may be shared between meta-compilations, and a single meta-compilation may generate any number of interfaces and/or implementations.

Clearly the One Definition Rule must still be respected by the interface and implementation files fed to the compiler. However a more relaxed Composite Definition Rule can now be imposed on the source files. Simply stated, the composite meaning of all like declarations must be the same in each meta-compilation. The composite meaning is explained in Section 4.4.
Meta-compilation for C++

Introduction

1.5.6 Meta-concepts

Compile-time programming is often referred to as meta-programming, and data structures describing data are often referred to as meta-data. In particular, classes are described by meta-classes. We therefore refer to FOG as a meta-compiler, and generally use the meta-prefix to refer to a conventional run-time C++ concept elevated to compile-time, or more precisely the new meta-compile-time that precedes conventional compile-time. It is however often convenient to loosely refer to meta-compile-time and compile-time together as just compile-time.

Thus meta-classes describe classes, and meta-inherit from base meta-classes: classes which are in turn the meta-classes of the corresponding base classes. A meta-class has meta-members which are meta-functions and meta-variables with static and non-static variants. Statements that are interpreted at meta-compile-time are meta-statements. Declarations for use at meta-compile-time are meta-declarations. The type system available for use at meta-compile-time comprises meta-types.

Meta-compilation involves a two stage translation from FOG source text declarations to C++ declarations suitable for emission. Source declarations are first converted into potential declarations (or specifiers) at which point the eventual scope into which the declaration contributes may be undetermined. Once the scope is determined, the potential declaration is installed as an actual declaration of the determined scope.

Meta-programs run within the compiler, and so it is more accurate to say that the meta-program is interpreted rather than executed. Of course a sophisticated meta-compiler could compile the meta-program and then (meta-)execute it. Meta-programming may operate on potential or actual declarations.

1.5.7 Meta-Programming

FOG supports static meta-programming, that is the execution of user supplied code at meta-compile-time. This code may analyze and modify declarations in order to ensure compliance with some constraint.

This may just involve monitoring declarations to ensure that some policy is observed. For instance a meta-program could verify the presence of a virtual destructor under all or selected conditions.

More powerfully, a meta-program may be used to generate code automatically to support run-time access to compile-time information. The generated code can be optimised to suit user requirements:

- a very modest form comparable to RTTI for simple debugging
- a more extensive set of tables or functions to support persistence or marshalling
- a very substantial set of tables to support full run-time meta-programming supported by a corresponding run-time executive

The facilities provided by FOG are very much focused on meta-programming as a manipulation of program declarations. This contrasts with but complements the manipulation of types, functions and expressions using template meta-programming.

1.6 FOG versions

The FOG grammar is a superset of the C++ grammar, with extensions to support compile-time programming. FOG is a translator from extended C++ to standard C++. The extensions are sufficient to render the C preprocessor redundant, although Cpp continues to be supported for compatibility.
It was originally thought that the goals of FOG could be achieved by relatively simple processing at a lexical level. This proved to be naive. Useful manipulation of C++ declarations requires accurate parsing of those declarations and therefore a fully fledged C++ parser is needed. C++ is context dependent, but flexible meta-programming requires source declarations to be interpreted before their context is known. A new approach is therefore introduced so that FOG can perform context-free syntactical analysis of C++. The evolution to the current implementation is described in Chapter 5.

Superset Implementation (version 2)

The description of the FOG meta-compiler in this thesis applies mainly to the current implementation that uses a superset grammar to support context-free parsing of FOG (and C++) source. The lexical and syntactical analyses of this meta-compiler are substantially complete. The semantic analysis and subsequent stages are fairly complete, and more than sufficient to demonstrate the soundness of the approach using simple examples. Missing functionality is mostly at the edge of the language and so concerns exceptions, partial template specialisations, namespace-aliases and using-directives.

In some areas, the examination and resolution of design decisions goes some way beyond the implementation.

Multi-pass Implementation (version 1)

The previous implementation resolved syntactical ambiguities using multiple passes. Ambiguities in meta-constructs were avoided by using a syntax-driven and consequently context-dependent method. This implementation suffered from deficiencies inherent in the syntax-driven method.

FOG and associated publications are available from the FOG home page:

http://www.computing.surrey.ac.uk/research/dsrg/foa
2 Related Work

2.1 Language Constructs

2.1.1 Syntax

Stroustrup has highlighted the inadequacies of the C preprocessor in [Stroustrup97], where he calls for its eventual demise. Very little work has been published on practical alternatives. Straightforward lexical alternatives such as m4 suffer from many of the same problems by operating independently of the underlying language. Operation in collaboration with C++ involves tackling the challenge of C++ syntax, which is difficult to parse and difficult to extend. It is hard to add new syntax to C++ and so the limited extensions available through meta-functions and meta-variables have an inevitably inferior appearance to a solution that introduces new keywords. Werther [Werther96] provides a sensible proposal for a completely new C++ syntax using more conventional syntactical styles like Ada or Pascal. Within a clean syntactical framework, it would be much easier for researchers to examine alternative syntaxes, and it would be possible for a meta-program to perform syntax extension.

2.1.2 Macros

Macros have a very long history, much of it rather old since macros were important to augment early "high" level languages. Macros remain essential for assembler programming. [Solntseff74] provides a survey of 22 extensible languages, classifying them as Type A to G according to whether language extension is performed during

- lexical analysis
- syntactic analysis
- parse tree production (semantic analysis)
- intermediate analysis (tree optimisation)
- code generation
- code conversion

Consideration of code conversion was then purely hypothetical but foreshadowed Java load-time activities. It is difficult to classify FOG, where substitution occurs as trees are manipulated during syntactic rather than semantic analysis. Re-use of parse trees corresponds to a Type C extension, but operation during syntactic analysis is Type B.

A different classification of just macros rather than language extension mechanisms is made by [Cheatham66]. Extension may occur

- preceding lexical analysis - text macros
- during syntactic analysis - syntactic macros
- following syntactic analysis - computational macros

Text macros correspond to C preprocessor macros, and computational macros correspond to C++ inline functions and templates. It is syntactic macros that are missing from C++. Since there is no distinction between syntactic and semantic analysis, FOG substitution and prototypes fall tidily into the syntactic macro classification.
An easier to understand classification is based upon the structure of the replacement. [Weise93] identifies

- character-level substitution - text macros
- token-level substitution
- syntax-level substitution - syntax macros
- semantic-level substitution

Token-level substitution as practised by the ANSI C preprocessor occurs between lexical analysis and syntactic analysis. (Or possibly as a late phase of lexical analysis as in Figure 5.4.) This does not quite fit the earlier classifications.

Semantic-level substitution is used by [Maddox89] to classify semantically-sensitive macros; problems distinguishing between the use of a name from a definition or invocation context are resolved by passing environments as parameters. This difficulty is also addressed by [Hieb92] where problems of losing source context are addressed by hygienic macros.

At the usage level, the situation is rather different. in FOG. 'Macro' operation is occurring at (meta-)compile-time rather than run-time in a language that supports hierarchical naming through classes and namespaces. The problems of resolution in the correct environment are addressed by appropriate use of \$ or \$'s as the substitution operator, and if necessary by passing an appropriate metaobject as a parameter.

At the implementation level, the problems and solutions are very similar in FOG; an abstract FogScopeContext class is the root of a hierarchy of some 60 odd environment defining classes that are passed between compilation routines (These classes make extensive use of the Decorator pattern to support addition of behaviour to pre-existing behaviour. The instances form a daisy chain of stack based objects, making for efficient creation and easy traversal to resolve problems such as multiple-dollar look-ups.) Source line tracing presents no significant problems for FOG.

The syntactic macro was introduced by [Cheatham66] and [Leavenworth66j. The macro and its arguments have syntactic types corresponding to parts of the language grammar. [Vidart74] gave these concepts a sound and efficient foundation using an Abstract Syntax Tree (AST) to represent them and avoid repeated syntactical analysis of source text.

[Weise93] applies these ideas to ANSI C and exploits a Lisp-like backquote to support a pattern template for substitution, avoiding the need for extensive call trees to rearrange the AST. Weise's approach is very much an extension to ANSI C introducing new keywords and 9 lexical operators. The approach in FOG is in some ways very similar to Weise's, however by giving existing concepts a compile-time meaning, and retaining a degree of consistency for all the corollaries, FOG achieves a notation that is more compact, supports character- or syntax-based substitution with only two new lexical operators (\$ and @) and no new reserved words. Where Weise needs backquotes and an explicit return statement to activate source-like declarations, FOG just treats all declarations as the return. In FOG, all concepts are put into a C++ Object Oriented perspective.

### 2.1.3 Joins

Relatively well-behaved coarse-grained merging of application functionality is provided by functions. More flexible but ill-disciplined merging is provided by macros. The BETA fragment system [Knudsen99] provides a finer-grained composition that observes predictable semantics for code fragments that satisfy the attributes of pre-declared slots.
2.1.4 Meta-classes

The concepts of meta-classes were first defined for Smalltalk. Languages such as CLOS have been extended with a MOP (MetaObject Protocol) [Kiczales91]. Even Java [Gosling97] has a class type for every class. C++ has rather lagged behind, perhaps through a mismatch of the run-time characteristics of traditional MOPs and the statically compiled philosophy of C++, perhaps through the compiler writer's desire to prevent further explosion of language complexity. FOG provides statically compiled meta-functionality, which can be used to define customised run-time meta-functionality.

Meta-classes were first introduced to support the configuration of objects at run-time in Smalltalk, and have subsequently become an important part of most Object Oriented languages. Limitations of the Smalltalk implementation led to the development of the simpler Deltatalk [Borning87]. The problems are resolved in the pure object model of ObjVlisp [Cointe87]. However, [Maes87] argues that the pure object model fails to distinguish the meta-level adequately. The restricted object model in FOG does precisely this; only metaobjects exist at compile-time, and only real objects at run-time.

A more versatile object model allowing the inheritance of meta-classes to differ from their classes is supported by CLOS and SOM. [Graube89] identified the resulting compatibility problems that arise during traversal of parts of the cycle: class, derived-class, derived-meta-class, meta-class, class. The history of these problems and their solutions in SOM 1, 2, and 2.1 are described in [Danforth94]. [Bouraqadi-Saidani98] synthesises additional meta-classes with multiple inheritance to resolve the problems. These problems do not arise in FOG since the two inheritance hierarchies are the same. They also do not arise since there is no object creation at meta-compile time and so no level traversal.

[Briot89] explains the motivation for distinct meta-class inheritance as an ad hoc solution to the propagation problem whereby a concrete class inherits the inappropriate property of abstractness from its abstract base class. This problem does not arise in FOG, since C++ offers at least two distinct solutions to the original problem using pure virtual functions or protected constructors.

The lack of meta-classes has always been a deficiency of C++, for which various proposals were suggested during standardisation. Eventually the standardisation committee compromised on the relatively limited functionality known as Run-Time Type Information (RTTI). A more substantive proposal [Buschmann92], is largely proprietary and so it is difficult to assess accurately. It defines a run-time Meta-Information Protocol providing more extensive data structures with global functions to support iteration. FOG provides a compile-time meta-level, in which application meta-programs may be used to create whatever run-time data structures are appropriate. These may vary from just class names to large descriptive tables for use by a run-time environment that supports run-time meta-programming.

2.2 Meta-level and Reflection

A procedure that manipulates declarations at compile-time might seem to be a simple generalisation of a macro, however harnessing the increased power, provided by this form of self-modifying code, offers ample scope for some fairly difficult papers.

2.2.1 The tower

[Smith84] coined the term reflective for a program that is self-aware. He introduces a minor Lisp variant 2-Lisp that subsumes Scheme, but provides a semantically rationalised notion of evaluation. The 2-Lisp dialect forms a sound
foundation upon which the reflective 3-Lisp dialect is realised. Further application dialects can then be defined recursively to form the 'reflective tower'.

The use of a different language dialect at each level of the tower presents challenges to analysis. [Wand88] describes the behaviour without using reflection. Multiple dialects are well-understood in theory, but a major inconvenience in practice. The difficulties are overlooked by many implementations, leading to the problem of meta-circularity when more than one meta-program operates upon the same program. Which meta-program operates first? Does the second meta-program operate on the source or the results of the first meta-program? How does a meta-program behave when it changes its own declarations? [Chiba96] notes that each level of the reflective tower must exist to define an ordering, so that one meta-program operates consistently on the results of a more nested meta-program and is isolated from its own and less nested meta-programs.

2.2.2 Metaobject Protocols

A reflective program manipulates the metaobjects that describe the program. The programming interface to these objects is defined by a MetaObject Protocol such as that provided for CLOS by [Kiczales91]. Each metaobject represents the reification of some programming concept, and so different reflective languages support very different MOPs. This may be as minimal as the RTTI facilities of C++, or as substantial as the meta-computation protocol of [Sobel96].

2.2.3 Languages

OpenC++ version 1 [Chiba93] provided a run-time MOP for C++, through the recognition of comment mark-ups requesting indirection of method calls through metaobjects. This approach was abandoned in favour of a largely compile-time MOP in OpenC++ version 2 [Chiba95], using a two-stage compilation process, first to build an enhanced compiler for the extended language and then to use that compiler. The two stage process avoids the costs of interpreted meta-execution, but unfortunately prevents the realisation of the reflective tower. Although OpenC++ extends most C++ concepts to the meta-level, somehow the language doesn't feel like C++; the extensions are rather haphazard, and much of the meta-programming involves considerable insight into the operation of a compiler. The working representation is a rather strange hybrid between a Lisp-like list and a C++ syntax tree in which punctuation remains significant. This is perhaps attributable to the development path through S++, a form of Scheme supporting C++ concepts.

OpenC++ claims to be based on the principles of the CLOS MOP [Kiczales91], but C++ concepts are so different that it is difficult to see any resemblance. The CLOS MOP was developed for run-time support, and is a natural formalisation of an API that is present anyway. Lists are well-supported by CLOS and so while the list manipulations involved in meta-programming may be difficult for a C++ programmer to understand, they are consistent and compact. When these concepts are transferred to C++, the alien nature of list processing, the consequent lack of language support and the very different C++ perspectives make for an uncomfortable programming environment.

MPC++ [Ishikawa96a] provides a compile-time meta-level that like OpenC++ supports fairly extensive interception of compiler activities and subsequent peek and poke meta-programming, using conventional C++ syntax shifted to the meta-level by $meta. MPC++ avoids the problems of the tower by supporting a stack of output streams, but since these are text rather than syntax-based, they lack the integrity of composition of declarations, statement nesting or character concatenation in FOG.
OpenC++ and MPC++ share the same principle of operation. A meta-program intercepts stages in the compilation process, recognises patterns\(^1\) in the code, and modifies the syntax tree by construction of alternate partial trees. The FOG approach is completely different. The pattern is explicitly identified by the invocation of some declaration (often a meta-function) that implements the pattern. Additional declarations may be composed to elaborate the implementation for more sophisticated requirements.

The peek and poke approach is able to achieve most programming goals, but requires the programming skills to peek and poke syntax trees and poses the challenge of recognising all variants of the target pattern. The declarative approach in FOG is less flexible, but only requires relatively conventional skills and may need the interception points to be explicitly identified, albeit by inline function calls. In FOG two declarations are woven by providing the two declarations. The peek and poke approach requires a program to be invoked, the join point to be identified in one tree, and the tree for the other declaration to be hand assembled and merged.

The ease with which OpenC++, MPC++, Sina/St (a precursor to C++/CF) and D (a precursor to AspectJ) could be used is discussed in [Lin99]. All are found lacking for a three-level architecture and so Adapter++ is presented. The problem would appear to be directly solvable with FOG.

Iguana [Gowing96] provides a run-time MOP in which many different activities can be reified on a per-object, per-method or per-class basis. The implementation cost is therefore determined by the required degree of reification. This provides considerable flexibility at run-time but offers little at compile-time, since Iguana is not a reflective compiler. [McAffer95] takes reification even further with seven components to a message invocation.

Meta-classes form part of the Java language definition, and so there is more language support for reflection and the interesting opportunity for a user-defined class loader to perform meta-programming at load-time. Dalang [Welch98] and Kava [Welch99] provide an ability to intercept method calls. Guaraná [Oliva98] introduces composer metaobjects to enforce composition policies.

OpenJava [Chiba98a] migrates the peek and poke concepts of OpenC++ to Java, providing a way of implementing parameterised classes [Chiba98b] that is rather at odds with more serious language proposals such as GJ [Bracha98], NextGen [Cartwright98] or [Agesen97]. Java Beans are used as the basis for a two-layer meta-level model by [Wu98], and by [Lorenz98] to provide a reflective implementation of the Visitor pattern.

### Applications

Reflection has been used to solve problems in a wide variety of applications. [Cartwright98] reports on the successful use of OpenC++ to simplify and enforce the interface to an AI library, but only after flattening their meta-program to avoid OpenC++ restrictions.

[Kasbekar98] again using OpenC++, identifies run-time data-dependencies so that a roll-back to a checkpoint can be done efficiently.

[Yokote92] describes the reflective Operating System Apertos.

Further applications involving aspects, communication, constraints, distribution, patterns, persistence and synchronisation are discussed elsewhere.

---

1. Not a design pattern, although the FOG invocation is design pattern-like.
2.3 Programming Styles

2.3.1 Patterns

The original GoF patterns book [Gamma95] has provoked considerable interest and a growing number of specialised conferences and workshops. Example implementations of the patterns were mostly in C++ but just occasionally in Smalltalk, so translations exist for Java [Grand98] and Smalltalk [Alpert98]. The POSA book [Buschmann98] provides further architectural and design patterns.

Unfortunately, from a programmer's perspective, the ability to present any problem as a pattern has broadened the field so that the additional coding patterns contributions provided by the PLOPD conferences [Coplien95b], [Vlissides96], [Martin97] and [Harrison99] are diluted by management and organisational patterns. However, remaining within the programming domain, [Fowler97] covers the rather different perspective required during analysis.

Patterns are very abstract and so pose considerable classification challenges. A pattern is often named rather arbitrarily by its author, and so looking for a pattern that concerns a particular kind of problem is hampered by the lack of clear vocabulary for problem or solution. Two attempts at classifying the basic patterns have been made. [Gil97] distinguishes between clichés (straightforward use of prevalent mechanisms), idioms (a built-in facility in some languages) and cadets (not built-in to any language). [Agerbo98] takes a harsher view of just the GoF patterns, discarding 2 as not patterns at all, 7 as Language-Dependent Design Patterns, and 2 as closely Related Design Patterns: only 12 survive to be classified as Fundamental Design Patterns.

A much more fundamental perspective on patterns is taken by [Pree94], where all collaborations are reduced to seven meta-patterns corresponding to different forms of 1:1 and 1:N relationship. Pree claims a high-level perspective, which is surely wrong: a complete set of object to object relationships is a suitable low-level abstraction that can form one of the layers for a parameterised pattern. The GenVoca approach [Batory97] with its cascade of small orthogonal behaviours would appear to combine well with Pree's meta-patterns, once a suitable tool, such as FOG, is available to glue the behaviours together.

Most attempts to represent patterns in code are informal. Soukup addresses the problems of implementing patterns, with [Soukup95] summarising the much more extensive treatment in [Soukup94]. Soukup’s solution, supported by the CodeFarms library, realises each pattern as a data-less class that is declared as a friend of each collaborator. Pattern operations are realised by static member functions of the pattern class, so the programming interface is unnatural: operations are invoked upon the pattern, not the primary collaborator. However, the behaviour is very regular and has extremely good characteristics with respect to include file dependencies. The implementation is hampered by the same limitations that motivated the removal of the One Definition Rule from C++: a special preprocessor has to be used, which could beneficially be replaced by FOG.

An alternative text-based mechanism is provided by SNIP [Wild96], although the enhancements effectively introduce two new languages to implement a kind of marked up source text with a rather accidental semantics for lexical composition.

A reflective implementation of two patterns using OpenJava is described in [Tatsubori98]. Instantiation of a pattern is quite tidy. Definition of a pattern involves a significant amount of peek and poke code and the introduction of reserved words.

The difficulty of actually fielding a reusable implementation of a particular solution is rather neglected. Vlissides participated in the development of a GUI tool to generate code for patterns automatically [Budinsky96], but then expresses
considerable reservations in his book [Vlissides98]. It is indeed difficult to conceive an automatic generator that will have sufficient flexibility to balance all the conflicting forces and select the appropriate cookbook implementation. There is rightly much generality and ambiguity in pattern descriptions. However programmers regularly re-use particular pattern implementations with which they are familiar, and providing an improvement over cut and paste for such re-use would be beneficial.

Some experimental work on a GUI-based interface for pattern instantiation in Smalltalk has been reported in [Florijn97] and [Meijers96]. It is difficult to assess quite what has been achieved. Their system seems to exploit the ability of a Smalltalk program to reconfigure dynamically at run-time allowing pattern objects to be cloned interactively to extend the system under development.

Early descriptions of patterns identified the participants as the collaborators, which tended to be classes in an implementation. More recent work [Riehle97] and [Riehle98] has concentrated on the different roles that the participants play, and thereby begun to establish a hierarchy in which some more sophisticated patterns are composites of simpler patterns.

Application of a pattern requires roles to be associated with classes. The early descriptions tended to assume that a particular set of roles was performed by each collaborator, blurring the distinction between roles and classes. The role-based perspective provides a more generic insight and offers more opportunity for providing flexible tool support.

Composition of patterns is also addressed by [Lauder98]. A very generalised abstract pattern is identified for decoupled collaborations that can be trimmed to satisfy the more specific behaviour of the Adapter, Facade, Mediator, Observer or Reactor [Schmidt95] patterns.

Although patterns and roles (and aspects) appeared rather late in the evolution of UML, the generalisation of a collaboration diagram described in the User Guide [Booch99] permits parameterised instantiation. This appears to provide the required notational support. Unfortunately the change is minor and recent, and so attracts little attention in the Reference Manual [Rumbaugh99] or the Unified Process [Jacobson99] and temporarily lacks graphical tool support.

An approach to the enforcement of compliance with pattern constraints is provided by [Hedin97a] using attribute extension [Hedin97b] to extend a language grammar to incorporate patterns directly.

Reverse engineering patterns from code is provided by DP++ [Bansiya98]. Heuristics are required to recognise the relatively ill-defined implementations that need to be found, and it is not possible to distinguish structurally similar but operationally different patterns.

Direct generation of code from CAD tools has been an unrealised goal for many years. The flexibility for the invocation of a single meta-function to generate declarations in many classes, and for meta-programming to enforce or create non-trivial program structure may provide the necessary support for a CAD tool that just emits parameterised calls to a suite of meta-functions implemented by a meta-library developer.

2.3.2 Aspects

Traditional programming approaches decompose a problem into functions or objects that often have a direct realisation in an implementation. An alternative decomposition can be made in terms of the concerns, properties or aspects of the system that the programmer must consider in order to satisfy system requirements. Effective decomposition identifies loosely coupled modules that can be implemented and tested independently, but decomposition from one perspective generally destroys modularity from other perspectives. For instance,
Related Work

A concern for error logging may pervade many functions and objects. Since a practical implementation is likely to be a function or object, the problem arises as to how to modularise the error logging aspect. The design of an aspect should be isolated from the application functionality but the implementation must be interwoven with the application code.

An informal estimate indicates that about 80% of an application is well suited to a structural decomposition, whereas the remaining 20% cuts across this structure and is more appropriately decomposed into aspects or programming concerns.

[Aksit96] discusses the need to maintain separation between the different concerns, and advocates the use of Composition Filters [Glandrup95]. Therefore, in C++/CF the different stages of message passing are reified, so that meta-code can intercept messages according to a variety of source and destination criteria. The practical implementation of C++/CF involves a custom preprocessor of limited capability that imposes considerable restrictions and obligations on the programmer. The greater capabilities of FOG could remove much of the inelegance.

The concepts of aspects have been taken up more generally in the new field of Aspect Oriented Programming. The review of the first ECOOP workshop [Mens97] considered whether AOP was really new or just a more palatable name for meta-programming, given that AOP problems could be solved by reflection. [Kiczales97] provides an extensive discussion of the need for AOP and some interesting examples involving loop fusion, arguing that a meta-programming approach is just a stepping stone. Eventually each aspect could be supported by a customised programming language or environment [Fradet99], [Seinturier99].

Combining the functionality of the aspect code with the non-aspect code requires a mechanism to define how the code should be combined and requires a weaver to perform the composition. [Ossher98] discusses the join points where composition occurs and highlights the dangers imposed by the invasive characteristics of extra Aspect-Oriented code in comparison to the additive characteristic of Object-Oriented code. In a rather different field, [Mulet95] describes constraints upon the composition of functions so that composition occurs predictably via nested invocation rather than ad hoc cut and paste. FOG can provide rather ad hoc code merging, but the use of nested meta-variables as in Section 7.4.1 provides an opportunity for greater discipline.

AspectJ [Lopes98] provides the flexibility for Java programmers to make controlled additions to classes via introduce and advise weaves. An aspect is introduced into Java as an instantiable entity, providing the flexibility to dynamically associate aspects and objects at run-time; the object behaviour can mutate. This provides direct language support for a particular and certainly useful pattern, and by introducing the support at load-time rather than compile-time is able to do so without specifying detailed implementation semantics. The aspect syntax forces tight coupling to the application, and so [Beaugnard99] suggests a relaxation to allow aspect, join and application to be independent.

FOG provides more extensive and more varied facilities than AspectJ, but needs to resort to meta-programs to implement the limited but useful capabilities directly available in AspectJ.

The flexibility of load-time weaving is also exploited by [Welch99]. Run-time weaving with support for aspects at the meta-level is advocated in [Böller99], [Lunau98] and [Prýor99]. A more pragmatic compromise in which some aspects are statically woven at compile-time while others are retained at run-time is suggested by [Matthij97].

2. Gregor Kiczales at an AspectJ tutorial
3. The current implementation weaves at compile-time.
A slightly different take on programming concerns arises in Subject Oriented Programming [Harrison93], where the differing perspectives of the same objects appropriate to different applications is considered. The concepts of different views of an object provided by CV++ [Shilling89] are taken much further so that each application may be written with its own subjective view of each object. When such objects are shared between subsystems or applications, an update to one subject must make consistent changes to the underlying object and alternate subjective views. Composition policies and language extensions to achieve this are described by [Ossher95]. FOG could be used to implement the associated disciplines.

2.3.3 Generative Programming

Generative Programming [Czarnecki97] seeks to provide highly configurable components that can be combined and optimised at compile-time so that minimal overheads are incurred by an application.

An early perspective into reusable components is provided by [Batory92]. Two independently developed domain-specific module generators, Genesis and Avoca, were found to have very similar design and implementation. The common concepts were combined to give the GenVoca principles of composing very thin fairly orthogonal layers to create a desired component. In [Batory93] these concepts are applied to C++ libraries, resulting in fewer source concepts to generate more, smaller and faster library components than standard C++ libraries. Further improvements in speed and flexibility are reported in [Batory94], using a succession of customised preprocessors and compilers: P1, P2 leading to P++ that adds support for realm and component to C++ [Singhal96]. Composition of components is based on a realm of interchangeable components with a common interface. The realm may therefore be used as a type signature. However, in practice, not all components are completely interchangeable, there may be constraints on, or prerequisites for, the ordering of compositions. [Batory97] identifies the need for upward and downward checking of constraints, using pre/post-conditions/restrictions. A related implementation of container libraries using template meta-programming is described in [Czarnecki99].

A direct form of generative programming is possible with imaginative use of C++ templates. [Myers95] describes the concepts of traits classes (template classes of parameters), which are used to pass a set of parameters and interrelationships as a single template parameter, thereby considerably simplifying the instantiation interface. These concepts underlie much of the flexibility of the Standard Template Library [C++98]. Inference of expression types at compile-time is exploited in [Veldhuizen94] to generate customised inline functions that outperform conventional library implementations. Control structures are realised by recursive template instantiation in [Veldhuizen95] supporting generation of more sophisticated customised inline functions for sin, FFT or bubble sort.

The practise of composition of behaviour using mix-ins is reported to incur a potentially exponential growth in the length of template class names in [Smaragdakis98], where an extra outer mix-in layer is introduced to resolve the problem.

The GenVoca approach seems well suited to the implementation of efficient reusable components from very simple building blocks. Template meta-programs provide effective techniques for composing the building blocks at compile-time, provided the composition results in a function or type. Unfortunately interesting components are more complicated. [Eisenecker97] suggests that GP subsumes AOP, but the template approach requires aspect functionality to use pre-existing parameterisation, rather than an independently developed weave. The facilities of FOG are required to compose more general declarations, in particular for extensions not supported by the base functionality.
2.4 Applications

C++ has been extended in minor ways by practical compilers [Stallman98] and [Microsoft97], and a few isolated language extensions such as [Baumgartner97] have been published. Researchers in many fields have chosen to use C++, but found it inadequate for their purposes. There are therefore many domain specific extensions to C++, just some of which are mentioned here.

Domain specific extensions, when fully integrated with C++, can provide a clean solution to the domain problem. However, many extensions are poorly integrated because of the size and complexity of C++ and so provide little more than a research tool. Many of the problems dealt with in a domain-specific fashion can be resolved in a domain-independent way by using the meta-level programming facilities of FOG. However FOG meta-programming is restricted to declarations and so the more radical changes of C++ [Larus96] in which data parallel semantics are introduced to expressions could probably not be addressed.

2.4.1 Design by contract

Design by contract advocates the use of pre- and post-conditions and invariants to define the behaviour of components, and it is beneficial for these properties to be expressed in implementation code.

Support for contracts is an integral part of Eiffel [Meyer92]. It has to be added for C++.

A++ [Cline90] extends C++ in a fairly natural way to support class assertions and invariants, which can in principle be optimised at compile-time.

CCEL [Duby92] adds a form of meta-programming using assertions in a predicate calculus so that constraints can be validated.

[Porat95] proposes some language extensions to support pre-conditions, post-conditions and invariants.

2.4.2 Persistence and Marshalling

A low level understanding of object layout is necessary for persistent storage of objects in databases or for marshalling objects whether for signalling between nodes in a communication network or distribution between nodes in a parallel processor.

Persistence is commonly supported by an extended language adding a persistent keyword to C++ as in OQL or E [Vemulapati95], or a replacement allocator such as pnew in O++. An alternative approach is taken by [Park96] using an object pre-header at negative address offset, so that persistent objects are interchangeable with non-persistent objects. A MOP approach is recommended by [Stroud94] to avoid the inflexibility imposed by the extended language PC++. A simple reflective system based upon the Java API is described by [Lee98].

Wilson and Lu [Wilson96] provides extended articles by 16 of the leading research teams using C++ for parallel processing. Some researchers used only library classes and run-time support code, and so remain entirely within the normal confines of the C++ language. Others introduce language extensions, which are variously implemented as translators to C++, or modified C++ compilers. MPC++ [Ishikawa96b] exploits meta-level facilities to support an extended syntax within a "standard" C++ compiler. Many of the C++ extensions appear unnecessary and some authors recognise that more imaginative use of C++ facilities, particularly those not readily available at the start of their research could have reduced the need for divergence.
2.4.3 Synchronisation

Synchronisation is critical for reliable multi-process or multi-processor applications. Concurrent access to data has to be restricted, a problem resolved in principle by monitors [Hoare74]. In practice there are typographical difficulties in ensuring that monitor protocols are not accidentally bypassed and genuine difficulties in ensuring that a synchronisation policy is sensibly applied by derived classes. Alternative strategies are considered in [Matsuoka93], where reflection is considered necessary to solve the inheritance anomaly. Reflection is exploited by [Stroud95] to implement atomic access to data types.

2.5 Summary

The different approaches demonstrate that language extension can occur at three different levels:

- Library classes and run-time environments can be developed without any language or compiler changes. FOG's increased capabilities at compile-time provide library developers with more options, perhaps supporting simpler interfaces, reduced requirements for user support code, or stronger compile-time detection of protocol violations.
- Translators that recognise one or two extra reserved words require development of the translator but do not affect the underlying compiler. FOG syntax macros provide an ability to introduce custom extensions to C++, enabling many of the characteristics of custom translators to be achieved by a general-purpose translator.
- New forms of code generation necessitate significant revision to both language and compiler. FOG offers very little to applications that need to rewrite the basic compiler.

The need for many different research teams to develop customised variants of C++ demonstrates the need to be able to extend C++ to support new domains.

Research in the fields of patterns and Generic Programming shows an increasing need to structure large software components from smaller ones.

Aspect Oriented Programming demonstrates the need to combine relatively independent software modules into a composite whole.

FOG provides facilities to assist in all these areas. Unfortunately, it is difficult to answer the critique that C++ is too large, and that adding meta-functionality is an enhancement too far. However, it is also difficult to avoid recognising that the absence of meta-functionality is restrictive for some domains and an inhibition to re-use for all.

Programmers need to be able to express their ideas in compact modules, so that any form of repetitive and consequently predictable practice can be captured by a module that can be re-used. Functions support this concept for algorithmic behaviour. Records and objects support this for data structures. Templates extend the concepts across a variety of data types. FOG derivation rules provide further extension to program declarations. More arbitrary flexibility requires compile-time or meta-programming with meta-functions capturing the repetitive program structure.

FOG, OpenC++ and MPC++ each provide meta-programming, but FOG's capabilities are distinctly inferior because FOG does not currently provide direct access to the underlying ASTs. However, because FOG operates by composing declarations, FOG meta-programming integrates with C++, and so replaces the C preprocessor, and solves many practical programming problems decoratively. Equivalent solutions using OpenC++ or MPC++ must use imperative peeking and poking. FOG is upward compatible with C++ and so can make a much stronger claim to be an extended C++ than OpenC++ of MPC++.
Expanding FOG's support for analysis of expression ASTs is a matter for further research.
3 FOG Grammar

Sections 1.4 and 1.5 provided a brief overview of the FOG extensions to C++ including:

- substitution
- concatenation
- composition
- meta-scopes
- derivation rules
- meta-programming

In this chapter these language extensions are presented in detail and the semantics specific to each extension are discussed. More general semantic issues are discussed in Chapter 4.

Description of the necessary but rather peripheral extensions required to support multiple intermediate files between the FOG translator and the C++ compiler is deferred till Chapter 6.

It is a moot point whether this chapter should appear early, or late or be relegated to an appendix. On the one hand, it contains too much important material to be an appendix and lays the foundation for the subsequent chapters. On the other hand some of the detailed descriptions become very detailed. The material is therefore presented here, and the reader is invited to skip directly to the summary in Section 3.5 on page 80, if satisfied by the overview in Chapter 1, or to skip to a following sub-section if a description becomes too detailed.

The changes are presented one at a time in this chapter. The summary of the changed grammar in Appendix A shows all changes combined and is structured to ease comparison with Annex A of [C++98].

The final section justifies the claim that FOG renders the C preprocessor redundant.

3.1 Grammar Extensions

Most of the FOG extensions contribute extra grammar, however the substitution and concatenation functionality is white-space sensitive and so must be performed earlier. This occurs during phase 6 of the C++ translation process summarised in Figure 3.1.

3.1.1 Substitution, Concatenation and Tokenization

The C++ standard defines (§2.1-6) translation phase 6 as:

Adjacent ordinary string literals are concatenated. Adjacent wide string literals are concatenated.

and (§2.1-7) phase 7 as:

White-space characters separating tokens are no longer significant. Each preprocessing token is converted to a token. The resulting tokens are syntactically and semantically analyzed and translated.

FOG generalises phase 6 processing to support more extensive concatenation and also the recognition of substitution operators. The parsed result of a substitution or non-trivial concatenation is referred to as a tree-literal, since it comprises a pre-parsed AST.

FOG generalises phase 7 to support tokenization of a tree-literal as an identifier, and to defer treatment of non-reserved words as identifiers.
The enhanced behaviour of FOG is most easily explained by first elaborating the simple C++ descriptions into grammars before showing the revised grammars.

3.1.1 C++ Phase 6 Concatenation Grammar

The C++ Phase 6 translations can be expressed as

```
string-literal_{cat}
  string-literal_{pp}
  string-literal_{cat}, whitespace_{pp}, string-literal_{pp}

'anything-else'_{cat}
  'anything-else'_{pp}
```

_{pp} denotes the preprocessor token input to phase 6 from phase 5 and _{cat} the concatenated output production passed from phase 6 to phase 7.

Single quotes as in "anything-else" surround a production whose meaning is obvious though difficult to express compactly.

3.1.1.2 C++ Phase 7 Tokenization Grammar

Phase 7 tokenization can be expressed by the following 'grammar'

```
'discard': whitespace // Token is discarded
'punctuation': 'punctuation'_{cat} // e.g., or += or ...
'reserved-word': identifier_{cat} // e.g. true or unsigned or if // If identifier_{cat} is a reserved word
character-literal:
  character-literal_{cat}
floating-literal:
  number-literal_{cat} // If number-literal_{cat} is floating point
```

Figure 3.1 C++ translation
3.1.1.3 FOG Phase 6 Concatenation Grammar

FOG replaces phase 6 by a white-space sensitive grammar to augment the C++

- concatenation of adjacent string-literals

by

- concatenation of unseparated character-literals
- concatenation of unseparated identifiers
- concatenation of unseparated number-literals
- concatenation of unseparated tree-literals

As in C++, the distinction between ordinary and wide-string literals can be ignored, since their adjacency leads to undefined behaviour (§2.13.4-3). Behaviour is only defined for a sequence of same-width string-literals or character-literals.

```
text-literal:
  character-literal
  identifier
  number-literal
  string-literal
  tree-literal

character-literal:
  character-literal
  identifier
  text-literal

identifier:
  identifier
  text-literal

number-literal:
  number-literal
  text-literal

string-literal:
  string-literal
  text-literal

identifier_cat:
  identifier
  text-literal

number-literal_cat:
  number-literal
  text-literal

string-literal_cat:
  string-literal
  text-literal

identifier_cat_whitespace:
  identifier
  whitespace
  text-literal

identifier_cat_whitespace:
  identifier
  whitespace
  text-literal

anything Else:
  "anything-else"
  text-literal
```

Translation of source tokens involves three significant textual representations:

1. The original source code spelling
2. Any escape sequences and digraphs in this representation are replaced during translation phase 5 to give
3. An internal textual (multi-)byte sequence

---

1. There is a momentary fourth true source code spelling before trigraphs are replaced in translation phase 1.
that comprises straightforward binary encoding of each character. This representation is in turn converted during the emission phase to

- a representation suitable for output

which may require regeneration of escape sequences. This regeneration propagates the original source spelling when the output corresponds directly (no concatenation) to a source token.

Concatenation operates by concatenation of internal sequences without regard to the character-literal, identifier, number-literal or string-literal categorisation.

The textual byte sequence of a numeric value is the source spelling, if the value originated from source. Otherwise the textual sequence is generated from the numeric value using a numeric to ASCII conversion. The default formatting is specified only to require precision not less than long for an integer-literal or float for a floating-literal. Specific built-in meta-functions may be used for more precisely controlled formatting.

The textual byte sequence of a tree-literal comprises the byte sequence of the main unqualified name in the tree-literal, such as the class-name of a class-specifier:

```cpp
auto class_specifier c = class Scope::Class {};  
auto identifier i = $c;  // i = Class
```

Preservation of unchanged format and conversion between identifier and string-literal perspectives provides most of the functionality of Cpp # operator stringizing. Generalisation to character-literals and number-literals provides consistency rather than significant new functionality.

It is surprising that this fundamental lexical extension causes only very minor incompatibilities, mandating protective spaces where only a perverse coding style would omit them:

- around the string-literal in a linkage-specification: extern "C" declaration
- around alternative tokens: and, and_eq, bitand, bitor, ..., xor, xor_eq
- following expression prefixes: return, sizeof, throw

This is described in Section 3.3.4.

There is no incompatibility regarding the L prefix for wide-characters or wide-strings since the L" and " are stripped during phase 3. Phase 6 processes the contents using a parallel rather than prefix annotation to signify a wide-string.

### FOG Phase 7 Tokenization Grammar

The concatenated preprocessor tokens are tokenized by the following 'grammar':

```cpp

discard/ / Token is discarded
whitespace
reserved-word:
  identifier cat
punctuation:
  punctuation pp
character-literal:
  character-literal cat
floating-literal:
  number-literal cat
  // If number-literal cat is floating point
integer-literal:
  number-literal cat
  // If number-literal cat is fixed point
string-literal:
  string-literal cat
meta-type-name:
  identifier cat
  // If identifier cat is a meta-type name
  // (and not a reserved word)
```

3.1.1.4 FOG Phase 7 Tokenization Grammar
Identifiers are categorised as one of reserved-word, meta-type-name (name of a built-in meta-type), non-reserved-word (word used in a FOG syntax extension) or other-identifier. The latter three and tree-literal are combined in the main grammar for uniform treatment where an identifier is expected.

The results of concatenation in phase 6 may not yield consistent tokens. For instance:

```cpp
auto identifier plus = '++';
x ${plus}${plus};
```

could be perceived as

```cpp
x ++;
```

If the tokenization grammar recognises that the identifier ++ can be tokenized as the corresponding punctuation token, then the example comprises an expression-statement that increments x.

The multi-pass grammar FOG implementation took this approach, reclassifying the byte sequences masquerading as identifiers into reserved-words, punctuation, number-literals or identifiers. However tree-literals are not necessarily resolvable during phase 6 without invoking a very tight coupling with syntactic and semantic analysis.

Retokenization of identifiers is no longer performed and so the only change in FOG phase 7 processing is the propagation without interpretation of tree-literals to the syntactic analysis, where they are treated, again without interpretation, as identifiers. (The justification for treating a tree-literal as an identifier is given in Section 4.2.3.)

The processing is therefore context-free and the example is interpreted as

- `identifier` with value x
- `identifier` with value ++
- the punctuation

This is a syntactically valid definition of a global variable named ++ of type x. It must be rejected during semantic analysis.

Retokenization can be achieved by using the `std::parse` built-in meta-function, using which, the increment behaviour may be obtained by:

```cpp
$std::parse("x "${plus}${plus}++;");
```

Whether trivial concatenations are resolved during phase 6 is an implementation option. However, certain complicated concatenations involving substitutions can only be resolved during semantic analysis.

The lack of direct support for retokenization does not introduce any incompatibility with C++, merely a limitation to the language extension. Very few programs are likely to require retokenization, and those that do will presumably only do so in a very restricted context, where the subtle behaviour may benefit from the need to expose it more clearly.
3.1.1.5 character-literal

In C++, a character-literal should comprise a single character, whereas a string-literal may contain any number of characters.

While C++ defines the concept of a multi-character-literal, the definition is of limited utility since it is implementation defined.

In order to support concatenation consistently, FOG generalises character-literals to encompass multi-character-literals and the zero-character-literal to treat them equivalently to string-literals during lexical and syntactic analysis. A character-literal may have any number of characters, however when a character-literal with other than one character is used in a C++ context (as a literal in a primary-expression), the behaviour is undefined.

This change supports concatenation consistently, but offers only minor functionality enhancements: an empty character can be used as a concatenation join or as a concatenation cast.

    auto for (int i = 0; i < 10; ++i)
    {
        static char digit_values[] = char($i); // 0 to 9
        static char digit_codes[] = ''$i; // '0' to '9'
    }

3.1.1.6 FOG tree-literals

Tree-literals support the use of a parsed AST from a meta-variable or meta-function.

Definition occurs using the meta-function or meta-variable declarations described in Section 3.1.5.5 and Section 3.1.5.6.

    auto enum specifier enumTree = enum f { F };

Access occurs through use of a $ or @ expression to provide a tree-literal_pp token.

    typedef $enumTree E;

This syntactic usage exploits the treatment of a pre-parsed AST tree-literal as an identifier by the main FOG grammar as described in Section 4.2.3.

tree-literal_pp:
    at-literal
    dollar-literal
    syntax-macro                     // Section 4.7
at-literal:
    @ tree-expression
    0 { tree-expression }               // Loosely referred to as an @-expression
dollar-literal:
    $ tree-expression
    0 { tree-expression }
    $ dollar-literal

The first two forms of tree-literal are triggered by a $ or @ introducer with an optional pair of braces to surround the actual expression. Syntax ambiguities are avoided in the absence of braces by defining the lexical expression as the longest possible token sequence. The semantics of syntax-level substitution is described in Section 4.2. The name-resolution rules are described in Section 4.3, and the significance of repeated $’s in Sections 4.3.5 and 4.3.6.

The third form of tree-literal supports a user-defined syntax-macro comprising a triggering identifier and relatively arbitrary subsequent syntax. Syntax macros are described in Section 4.7.

Analysis of the tree-literal invocation sequence is initiated by recognition of the trigger token in the white-space sensitive phase 5 concatenation grammar. This activates a nested whitespace insensitive syntactic analysis to identify the syntax
tree to be returned for use by the interrupted syntactic analysis. This syntax tree represents the invocation; it is not evaluated, since the invocation may not be directly or uniquely resolvable.

The nested analysis to identify the tree-literal invocation obtains tokens from the same source as, and in the same way as, each token of the interrupted analysis. Nested recognition of tree-literals is therefore supported with $a$b$c resolved as ${a${b${c}}).

The tree-expression defining the metaobject referred to by a $ or @ expression is defined by:

```
primarx-tree-expression:
  meta-scoped-id // Section 3.1.5.1
  ( tree-expression )
postfix-tree-expression:
  primary-tree-expression
  postfix-tree-expression ( tree-argument-listopt ) // meta-function call
  postfix-tree-expression [ expression ] // meta-array index
  postfix-tree-expression . scoped-id // meta-member selection
  postfix-tree-expression -> scoped-id // meta-member selection via iterator

primary-tree-expression:
  meta-scoped-id

postfix-tree-expression:
  primary-tree-expression
  postfix-tree-expression
  postfix-tree-expression
  postfix-tree-expression
  postfix-tree-expression

primary-tree-expression:
  meta-scoped-id
  ( tree-expression )

postfix-tree-expression:
  primary-tree-expression
  postfix-tree-expression
  postfix-tree-expression
  postfix-tree-expression
  postfix-tree-expression

```

meta-scoped-id is the optionally scoped name of a metaobject (in the meta-name-space).

A meta-function-call invokes a user-defined or built-in meta-function, globally or on a selected metaobject.

Array indexing selects a list element.

Member-selection selects a named member of a metaobject. Indirect member selection and indirection apply only to metaobjects of iterator meta-type, since these are the only form of meta-pointer.

**tree-arguments and tree-statements**

Relatively arbitrary segments of program code may be passed as pre-parsed syntax trees to meta-functions as in the meta-function call above and as initializers for meta-variables. These arbitrary code segments must satisfy the syntax of a tree-argument when used in a comma-separated context, such as a meta-function call. In the more general whitespace-separated context the tree-statement syntax must be satisfied.

```
tree-argument-list:
  tree-argument
  tree-argument-list , tree-argument

tree-argument:
  tree-statement
  unterminated-tree-argument

tree-statement:
  terminated-tree-argument
  unterminated-tree-argumentopt ;

compound-tree-statement:
  ( tree-statement-seqopt )

tree-statement-seq:
  tree-statement
  tree-statement-seq tree-statement
```

The constructs that may contribute to a tree-argument or a tree-statement are split into two categories.

terminated-tree-arguments have an inherent lexical termination, enabling parsing without any lookahead

```
namespace X {} // namespace-definition does not need a ;
```
Unterminated-tree-arguments require lookahead to the following punctuation:

```cpp
enum Y {} // unterminated and so indeterminate
enum Y {}; // ... enum-specifier
enum Y {} a = 0; // ... simple-declaration
```

Terminated-tree-argument:

- `asm-definition`
- `compound-tree-statement`
- `control-statement`
- `declaration-statement`
- `explicit-instantiation`
- `explicit-specialization`
- `expression-statement`
- `file-dependency-declaration`
- `file-placement-declaration`
- `filespace-declaration`
- `function-definition`
- `include-declaration`
- `linkage-specification`
- `namespace-alias-definition`
- `namespace-declaration`
- `namespace-definition`
- `template-declaration`
- `using-declaration`
- `using-directive`
- `auto meta-class-declaration`
- `auto meta-control-declaration`
- `auto meta-expression-statement`
- `auto meta-function-definition`
- `auto meta-variable-declaration`

Unterminated-tree-argument:

- `access-specifier`
- `accessibility-specifier`
- `base-specifier`
- `built-in-type-id`
- `class-specifier`
- `condition`
- `cv-qualifier`
- `decl-specifier`
- `enum-specifier`
- `enumerator-definition`
- `expression`
- `filespace-specifier`
- `function-try-block`
- `handler-seq`
- `initializer-clause`
- `mem-initializer`
- `parameter-declaration`
- `reserved-word`
- `simple-type-parameter`
- `storage-class-specifier`
- `template-argument`
- `template-class-specifier`
- `auto meta-class-specifier`

When used as a `tree-argument` in the comma-separated `tree-argument-list` the extra semicolon is optional for the unterminated productions.

```
$F(int a;, int b, c, d;);
```

When used as a `tree-statement` as the initializer of a meta-variable the semicolon is mandatory for the unterminated productions.

```cpp
auto statement s = { a(); b(); c(); };  
auto declaration d = int a; 
auto identifier i = if;
```

The trailing semicolon satisfies syntactic requirements, but has no semantic meaning. There is therefore no semantic difference between the `expression` argument in:

```
$meta_something(a=5)
```
and the expression-statement argument in:

```cpp
$m\text{eta\_something}(a=5;)
```

The argument is parsed as the more flexible expression, which may be converted to an expression-statement if required to satisfy some meta-type constraint, such as the formal meta-type of the meta\_something parameter. They may be used interchangeably as tree-arguments.

**Ambiguity**

There is an ambiguity between multiple comma-separated components of a specific tree-argument and multiple tree-arguments. The meta-function invocation

```cpp
$\text{f}(\text{int\ a, b, c, if\ (d)\ e, f;\ else\ g, h};)
```

comprises four arguments, the fourth of which is overlined and obviously a selection-statement, however the earlier three could be an init-declaration-list, a parameter-declaration-clause, or perhaps an exception-specification followed by a two element expression. The ambiguity is not resolved until the actual arguments are associated with formal parameters. When semantic interpretation occurs, the declaration of \text{f} may be found to be:

```cpp
\text{auto\ statement\ f}(\text{parameter\ declaration\ p1, identifier\ p2[], statement\ p3})
```

The available arguments are then associated from left to right, associating as many arguments as possible with one parameter before advancing to the next parameter. \text{int\ a} is associated with the parameter-declaration \text{p1}. The \text{p2} formal parameter can associate with an exposed list of identifiers (see Section 4.1.5). The second and third arguments are syntactically valid identifiers and so the list comprising \text{b} and \text{c} is associated as the list for \text{p2}. Finally the fourth argument is associated with \text{p3}.

The ambiguity resolution resulting from unconditionally associating arguments with the left-most parameter is consistent with similar disambiguation policies such as §5.3.4-2 for trailing *'s in new-declarators.

In practice, meta-functions with argument lists that comprise an exposed list, followed by further arguments, should be avoided since left to right association may prevent subsequent arguments being passed. For instance, it is not possible to pass an identifier representing a simple expression-statement as the fourth argument to \text{f} above since the identifier would be associated with the preceding exposed list parameter.

It is not possible to solve the problem by introducing parentheses or braces, since this would further overload existing punctuation and introduce ambiguities with respect to their existing meaning.

**Syntax coverage**

The productions listed for terminated-tree-argument and unterminated-tree-argument cover most of the C++ constructs and exhibit significant redundancy. In practice it is more helpful to identify what is not covered, why it is not covered, and how the limitation can be worked around.

The limitations represent very minor limitations on the language extension. They do not introduce any C++ incompatibility.

- an anonymous bit-field cannot be specified because a labeled-statement is unconditionally preferred
  - give the bit-field a name or
  - provide an access-specifier or
  - use a class or typename prefix for the type
do followed by semicolon is presumed to be the start of an iteration statement, so do as a reserved-word must not be followed by semicolon
- omit the semicolon in a tree-argument
- use "do" and implicit (or explicit) string to identifier conversion

operator followed by a comma is presumed to be the sequencing function name, so operator as a reserved-word cannot be followed by a list-separating comma
- use operator;

meta-declarations in tree-arguments cannot be specified without a leading auto to avoid ambiguity with conventional declarations and statements.
- use auto prefix for all meta-declarations
- omit auto prefix for all normal declarations.

major punctuation defines lexical structure and so cannot be passed as a punctuation argument. This affects: {

3.1.7 Design Rationale
Lists and Trees
Support for list (or more strictly tree structured) arguments as described above is convenient and the disambiguation rule solves some inelegant practical problems, at the expense of being a little cute.

One simpler alternative policy identifies maximal length sub-list elements unconditionally without reference to the invoking context. This unfortunately requires the additional semicolons (shown with an overline) to prevent unwanted grouping:
\[ f(\text{int } a, b, c, \text{if (d) e, f; else g, h;}) \]
and more unacceptably requires three identifiers to be passed as:
\[ g(a, b, c) \]

Another alternative policy prohibits transparent passing of list elements, which is obviously less flexible. ()'s or {}'s must then be used to encapsulate the lists for those syntaxes where bracketing is permitted. A syntax extension is needed for other syntaxes.

All three policies are difficult to implement in practice because a generalised superset grammar needs to treat a potential constructor argument list
\[ a : b(c), d(e) \]
as an unresolved prefix that could be a bit-field followed by a variable, or a constructor and initializers. The solution in Appendix C exploits the generalised monomorphic characteristic of each argument to avoid premature parsing decisions.

Comma-separated lists of arguments that may themselves comprise comma-separated lists are therefore supported, since support imposes fewer programming constraints and no extra implementation difficulty.

auto declaration f(\text{parameter\_declaration\_clause \text{p}});
can be invoked as:
\[ f(\text{int } a, \text{char } *b, \ldots); \quad \text{// Ellipsis token as third argument:} \]
Sequences

Sequences (unseparated lists) of arguments are not supported to avoid a syntax ambiguity between a sequence and further syntax as the initializer of a meta-variable list:

```cpp
auto statement meta_variable[] = stmt1; stmt2; stmt3;
```

This is not a limitation since a sequence can be expressed in its compound form.

```cpp
auto statement meta_variable[] = { stmt1; stmt2; } stmt3;
```

Labels

The goto form of a labeled-statement:

```cpp
label : statement
```

and an anonymous bit-field

```cpp
type : 5 ;
```

are ambiguous and have dissimilar syntax tree structure; a label decorates a statement, whereas a bit-field width decorates a declarator.

It does not seem worth significant effort to unify these rare constructs. The label form is therefore excluded from a meta-control-declaration to avoid changing the semantics of a member-declaration. However, when a bit-field is added to an init-declarator and consequently to a statement, the syntactic conflict must be resolved to the label. The same conflict arises for a tree-argument and so the same unconditional resolution of identifier : as a label is made for compatibility with C++. This behaviour is a little surprising and so the equivalent ambiguity for tree-literal : is resolved to the more useful bit-field.

3.1.2 Names

FOG syntactic analysis has to be context-free while operating on both potential and actual declarations, since it is inevitable that the context cannot be known for potential declarations, for which complete identification of all enclosing scopes is missing.

Chapter 5 shows how the traditional template and type name context dependencies are eliminated using the superset grammar.

Since FOG does not use template or type name information, it is necessary to generalise the syntax of names to eliminate the productions dependent upon semantic name classification:

```cpp
typedef-name:
  identifier

namespace-name:
  original-namespace-name
  namespace-alias

original-namespace-name:
  identifier

namespace-alias:
  identifier

class-name:
  identifier
template-id

type-name:
  identifier

template-name:
  identifier

template-id:
  template-name < template-argument-list >
```
Much simpler unclassified name productions are used instead:

```
id:
  identifier
  identifier < template-argument-list >
  template identifier < template-argument-list >
```

```
nested-id:
  id
  id : : nested-id
```

```
scoped-id:
  : :opt nested-id
```

A straightforward conversion to a context-free grammar replaces the C++ usage of `xxx-name` by `id`.

A further syntax generalisation permits declarations to appear as interface style declarations inside class braces or implementation style declarations outside. This requires that `xxx-name` be replaced by `scoped-id`.

In addition to the regular names based upon identifiers and templates, there are the special function names that involve punctuation characters:

```
special-function-id:
  id
  conversion-function-id
  operator-function-id
```

```
nested-special-function-id:
  special-function-id
  id : : nested-function-special-id
```

```
scoped-special-function-id:
  : :opt nested-special-function-id
```

These new non-terminals replace the existing names for declarators, expressions and declarations:

```
declarator-id:
  ->opt id-expression
  ->opt nested-name-specifier opt type-name
  scoped-id
  scoped-special-function-id
```

```
primary-expression:
  literal
  this
  ->identifier
  ->operator-function-id
  ->qualified-id
  ( expression )
  id-expression
declarator-id
```

```
using-declaration:
  using typename_opt ->opt nested-name-specifier unqualified-id declarator-id ;
  using ->opt unqualified-id ;
```

Type specifiers are simplified:

```
simple-type-specifier:
  ->opt nested-name-specifier opt type-name scoped-id
  char
```

```
enum-specifier:
  enum identifier_opt scoped-id_opt ( enumerator-list_opt )
```

2. Resolution of the context-dependency for `<` is discussed in Section 5.8.

3. An unqualified destructor is ambiguous with respect to a complement expression, and very rarely valid. In practice it is easier to exclude the local destructor from `special-function-id` and introduce it only once nested in `nested-special-function-id`.
elaborated-type-specifier:
  class-key <<>> nested-name-specifier identifer scoped-id
  enum <<>> nested-name-specifier identifer scoped-id
  typename <<>> nested-name-specifier identifer scoped-id
  typename <<>> nested-name-specifier identifer template-argument-list

class-head:
  class-key identifier scoped-id base-clause
  class-key nested-name-specifier identifer base-clause

base-specifier:
  <<>> nested-name-specifier class-name scoped-id

Namespace definitions are simplified dramatically after eliminating semantic
distinctions from the syntactic grammar:

namespace-definition:
  named-namespace-definition
  unnamed-namespace-definition
  namespace scoped-id (namespace-body)

named-namespace-definition:
  original-namespace-definition
  extension-namespace-definition

original-namespace-definition:
  namespace identifier (namespace-body)

extension-namespace-definition:
  namespace original-namespace-name (namespace-body)

unnamed-namespace-definition:
  namespace (namespace-body)

Impact

The simplified naming defers semantic constraints so that the grammar defines
just syntax. As a result, the grammar covers much that is illegal; in particular,
every occurrence of a templated-name requires semantic validation and very
occasionally correction as well.

3.1.3 Syntax Generalisation

Perhaps the most significant simple language change in FOG is the relaxation of
the One Definition Rule (§3.2) described in Section 4.4, so that multiple
declarations compose to give an extended declaration rather than an error. This
change is entirely semantic.

A further generalisation is the unification of syntax to combine interface and
implementation declarations. This is almost entirely semantic, since the C++
grammar for functions and variables embraces most of the required FOG
generalisations within syntax that is only semantically invalid in C++.

Some minor syntax generalisations to resolve anomalies are described in this
section. Some slightly more significant enhancements appear in the next section.

3.1.3.1 Forward declaration for namespace

A FOG declaration may first appear as:

int Scope::name;

ame is added to Scope, which must be a previously declared namespace or class.
C++ already allows a prior declaration of a class to be provided by
class Scope;

A similar declaration for a namespace should be possible. So FOG adds:

declaration:
  namespace-declaration

// Extension of
namespace-declaration:
    namespace scoped-id ;

Impact

This is an unnecessary extension. It just provides consistency without introducing any problems. A namespace could alternatively be forward declared by:
    namespace Scope {}

3.1.3.2 access-specifiers as decl-specifiers

The public/protected/private accessibility of a member declaration is necessarily specified in C++ by a preceding and relatively independent access-specifier. Other characteristics such as inline or static are specified by decl-specifiers.

    class MyClass
    {
        protected:
            inline static void protected_method();
    };

In order to avoid declarations relying on surrounding context and causing indeterminacies during composition, FOG allows the access-specifier to be specified as part of a declaration.

    decl-specifier: // Extension of
        access-specifier

The presence of an access-specifier as part of a decl-specifier affects only the specified declaration. Subsequent declarations continue to use the prevailing default accessibility.

    class MyClass
    {
        protected:
            public virtual void public_method();
            inline static void protected_method();
    };
    private void MyClass::private_method();

Impact

This generalisation introduces a syntax ambiguity whereby public: could introduce an implicitly int anonymous bit-field. There is no such thing and the ambiguity is resolved.

3.1.3.3 Pure-virtual

C++ allows a pure-virtual function to be declared

    class MyClass
    {
        virtual int f() = 0;
    };

and then implemented

    int MyClass::f() { return 0; }

but does not allow the two declarations to be combined, so that the implementation is inlined within the interface. This strange prohibition is removed, in order to allow a complete function definition in one FOG declaration.

function-definition:
    decl-specifier-seqopt declarator pure-specifieropt ctor-initializeropt function-body
    decl-specifier-seqopt declarator pure-specifieropt function-try-block
The following is invalid C++, but valid FOG.

```cpp
class MyClass
{
    virtual int f() = 0 { return 0; }
};
```

**Impact**

The extra term poses no additional problem to the generalised superset grammar. It could pose significant problems to a conventional grammar. This may be the reason for the current exclusion.

**3.1.3.4 !static**

The static keyword specifies whether a member function or variable is associated with a class or an instance. This is a fundamental programming decision that should not normally be changed as a result of composing multiple declarations.

In C++, the class usage is explicitly specified, the instance usage is implicit. In order to make the instance usage explicit as well, and so ensure that any inconsistent composition is detected, the inverse behaviour of certain keywords can be specified explicitly.

```cpp
storage-class-specifier:  // Extension of
                    static
                    !static
```

When composing declarations it is only necessary to supply enough of the declaration name to identify the declaration unambiguously. The remaining parts of multiple declarations compose. Unspecified static may compose with either static or !static. However an attempted composition of static and !static gives an error.

**Implementation**

The ! operator in an expression using a generalised parse of a name can be ambiguous with only one token of look-ahead.

```cpp
(type) ! static a  // Cast of non-static a
(type) ! static a  // Cast of complement of static a
```

Neither interpretation is semantically valid, so the generalised name parsing excludes decl-specifiers appearing as prefixes.

!const and !volatile were originally supported but withdrawn for reasons described in Appendix F.1.1.

**3.1.3.5 !inline**

In C++, the request to inline function may be explicitly provided in the interface or in the implementation, depending on whether the inline keyword is associated with the interface or the implementation declaration. FOG merges these declarations and so the presence or absence of inline in a FOG definition cannot express the three C++ alternatives. The syntax of inline is therefore extended to express all three intents explicitly.

```cpp
function-specifier:  // Extension of
                    inline
                    !inline
                    !inline/implementation
                    !inline/interface
```

29-June-2001
no inline

In the absence of any form of inline keyword, FOG must decide whether functions are to be defined in the interface or implementation file.

class X
{
    int f(int a) { return a-1; }
};

Functions are placed in the implementation unless all the following criteria are satisfied:

- The function is declared as above within class braces.
- The function body code is simple, on the basis of a complexity estimate formed by counting the number of accesses and operations and comparing it with a command line threshold that has a default value of 10.
- The function is not virtual.
- The function is not static at namespace scope.

inline

The default form of inline is interpreted in a context sensitive fashion in order to provide compatibility with common C++ coding styles:

class X
{
    inline void f1();
};
inline void X::f2() { /* ... */ }

An inline appearing within class braces as in f1 is conditionally inlined within the interface, whereas a function such as f2 with only an inline outside class braces is conditionally inlined in the implementation.

inline/interface

Requests that the function be inlined in the interface.

inline/implementation

Requests that the function be inlined in the implementation (and therefore not in the interface).

!inline

Specifies that the function is not to be inlined in the interface or in the implementation.

Impact

Parsing of a generalised name accepts a decl-specifier as a suffix and so namel inline / interface (name2);
could be an expression involving a division and a function call or a more explicitly positioned inline function declaration. This is a false conflict since inline can never occur in an expression, so inline followed by / is unconditionally resolved to an extended form of inline.

3.1.3.6 !virtual

The syntax for virtual is expanded to support its negation and to give an alternative and less cryptic way of specifying pure-virtual.
function-specifier: // Extension of
    virtual
    !virtual
    virtual / pure

Impact

The same ambiguity arises and the same resolution is used as for inline in Section 3.1.3.5.

3.1.4 Syntax Enhancements

In Section 3.1.3, generalisations and very minor enhancements were introduced to support composition consistently. In this section more significant enhancements are introduced again in support of composition.

3.1.4.1 Default member initializer

In C++, it is easy for a constructor to leave member variables of simple types uninitialized. For classes with a non-trivial number of variables or constructors this can be a maintenance problem.

FOG allows one set of compositions to add member variables and another to add constructors. There is therefore ample scope for two compositions to aggravate the problem of uninitialized member variables, so FOG allows a default initializer to be specified for member variables.

The (C++) syntax for the declaration of global variables and for the definition of any variable outside a class supports initialization:

```
simple-declaration:
    decl-specifier-seq_opt init-declarator-list_opt :

init-declarator-list:
    init-declarator
    init-declarator-list , init-declarator

init-declarator:
    declarator initializer_opt

initializer:
    = initializer-clause
    { expression-list }

initializer-clause:
    assignment-expression
    ( initializer-list_opt )
    ( )
```

This is incorporated into the more restrictive syntax for definition of member variables within classes.

```
member-declaration:
    decl-specifier-seq_opt member-declarator-list_opt :

member-declarator-list:
    member-declarator
    member-declarator-list , member-declarator

member-declarator:
    declarator pure-specifier_opt
definite-constant-initializer
    declarator constant-initializer_opt
    identifier_opt : constant-expression

constant-initializer:
    = constant-expression
```

The default initialization of member variables may be declared:
class MyClass
{
    bool _satisfies_predicate = false;
    int _usages[3] = { 1, 2, 3 };
    That _that( *this);
};

Semantics

The default initializer provides an explicit initial value for use in every constructor that does not provide an initializer. The value is never used in a copy constructor, since a copy constructor provides an implicit initializer for each member.

C++ does not support direct initialization of array members during construction. The array initialization must therefore be synthesised by code placed at the start of the constructor body, and so the construction order of array members is not defined.

Syntax Ambiguity

The constructor form of \textit{initializer} introduces the function-declaration/constructor-invocation ambiguity into a class (Section 5.5.3.2).

\begin{Verbatim}
\begin{verbatim}
class A
{
    int a(a_type); // member function declaration
    int b(not_a_type); // member variable and initialization
};
\end{verbatim}
\end{Verbatim}

It is amenable to exactly the same resolution as outside a class. Resolution favours the declaration perspective and so preserves upward compatibility with existing C++ code.

The assignment form is clearer but unable to express multi-argument construction directly. (The result of an explicit constructor call can be used less directly.)

3.1.4.2 \textit{gcc} indexed array initializer

In order to support composition of arrays usefully, it is necessary to be able to specify the location of array initializers. The \textit{gcc} [Stallman98] indexed array initialization syntax is therefore supported:

\begin{Verbatim}
\begin{verbatim}
initializer-clause:
    assignment-expression
    [ constant-expression ] assignment-expression
    [ initializer-list_opt ]
};
\end{verbatim}
\end{Verbatim}

Each array initializer may be prefixed by an expression specifying its position.

Implementation

Parsing of this syntax causes no problems in a precise grammar, however an ambiguity arises in the superset grammar, since a prefix \textit{[]} is already recognised to avoid a conflict for \textit{delete} \textit{[]} \textit{cast-expression}. Generalising the solution to the \textit{delete} \textit{[]} conflict, so that \textit{[]} \textit{constant-expression} is parsed as another form of cast, accepts the indexed initializer without extra syntax.

\begin{Verbatim}
\begin{verbatim}
abstract_expression:
    parenthesis_clause // Like '(' expression_opt ')'
    '' constant_expression ''
\end{verbatim}
\end{Verbatim}

\begin{Verbatim}
\begin{verbatim}
abstract_expression:
    unary_expression
    abstract_expression cast_expression
\end{verbatim}
\end{Verbatim}
3.1.4.3 compound-declaration

\[
\text{compound-declaration:} \\
\{ \text{declaration-seqopt} \}
\]

compound-declaration is introduced to support polymorphic use of multiple declarations as a single declaration, in the same way as multiple statements can behave as a single statement. However this is a purely lexical grouping; no nested declarative region is defined. The new region has no name; treating it as a declarative region would prevent access to names forming part of a declaration. The analogy is therefore with an anonymous union, whose names are externally visible, rather than with a function block, whose names are local.

3.1.4.4 using

A C++ using-declaration supports the re-use of the name of a base-class declaration in a derived class.

```cpp
class Lock
{
  //...
  public:
    bool is_locked() const;
};
class LockableWidget : public Widget, private Lock
{
  // ...
  public:
    using Lock::is_locked;  // Make private base class name public.
};
```

FOG generalises this concept in a re-using-declaration to support re-use of the name of any declaration and re-use (extension) of an existing declaration.

```cpp
class Debug
{
  //...
  public:
    static bool diagnose();
};
class LockableWidget : public Widget, private Lock
{
  // ...
  public:
    // Incorporate Lock::is_locked() as is_locked()
    using Lock::is_locked;
    // Incorporate Debug::diagnose() as show_conflicts()
    using Debug::diagnose show_conflicts;
};
```

The extended syntax supports:
- signature re-use
- function placement
- built-in functionality extension

The specific syntax for a using-declaration is removed and covered by adding using to decl-specifier.

```cpp
using-declaration:
  using typename_opt :opt nested-name-specifier unqualified-id
  using : opt unqualified-id

decl-specifier:
  using

// Extension of
```

This generalization covers the existing syntax as part of the declarations now contributing to:
simple-declaration:
    decl-specifier-seq\_opt init-declarator-list\_opt ;

function-definition:
    decl-specifier-seq\_opt declarator pure-specifier\_opt ctor-initializer\_opt function-body
    decl-specifier-seq\_opt declarator pure-specifier\_opt function-try-block

A long form of the new re-using-declaration

    using A::b C::d;       // Use signature of A::b to define C::d

is present when

- decl-specifier-seq\_opt includes using
- decl-specifier-seq\_opt includes a type-specifier

A short form of the new re-using-declaration,

    using A::b;           // Use signature of A::b to define b

that subsumes the existing syntax and functionality of using-declaration, is present when

- decl-specifier-seq\_opt includes using
- decl-specifier-seq\_opt includes no type-specifier

Signature re-use

In deep polymorphic, or wide isomorphic object hierarchies, it is common for the same function signature to recur in many, if not all, classes in the hierarchy. This incurs a little lexical redundancy, and acts as a barrier to code evolution; a change to a function signature may involve a very substantial amount of editing. FOG allows a function signature to be defined once and re-used many times, so that changes to the function declaration can be made in one place.

class A
{
    public void protocol(int a, double b) const { /* ... */ }
};
class B : public A {
{
    using B::protocol { /* ... */ }           // Short form
};
    using A::protocol B::protocol { /* ... */ }   // Long form

Both using lines contribute code to the function

    public void B::protocol(int a, double b) const;

The short-form creates or extends the entity that must already be unambiguously visible with the name B::protocol Since A::protocol is visible as B::protocol, the function B::protocol is therefore created with the same signature as A::protocol.

The long-form uses the signature of A::protocol (which could be a typedef) to create or extend the function named B::protocol. Overload resolution of B::protocol is performed using the signature from A::protocol to select one precisely matching alternative. There is no overload resolution for A::protocol, however distinct names can be associated with each overload by using typedefs to define A::protocol.

With either form, the signature of A::protocol is being re-used, enabling a change to the signature to be made in one place. However re-used signatures must also re-use parameter names, since there is no need for them to be respecified. This is a little inconvenient, since parameter names now have a more
global import. However there may also be some advantages to the requirement for consistent parameter names in closely related functions.

**Function placement and tuning of built-in functionality**

The C++ compiler may automatically generate code for

- default constructor
- copy constructor
- assignment operator
- destructor
- dereferencing operator (unary &)

With the exception of the destructor, whose functionality is only extensible, any attempt to modify the auto-generated code requires manual re-implementation of the entire functionality. Trivial modifications such as specification of the access, inlining, virtual or placement in a specific file should be possible.

The extended *re-using-declaration* syntax supports this, since re-using a function involves composition with its existing functionality. Therefore within a class declaration, where `@Scope` resolves to the class name:

```cpp
using virtual -@Scope;
```

defines the destructor as virtual, without affecting any other declaration that specifies accessibility.

```cpp
using !inline -@Scope;
```

forces an out-of-line implementation of the destructor avoiding any unwanted include file dependencies that might result from the default inline version.

```cpp
using protected operator=(const @Scope&) { _assigns++; }
```

specifies that the assignment operator is to have protected access and adds a counter update to the existing (default) functionality.

```cpp
using @Scope(const @Scope&) : _share_count(1) {}
```

overrides the initialization of one member in a copy constructor, leaving other members unaffected and therefore retaining their default member-wise copy, whereas:

```cpp
@Scope(const @Scope&) : _share_count(1) {}
```

specifies an explicit initialization of one member, and a default initialization of all other members.

**Semantics**

The *re-using-declaration* may be elaborated with an *access-specifier*, *decl-specifiers*, *parameter-declaration-clause*, default arguments, *initializers*, *function-bodies* and *object-statement-clauses*. These compose with existing declarations as described in Section 4.4.8.

The long form of *re-using-declaration* applies functionality from the source declaration identified in the *decl-specifier* to the target declaration identified in (each) *init-declarator* or *declarator*. Source and target declarations may be independently scoped and resolved with respect to the surrounding declarative region.

The short form of *re-using-declaration* subsumes the existing *using-declaration*. It specifies both source and target declarations as the *init-declarator* or *declarator*. For compatibility with the existing *using-declaration*, any specified scope must serve to locate the source declaration. The target declaration is therefore necessarily part of the surrounding declarative region.
For both forms, the name provided as the source declaration must be visible within the surrounding declarative region. decl-specifiers forming part of the source declaration are copied to the target declaration, except that conflicting decl-specifiers are discarded in favour of those forming part of the re-using-declaration.

implementation

When an overloaded signature is re-used, it is not clear which signature is required. The deprecated ARM C++ access-declaration suffered from this problem too. In FOG, the problem was originally solved by introducing a nick-naming capability so that overloads could be given alternate names:

```cpp
MyClass::MyClass() overload=default_constructor
MyClass::MyClass(const @Scope&) overload=copy_constructor
MyClass& MyClass::operator=(const @Scope&) overload=assign
using default_constructor { /* ... */ }
```

However it was realised that this was unnecessary as well as clumsy. A more generalised use of a typedef is better:

```cpp
auto namespace HandySignatures { //4
    typedef default_constructor();
    typedef copy_constructor(const @Scope& thatObject);
    typedef @Scope assign(const @Scope& thatObject);
};
```

supports use as

```cpp
using HandySignatures::assign MyClass::operator=( /* ... */ )
```

The typedefs stretch the syntactic legality of C++, but only define what was previously meaningless. The constructor typedefs lack a type-specifier, and so there is a potential ambiguity for

```cpp
typedef a(b());
```

between

- the constructor typedef of a taking a pointer to function argument
- the redundantly parenthesised function b returning a.

Only the latter is valid in C++. The former is a new alternative interpretation that must be ignored. The ambiguity does not arise when the missing parameter names are specified as is necessary for the definition of a constructor signature to be useful.

```cpp
typedef a(b (*)(c));
```

The typedef approach supports sharing of signatures independent of inheritance. The nickname approach needed further elaboration to support this.

The parameter names are retained as part of the typedef so that they form part of the re-used signature.

3.1.4.5 using template

A using keyword may prefix a template-specialization to indicate that the subsequent declarations re-use and so compose with, rather than replace, the declarations from the less specialized template.

```cpp
template-declaration:
    export_opt using using template < template-parameter-list > declaration
```

4. class could be used rather than auto namespace. The use of a meta-namespace just serves to eliminate unnecessary declarations emitted for compilation.
explicit-specialization:
  using\_opt template < > declaration

Impact

This usage introduces a parsing lookahead problem with respect to
  using template name < args >;

The lookahead is eliminated by accepting any decl-specifier rather than just using
during syntactical analysis and then rejecting the spurious alternatives during
semantic analysis.

3.1.4.6 Object-statement-scopes

Specialized placement of a declaration in a specific file, or accurate resolution of
dependencies may require use of FOG extensions to annotate the declaration.
Adding the additional syntax to support these declarations is rather difficult, see
Appendix F.1.

When composing declarations, it may be necessary to share meta-context
between contributions. Defining meta-variables at class scope to share this
context can lead to unpleasant interactions when similar composition policies or
meta-programs affect more than one function.

Both of these problems are resolved by the introduction of the concept of an
object-statement-scope: a declarative region exclusively for use at meta-compilation-
time. Meta-declarations may be placed in this scope, and shared between
contributions to the object. The scope is defined by an object-statements-clause, within
which, annotations can be placed without introducing syntactical conflicts. It is
only necessary to identify one syntax extension that does not conflict with existing
syntax. This is achieved by using : { and } to delimit the region.

init-declarator:
  declarator initializer\_opt object-statements-clause\_opt

member-declarator:
  declarator pure-specifier\_opt object-statements-clause\_opt
  declarator constant-initializer\_opt object-statements-clause\_opt
  identifier\_opt : constant-expression object-statements-clause\_opt

object-statements-clause:
  : { object-statement-seq\_opt }

object-statement-seq:
  object-statement
  object-statement-seq object-statement

object-statement:
  initializer ;
  function-used-block
  file-dependency-declaration // Appendix F.4.5
  file-placement-declaration // Appendix F.4.3
  filespace-declaration // Appendix F.4.4
  meta-control-declaration // Section 3.1.5.8
  auto meta-control-declaration // Section 3.1.5.10
  auto meta-expression-statement // Section 3.1.5.6
  auto meta-function-definition // Section 3.1.5.5
  auto meta-variable-declaration // Section 3.1.4.8
  derived-clause object-statement
  derived-clause : { object-statement-seq\_opt } // Section 3.1.4.8

The rule involving an initializer is only semantically valid for a variable-statement-
scope (an object-statement-scope associated with a variable).
The rule involving a function-used-block is only semantically valid for a function-
statement-scope (an object-statement-scope associated with a function).

Section 4.3.2 describes the revised search order for meta-name resolution within
the object, then the object scope and then class scopes.
Impact

The syntax is unambiguous because colon is only followed by \( \) in C++ in the
limited context of a \textit{label-statement} of the form

\[
\text{identifier} : \{ \text{statement-seq}\_\text{opt} \}
\]

The usage as an \textit{object-statements-clause} has a more substantial prefix.

Care is required to avoid a shift-reduce conflict with only one token of lookahead.

3.1.4.7 Function-statement-scopes

Object-statement-scopes or more specifically function-statement-scopes are
essential for annotating the contributions to a composed function. Each
contribution may have its own constructor initializers and function body re-
interpreted in derived classes in accordance with a derivation rule.

The contribution may be further annotated to define its include \textquote{file} dependencies,
its positioning relative to other contributions and a position in the overall function
structure.

\[
\text{function-used-block:}
\begin{align*}
\text{ctor-initializer} ; \\
\text{ctor-initializer}\_\text{opt} \text{function-body} \\
\text{function-try-block} \\
\text{using file-id-list function-used-block} \end{align*}
\]

\textit{using file-id-list function-used-block} // Appendix F.4.5

Note that a function-statement-scope is a syntactic extension of \textit{init-declarator} and
consequently requires a trailing semicolon to form a \textit{simple-declaration}.

(Implementation as a \textit{function-definition} leads to challenging conflicts.)

\begin{verbatim}
friend ostream& operator<<(ostream& s, const MyClass& myClass) :
{ using ostream { /* ... */ return s; } }
\end{verbatim}

Usage of commas to separate multiple declarations with \textit{object-statements-clauses}
should be considered extremely bad style, however an outright prohibition
appears to add a slight complexity rather than a simplification.

Segments

Program segments identify five distinct domains of composition. Code
contributions are composed independently for each segment, but emitted as one
contiguous code body. The segments are emitted in the order listed for:

\[
\text{segment:}
\begin{align*}
\text{entry} \\
\text{pre} \\
\text{body} \\
\text{post} \\
\text{exit} \end{align*}
\]

The \textit{body} is the default segment in which code is normally placed.

\textit{entry} and \textit{exit} segments bracket the rest of the code. The intention is that the
\textit{entry} segment contain any required declarations and the \textit{exit} segment a \textit{return}
\textit{statement}.

\textit{pre} and \textit{post} are intended for passive code that wraps pre-condition and post-
condition checks or diagnostics around the active part of the function.

These default policies are informal. Composition of function bodies and the
redefinition of function structure is described in Section 4.4.8.
3.1.4.8 Derivation Rules

A derivation rule determines how a potential declaration is interpreted so as to automatically generate derived declarations. A declaration is conventionally supplied for a specific scope, which is referred to as the root scope of that potential declaration. Derivation rules consider the inheritance hierarchy at and below that root scope, evaluating a predicate expression in the meta-name-space of each class to determine whether the derivation rule is enabled.

derived-clause:
    derived ( meta-conditional-expression )

The following pair of declarations

```cpp
protected inline static int Class::static_inheritance_depth()
{
    derived (is_root())
    { return 0; }
    derived (!is_root())
    { return @(Super)::static_inheritance_depth() + 1; }
}

public virtual int Class::dynamic_inheritance_depth() const
{
    derived (true)
    { return static_inheritance_depth(); }
}
```

define a pair of functions that are implemented for Class and all its derived classes so that invocation of dynamic_inheritance_depth() on p, a pointer to a Class object:

```cpp
Class& p = ...;
... = p.dynamic_inheritance_depth();
```

returns the actual inheritance depth of p. The implementation of the static inline function static_inheritance_depth() for the root scope, where the is_root() predicate is satisfied, provides a function body that just returns zero. The implementation for derived classes, where the !is_root() predicate is satisfied, returns the super-class depth +1. The true predicate is always satisfied, and so dynamic_inheritance_depth() is a virtual function for the root scope and all derived classes, which ensures that the correct depth is returned.

The :{} object-statement-scope contains a number of object-statements. The meta-function invoked within the derivation predicate is therefore resolved with respect to the prevailing meta-object to locate the built-in object_statement::is_root(). As will be seen in subsequent examples a Scope. prefix may be used to resolve the meta-function with respect to the scope.

The predicate meta-conditional-expression may involve user-defined or built-in meta-functions and meta-variables. Two groups of built-in functions are provided primarily for use as derivation rule predicates. One group defines structural predicates upon the inheritance tree. The second group defines abstract predicates dependent upon the position of pure virtual functions in the inheritance hierarchy.

Structural Predicates

**true**

Specifies that the declaration is to be applied throughout the inheritance hierarchy: in the root scope and all its derived classes.

**object_statement::is_root()**

Specifies that the declaration is to be applied to the root scope. This is the default derivation rule and ensures upward compatibility with C++.
Meta-compilation for C++

FOG Grammar

object_statement::is_leaf()
scope::is_leaf()

Specifies that the declaration is to be applied to all classes in the inheritance hierarchy that have no derived classes. In the degenerate case, this may be just the root scope.

Leaf-ness is a class (or struct or union) property and so the information is provided by scope::is_leaf(). object_statement::is_leaf() is provided for convenience; it just delegates the inquiry to the scope.

Leaf-ness is independently determined from the types declared during each meta-compilation session. There is no overall global view across meta-compilation sessions, so if further derivation occurs unknown to one session, leaf-based decisions may be inaccurate in another.

More complicated conditions can be built-up using expression operators:

derived(!is_root() && !is_leaf())

selects all intermediate nodes in an inheritance hierarchy.

Abstract Predicates

Abstract predicates support efficient and appropriate generation of code depending on the presence of pure virtual functions. The !is_pure() predicate may be used to avoid generation of code that illegally attempts to new an abstract class. The is_boundary() predicate may place code just once at the inheritance boundary between abstract and concrete classes.

scope::is_pure()

Specifies that the declaration is to be applied to all classes in the inheritance hierarchy that have at least one pure virtual function.

scope::is_boundary()

Specifies that the declaration is to be applied to the least derived class in the inheritance hierarchy derived from the root scope for which there are no pure virtual functions.

function::is_boundary()

Specifies that the declaration is to be applied to the least derived class in the inheritance hierarchy derived from the root scope for which the virtual function is not pure.

Semantics

In the absence of a derivation rule, a declaration contributes to its root scope.

With a derivation rule, the declaration name is provisionally present in the root scope and all derived scopes. Once it can be determined that the derivation predicate cannot be satisfied, the declaration is disabled, and consequently is not emitted.

Some predicates, such as true or is_root(), can be evaluated immediately.

Other predicates, such as is_leaf() or Scope.is_pure() can be affected by further declarations or meta-programming and so cannot be evaluated promptly. Resolution is therefore automatically deferred until the code emission phase of meta-compilation. Deferred evaluation of user-defined predicates may be enforced by use of an @ operator to delay evaluation until the body is resolved during code emission.
The derivation predicate gates the body of a declaration

- *function-body* and return type of a function
- *ctor-initializers* of a constructor
- *initializer* and type of a variable
- value of a typedef

The body is not evaluated until the predicate has been resolved, thereby avoiding problems that might occur through use of invalid manipulations, such as a base-class (@Super) of a root scope.

Deferred evaluation of a body is only possible if the body is not used. References to the body must therefore also use the @ operator to defer evaluation. Direct use of the body such as the use of a typedef to define a function signature are an error if the body is not immediately resolvable.

**Implementation**

The presence of derivation rules dependent upon the abstract context of a class leads to a potential ambiguity:

```cpp
class Root {
    virtual void f1() = 0 :
        derived(@is_boundary())
    };

class Branch : public Root ;
class Leaf : public Branch ;
```

Which, if any, of `Root::f1`, `Branch::f1` and `Leaf::f1` should be implemented to define the boundary? The `Root` class has an explicit pure virtual, is clearly abstract and so cannot constitute a boundary. Implementation of none or one of `Branch::f1` and `Leaf::f1` gives a consistent behaviour.

FOG implements `Branch::f1` as a result of compiling classes in a least-derived order, and of making the presumption that an ambiguously pure/concrete class is concrete while evaluating the derivation predicates of its members. This ensures that practical problems yield a stable solution. Since the predicates may be arbitrary expressions, a predicate may introduce a contradiction for which an error message is produced.

**Historical Note**

Derivation rules were introduced in FOG before meta-programming. They offer little that cannot be achieved by meta-programming, but do so with a much more compact and manageable syntax for many common cases.

**3.1.5 Meta-Programming**

Meta-programming extends C++ run-time behaviour by providing very similar behaviour at (meta-)compile-time.

In order to introduce additional functionality without new reserved words, the existing syntax is heavily overloaded, which has the advantage of requiring very little new syntax to be learnt, but incurs the risk that the rather different behaviour may be overlooked.

The auto keyword is used to introduce meta-functionality. Its existing usage is only valid as a decl-specifier within a function and so all other usage of auto is retracted, at no cost to semantically valid programs.
The apparent loss of support for `auto` for local variables in functions is resolved by broadening the replacement `meta-expression-statement` syntax to cover the old usage and then recognising the old style usage during semantic analysis.

Integration

FOG introduces a number of new distinct syntaxes and generalises some existing syntaxes. In order to see how these integrate with the C++ grammar, the three contexts in which the C++ grammar supports a diverse range of syntaxes are presented.

namespace statements

Contributions to namespaces occur as a sequence of `declarations` within the top-level (unnamed global namespace), within namespaces, and within external linkages:

```
declaration-seq:
  declaration
  declaration-seq declaration

translation-unit:
  declaration-seqopt

namespace-body:
  declaration-seqopt

linkage-specification:
  extern string-literal { declaration-seqopt }
  extern string-literal declaration
```

member statements

Contributions to classes occur as a sequence of `member-declaration`

```
member-specification:
  member-declaration member-specificationopt
  access-specifier : member-specificationopt

class-specifier:
  class-head { member-specificationopt }
```

function statements

Contributions to function bodies occur as a sequence of `statement`

```
function-body:
  compound-statement

compound-statement:
  { statement-seqopt }

statement-seq:
  statement
  statement-seq statement
```

C++ contributions therefore arise as a `declaration`, `member-declaration` or `statement`. FOG adds a fourth context in which `object-statements` contribute to an `object-statement-clause` in order to qualify the behaviour of a function, typedef or variable (see Section 3.1.4.6). A fifth generic context arises when almost arbitrary syntax is parsed as a `tree-statement` to form a `tree-literal` (see Section 3.1.1.6).

Meta-functionality extends each of the three existing contexts and contributes to the two new contexts. The capabilities of each context are difficult to grasp from the grammar and so the contributions to each context are tabulated in Table 3.1.
<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>declaration</th>
<th>member-declaration</th>
<th>statement</th>
<th>object statement</th>
<th>tree statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compound Statements</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compound-declaration</td>
<td>FOG</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compound-statement</td>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compound-tree-statement</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Control Statements and Declarations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>control-statement</td>
<td>C++</td>
<td></td>
<td></td>
<td>FOG</td>
<td></td>
</tr>
<tr>
<td>try-block</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>meta-control-declaration</td>
<td>auto</td>
<td>FOG</td>
<td>auto</td>
<td>FOG</td>
<td></td>
</tr>
<tr>
<td><strong>Expression Statements</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expression-statement</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>meta-expression-statement</td>
<td>auto</td>
<td>FOG</td>
<td>auto</td>
<td>FOG</td>
<td></td>
</tr>
<tr>
<td><strong>Declaration Statements</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>function-definition</td>
<td>C++</td>
<td></td>
<td></td>
<td>FOG</td>
<td></td>
</tr>
<tr>
<td>template-declaration</td>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>using-declaration</td>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simple-declaration</td>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'simple-member-declaration</td>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=assignment-expression</td>
<td>C++</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>= ( ... )</td>
<td>C++</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(expression-list)</td>
<td>C++</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=constant-expression</td>
<td>(C++)</td>
<td>C++</td>
<td>(C++)</td>
<td>(FOG)</td>
<td></td>
</tr>
<tr>
<td>bit-field</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Others</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>explicit-instantiation</td>
<td>C++</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>explicit-specialization</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linkage-specification</td>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>namespace-definition</td>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>asm-definition</td>
<td>C++</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>namespace-alias-definition</td>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>using-directive</td>
<td>C++</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>access-declaration</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>access-specifier :</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>file-dependency-declaration</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>file-placement-declaration</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>filesystem-declaration</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>include-declaration</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>namespace-declaration</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syntax-macro-definition</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'most-other-things'</td>
<td>FOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.1 Statement and Declaration Grammar**

a. *access-declaration* is deprecated and therefore now appears as just *qualified-id* in the grammar.

b. *access-specifier :* is syntactically a *member-declaration* since it is interchangeable in its only usage which is as a sequence of *member-declarations* in a *member-specification*.

The first column identifies the nature of a contribution and the five subsequent columns show how that contribution applies to each context.
The contributions to simple-declaration and its member-declaration counterpart are further split into additional rows that reflect the alternate forms of initialization: scalar initializer (run-time evaluation), array initializer, constructor, scalar initializer (compile-time evaluation) and bit-field.

Boxes marked
- C++ identify a contribution in the standard C++ grammar
- FOG identify FOG extensions
- auto identify FOG extensions disambiguated by an auto keyword
- () identify degenerate grammar covered by another contribution

Thus a standard C++ member-declaration covers contributions from a function-definition, template-declaration, using-declaration, member-declaration (without initializer, with a constant-expression initializer, or with a bit-field), access-declaration and access-specifier.

FOG removes the syntactic distinction between declaration and member-declaration, and adds meta-functionality, resulting in the many further contributions shown by the boxes annotated as FOG.

The syntax of declarations is extended by the addition of meta-expression-statements and meta-control-declarations so that conditional and iterated compilation may embrace declarations. The existing control-statement syntax is re-used as is, with the result that declaration/expression ambiguities now occur at the declaration as well as statement level; extension of the existing disambiguation rule (§6.8) to favour declarations preserves compatibility.

gotos and associated statement labels are not re-used in meta-control-declarations, primarily because the label syntax has a challenging ambiguity with respect to an anonymous bit-field.

Since meta-control-declarations re-use control-statement syntax, they cannot be added directly to statement syntax. Boxes labelled auto therefore indicate where an auto prefix is necessary to disambiguate between a standard C++ construct and extended FOG meta-construct.

Meta-variables and meta-functions are added by overloading the auto keyword to mean "meta". Since the auto keyword has no meaning outside of functions this change merely gives meaning to constructs that are meaningless in C++.

In lexical form, the FOG extensions are:

```
statement:                        // Extension of
    auto control-statement
    auto meta-expression-statement

control-statement:               // New non-terminal with old functionality
    labeled-statement
    selection-statement
    iteration-statement
    jump-statement

declaration:                    // Extension of
    namespace-declaration     // Section 3.1.3.1
    accessibility-specifier   // Section 3.1.4.3
    compound-declaration      // Section 3.1.5.8
    meta-control-declaration  // Section 3.1.5.10
    auto meta-control-declaration
    expression-statement      // Section 3.1.5.10
    auto meta-expression-statement
    auto meta-class-declaration
    auto meta-function-definition
    auto meta-variable-definition
    syntax-macro-definition   // Section 4.7
    include-declaration       // Appendix F.4.6
    file-dependency-declaration // Appendix F.4.5
    file-placement-declaration // Appendix F.4.3
    filespace-declaration     // Appendix F.4.4
```
The syntactic difference between declaration and member-declaration is eliminated and consequently the member-declaration syntax must be added to init-declarator and an accessibility-specifier to declaration. The changes are syntactic only; C++ declarations that are not syntactically valid as member-declarations, and conversely, C++ member-declarations that are not syntactically valid as declarations, are semantically invalid in FOG. Accessibility and bit-fields may not be specified for namespaces and explicit-instantiations or using-directives may not be specified within classes.

`class-specifier: class-head { member-specification? declaration-seq? }`  
`accessibility-specifier: access-specifier:`  
`member-specification: member-declaration:`  
`member-declarator-list: member-declarator:`

### 3.1.5.1 Meta-names

The built-in types have meta-classes and the auto meta-class is the root of all meta-classes. We therefore define names that incorporate these alternative scopes, including their meta-constructors and meta-destructors.

`built-in-type-id: // e.g. unsigned int`  
`    built-in-type-specifier`  
`    built-in-type-id built-in-type-specifier`  
`meta-id: id meta-type auto`  
`meta-nested-id: meta-id ~ meta-id meta-id : : meta-nested-id`  
`meta-scope-id: // e.g. : : auto::symbol_table`  
`    : : opt meta-nested-id`

### 3.1.5.2 Meta-classes

Section 4.5 describes how every user-defined and built-in type has a meta-class with the same name. The meta-class is discarded after meta-compilation completes, so that the meta-class forms no part of the emitted code. The meta-class defines additional functionality for use at compile-time.

Meta-classes for classes that have no declarations may not be needed at compile-time and so such classes may not need emission. In order to forward reference such a meta-class, or to diagnose any inadvertent use of compile-time declarations, an auto may prefix a very similar syntax to a class-specifier. This asserts that the meta-class alone is required, avoiding an empty class declaration cluttering the generated output.

`meta-class-id: meta-id meta-id : : meta-class-id`  
`meta-class-key: class-key namespace`
meta-class-specifier:
  meta-class-key meta-class-id
  meta-class-key meta-class-id base-specifier-clauseopt { declaration-seqopt }

meta-class-declaration:
  meta-class-specifier;

declaration: // Extension of a u t i o m e t a - c l a s s - d e c l a r a t i o n
  access-specifiers are ignored for meta-declarations in a meta-class. All meta-declarations are therefore public. The presumption is that meta-functionality contributes to a tightly coupled pool of collaborating code, where access restrictions would be an inconvenience rather than an asset.

Impact

Re-use of reserved-words such as class as meta-type-names creates some inconvenient ambiguities with respect to meta-variable-declaration and meta-function-definitions. Most of these can be resolved by careful implementation to share common parsed prefixes. It is not obvious how to solve the problem of a global name as a meta-class-id, for which the leading :: in

auto class ::MyClass ( /* ... */ );

signifies the nested meta-class class: :MyClass of the class meta-type. Introducing parentheses:

auto class (::MyClass) { /* ... */ };

satisfies a generalised syntax for the meta-constructor of the class meta-type. However this cannot be unified because the body of a meta-constructor comprises statements whereas the body of a meta-class comprises declarations.

Therefore FOG does not support explicitly global scoping in meta-class-specifiers. An inelegant workaround is:

auto identifier globalScopeId = "";
auto class ${globalScopeId}: :MyClass { /* ... */ };

3.1.5.3 Base Meta-classes

Every user-defined and built-in type has a corresponding meta-class, whose meta-inheritance corresponds to the compile-time inheritance. The base meta-classes are the meta-classes of the base-classes. The meta-inheritance is augmented so that every meta-class without a base meta-class meta-inherits from the built-in root meta-class named auto. Additional meta-inheritance may be specified by using the auto keyword as an access-specifier in a base-specifier.

base-specifier:
  +opt nested-name-specifier opt class-name
  virtual access-specifier +opt nested-name-specifier opt class-name
  access-specifier virtual +opt nested-name-specifier opt class-name
  scoped-id
  built-in-type-id
  access-specifier base-specifier
  virtual base-specifier
  ! virtual base-specifier
  auto base-specifier

The rewritten recursion removes the syntactic limitations on multiple access-specifiers. It is therefore a semantic error for more than one of auto and the three distinct access-specifiers or for both virtual and !virtual to be supplied.

The virtual keyword is ignored for base meta-classes whose behaviour is always virtual; only one copy of a meta-class is inherited.
3.1.5.4 Meta-types

The arguments and returns of meta-functions and the values of meta-variables are defined by meta-types. All meta-types are built-in and there is no facility for user-defined meta-types. (User-defined classes are not meta-types, although extending meta-class definitions to support user-defined assignment might be a logical extension.) The defined set is

\[
\text{meta-type:}\ 
\begin{align*}
\text{meta-type-name} & \quad \text{built-in-type-id} \\
\text{enum} & \quad \text{typedef} \\
\text{typename} & \quad \text{using} \\
\end{align*}
\]

The direct mention of typedef, and indirect mention of class and signed reinstates names that duplicate reserved words and which were consequently tokenized as a reserved-word rather than a meta-type-name (Section 3.1.1.4). All the built-in C++ numeric types are available for use as meta-types for compile-time calculations. (The current FOG implementation maps the 20 distinct C++ types to one of: bool, unsigned, signed or double.)

\[
\text{meta-type-name:}\ 
\begin{align*}
\text{intrinsic-meta-type-name} & \quad \text{actual-meta-type-name} \\
\text{potential-meta-type-name} & \\
\end{align*}
\]

The intrinsic meta-types define concepts that do not depend upon their program context.

\[
\text{intrinsic-meta-type-name:} \quad \text{one of} \ 
\begin{align*}
\text{array_modifier} & \quad \text{assignment_expression} \\
\text{character} & \quad \text{class_key} \\
\text{constant_expression} & \quad \text{cv_qualifier} \\
\text{declSpecifier} & \quad \text{declaration} \\
\text{expression} & \quad \text{function_modifier} \\
\text{handler} & \quad \text{identifier} \\
\text{initializer_clause} & \quad \text{iterator} \\
\text{keyword} & \quad \text{meta_type} \\
\text{modifier} & \quad \text{name} \\
\text{number} & \quad \text{pointer_modifier} \\
\text{punctuation} & \quad \text{reference_modifier} \\
\text{reserved} & \quad \text{scoped_modifier} \\
\text{statement} & \quad \text{string} \\
\text{template_argument} & \quad \text{token} \\
\text{tree_literal} & \quad \text{using_directive} \\
\end{align*}
\]

Potential meta-types define concepts that have limited meaning until associated with some parent context.

\[
\text{potential-meta-type-name:} \quad \text{one of} \ 
\begin{align*}
\text{baseSpecifier} & \quad \text{built_in_type_specifier} \\
\text{classSpecifier} & \quad \text{elaborated_type_specifier} \\
\text{enumSpecifier} & \quad \text{file_dependency_specifier} \\
\text{file_placement_specifier} & \quad \text{enumerator_definition} \\
\text{exception_specification} & \quad \text{filesystem_specifier} \\
\text{function_specifier} & \quad \text{linkage_specification} \\
\text{meta_class_specifier} & \quad \text{meta_function_specifier} \\
\text{meta_parameter_specifier} & \quad \text{meta_variable_specifier} \\
\text{namespace_definition} & \quad \text{namespace_alias_definition} \\
\text{object_specifier} & \quad \text{object_statement} \\
\text{parameter_specifier} & \quad \text{scopeSpecifier} \\
\text{specifier} & \quad \text{template_parameter_specifier} \\
\text{templated_parameter_specifier} & \quad \text{type_parameter_specifier} \\
\text{type_specifier} & \quad \text{typedef_specifier} \\
\text{using_declaration} & \quad \text{value_parameter_specifier} \\
\text{variable_specifier} & \\
\end{align*}
\]

Actual meta-types define concepts that have been associated with a parent context, and in many cases correspond to emitted declarations.
### Meta-variables

Meta-variables support storage of values at compile-time. Their definition differs from conventional variables through the use of an `auto` prefix, the requirement for an initializer, and the acceptance of almost any syntactically valid construct as that initializer.

```plaintext
meta-variable-declaration:
  staticopt constopt meta-type meta-scoped-id exposed-treeopt = tree-statement
  staticopt constopt meta-type meta-scoped-id exposed-treeopt object-statements-clause
  staticopt constopt meta-type (meta-scoped-id) exposed-treeopt = tree-statement
  staticopt constopt meta-type (meta-scoped-id) exposed-treeopt object-statements-clause
```

**exposed-tree:**

```plaintext
[]
```

**declaration:**

// Extension of `auto meta-variable-declaration`

When `[ ]`'s are omitted, the syntax defines a scalar meta-variable with a single initializer. The single initializer may be a tree of initializers with `compound-tree-statements` used as `tree-statements` to create the tree structure. All leaves in the initializer tree must satisfy the syntax of the meta-type. When subsequently assigned to an iterator, the iteration domain comprises the one root element.

When `[ ]`'s are present, the syntax similarly defines a meta-variable but from a compound initializer. When subsequently assigned to an iterator, the iteration domain comprises the first generation of children, thereby treating the tree as an array.

The semantics of the composed list and tree types are discussed in Section 4.1.5.

---

5. Some meta-type names are also reserved words. The usage as a meta-type name augments usage as a reserved word.
static meta-variables have a single value shared by all derived meta-classes.

!static meta-variables have distinct copies for each derived class.

const meta-variables may have only a single unchanging value. Assignment or re declaration is illegal.

non-const meta-variables may change value either by assignment or by composition with a further declaration.

Meta-variables are non-const and !static by default.

Impact

The meta-expression-statement and meta-variable-declaration syntaxes exhibit expression/declaration ambiguities. The existing disambiguation rule is extended to resolve an ambiguity in favour of the meta-declaration.

Members of the meta-classes of the built-in types and members of globally scoped meta-classes can be difficult to specify:

```cpp
auto int int::a = 0;
auto int ::MyClass::a = 0;
```

In the first case, the disambiguation rule for built-in types maximises the length of built-in type specifiers, treats `int int` as a single type and so the example is an assignment expression.

In the second case, extrapolation from the §7.1-2 disambiguation rule, that maximises the length of a type-specifier-seq, maximises the length of the alternating names and scopes and so once again the example is an assignment to the semantically illegal nested class of `int`.

Parentheses must be used to define valid declarations.

```cpp
auto int (int::a) = 0;
auto int (::MyClass::a) = 0;
```

Each of these examples is now syntactically valid as both a meta-expression and a meta-declaration, so the disambiguation rule resolves in favour of the required meta-declaration.

There is no ambiguity with respect to meta-constructor declarations, since there are no pure virtual meta-functions and meta-constructors have no parameters.

### 3.1.5.6 Meta-functions

Meta-functions provide code for execution during the meta-compilation process.

```cpp
declaration: // Extension of
  auto meta-function-definition

meta-function-definition: // Part of
  staticopt meta-type meta-scoped-id { meta-parameter-listopt } exposed-treeopt
  compound-tree-statement
  object-statements-clause

meta-parameter-list:
  meta-parameter
  meta-parameter-list , meta-parameter

meta-parameter:
  meta-type identifier exposed-treeopt
  meta-type identifier exposed-treeopt = tree-argument
```

static meta-functions have a single copy shared by all derived classes.

!static meta-functions have distinct copies for each derived class.

The distinction between the two is relatively subtle given that a single instance always exists and so there is always a meta-object available. The non-static meta-function operates in the derived scope and so any use of `$Dynamic` returns
the derived scope, and any access to a meta-variable is made with respect to the derived scope.

The syntax is very similar to a conventional function declaration, except for the use of the much simpler and restrictive meta-type system. Default parameter values are supported, but not overloading or exception specifications.

The lack of support for overloading was once necessary because resolution of the meta-function was necessary to identify the syntax with which each argument was parsed. The evolution to a context free grammar, as described in Chapter 5, removes this constraint. Overloading could now be supported. It is just a matter of defining overload resolution rules that are in keeping with existing C++ overload resolution policies, but which are also appropriate for the dynamic rather than static type information available during meta-compilation.

An alternative form of definition and invocation is provided by the syntax-macro defined in Section 4.7.

### 3.1.5.7 Meta-constructor and Meta-destructor

Meta-constructors and meta-destructors provide for relatively independent meta-programs. They are invoked automatically during the meta-construction and meta-destruction compilation phases. They therefore have no parameters.

```plaintext
meta-nested-constructor-id:
  meta-id
  meta-id : : meta-nested-constructor-id

meta-scoped-constructor-id:
  : opt meta-nested-constructor-id

meta-nested-destructor-id:
  meta-id
  meta-id : : meta-nested-destructor-id

meta-scoped-destructor-id:
  : opt meta-nested-destructor-id

meta-function-definition:  // Part of meta-scoped-constructor-id
  meta-scoped-constructor-id ( ) compound-tree-statement
  meta-scoped-constructor-id ( ) object-statements-clause

meta-scoped-constructor-id ( ) compound-tree-statement

meta-scoped-destructor-id ( ) compound-tree-statement

meta-scoped-destructor-id ( ) object-statements-clause
```

### 3.1.5.8 Meta-control-statements and meta-control-declarations

Meta-statements control compilation. The control part of a meta-control-statement or meta-control-declaration is evaluated, as the source text is analysed or a meta-program executed, to affect the interpretation of the child statement or declaration.

Within a statement-seq, the conventional program control statements retain their runtime meaning. Additional meta-programming control applies when an auto prefix is used.

```plaintext
statement:  // Extension of control-statement
  control-statement
  auto control-statement

control-statement:
  labeled-statement
  selection-statement
  iteration-statement
  jump-statement
```

Within a declaration-seq, C++ provides no program control and so the existing statement syntax is re-used with an optional auto prefix to define a meta-control-declaration.

```plaintext
declaration:  // Extension of meta-control-declaration
  meta-control-declaration
  auto meta-control-declaration
```
meta-control-declaration:
- case constant-expression : declaration
- default : declaration
- do declaration while ( expression ) ;
- for ( for-init-statement condition_opt ; expression_opt ) declaration
- if ( condition ) declaration
- if ( condition ) declaration else declaration
- switch ( expression ) declaration
- while ( condition ) declaration
- jump-statement

The meta-control-declaration syntax repeats nearly all the program control syntax of statement replacing child statements by child declarations. The goto form of labeled-statement is omitted to avoid introduction of a syntax ambiguity with respect to an anonymous bit-field. goto is therefore not supported for meta-programming.

A return meta-statement has no meaning, since the semantics of meta-function execution involve a return of the entire meta-function body as a tree for interpretation in the calling context (Section 4.3).

Impact

Interspersing compile-time and run-time control apparently introduces more ambiguities:

```cpp
do {
    if (...) {
        auto do {
            auto if (...) {}
        } while (...);
    } else (...) 
} while (...);
```

How do the ifs, elses, dos and whiles pair up?

The meta-syntax is integrated with the language, and statements nest. The pairing is therefore exactly the same as would be the case with the auto keywords removed and all meta-statements changed to statements. The dangling else ambiguity is resolved as always to the nearest if, or auto if.

It is unnecessary and highly undesirable to introduce an auto prefix for else or the while of do...while. This would permit the meta-programming control flow to interleave programming control flow. Although this is permitted by the C preprocessor, it leads to difficult to understand code. The following example demonstrates how preprocessor directives can be used in a very unstructured fashion.

```cpp
if {
    a();
} #if B
else {
    #endif
    b();
    #if !B
    else {
        #endif
        c();
    }
```
The distinct preprocessor syntax makes the example comprehensible. If this were permitted in FOG, the use of almost identical syntax for meta-statements and statements would render interleaved code unintelligible.

There is also a pragmatic reason to avoid prefixing `else` with `auto`; 2 tokens of lookahead are required to resolve a dangling `auto else` ambiguity.

The `declarations` in selection and iteration meta-statements have a data-dependent interpretation. These productions must therefore be parsed with the appropriate number of side-effects. Section 4.2.1 describes how this may be achieved either by using a simple lookahead parser to cache the unparsed tokens for deferred parsing under control of the parent statement. Alternatively, AST nodes may be constructed directly, but without any resolution of tree-literals. Resolution of tree-literals is initiated by the parent statement. The earlier multi-pass FOG implementation was context-dependent and took the former approach. The superset grammar is context-free and takes the latter.

3.1.5.9 Meta-typedef

A typedef defines a more convenient name for a run-time type during compilation. That name has no meaning at run-time.

A meta-typedef could therefore define a more convenient name for a compile-time type during (meta-)compilation. That name also has no meaning at run-time.

meta-typedefs are not currently implemented in FOG. There is no obvious reason why they should not be. Arguably, the typedefs in HandySignatures on page 54 are meta-typedefs.

3.1.5.10 Meta-expression-statement

A `meta-expression-statement` takes two forms, each of which must be preceded by an `auto` prefix to distinguish from the very similar syntax of an `expression-statement` in contexts where an ambiguity could arise: function bodies and tree-literals. In other contexts (declarations) an `auto` prefix is optional.

The first form performs some operation as a side effect:

```cpp
auto a++;
auto f();
```

The second form updates a meta-variable

```cpp
auto a = 5;
auto listVariable += { "a", a * b, g[] };
```

```cpp
meta-primary-expression:
  literal
  this
  meta-scoped-id
  meta-type meta-nested-id
  - ( tree-argument-listopt )

meta-postfix-expression:
  meta-primary-expression
  meta-postfix-expression ( tree-argument-listopt )
  meta-postfix-expression [ expressionopt ]
  meta-postfix-expression . declarator-id
  meta-postfix-expression -> declarator-id
  meta-postfix-expression ++
  meta-postfix-expression --
```
meta-unary-expression:
   meta-postfix-expression
   ++ meta-unary-expression
   -- meta-unary-expression
   * meta-unary-expression
   + meta-unary-expression
   - meta-unary-expression
   / meta-unary-expression
   \ meta-unary-expression
   sizeof unary-expression

meta-multiplicative-expression:
   meta-unary-expression
   meta-multiplicative-expression * meta-unary-expression
   meta-multiplicative-expression / meta-unary-expression
   meta-multiplicative-expression % meta-unary-expression

meta-additive-expression:
   meta-multiplicative-expression
   meta-additive-expression + meta-multiplicative-expression
   meta-additive-expression - meta-multiplicative-expression

meta-shift-expression:
   meta-additive-expression
   meta-shift-expression << meta-additive-expression
   meta-shift-expression >> meta-additive-expression

meta-relational-expression:
   meta-shift-expression
   meta-relational-expression < meta-shift-expression
   meta-relational-expression > meta-shift-expression
   meta-relational-expression <= meta-shift-expression
   meta-relational-expression >= meta-shift-expression

meta-equality-expression:
   meta-relational-expression
   meta-equality-expression == meta-relational-expression
   meta-equality-expression != meta-relational-expression

meta-and-expression:
   meta-equality-expression
   meta-and-expression & meta-equality-expression

meta-exclusive-or-expression:
   meta-and-expression
   meta-exclusive-or-expression ^ meta-and-expression

meta-inclusive-or-expression:
   meta-exclusive-or-expression
   meta-inclusive-or-expression | meta-exclusive-or-expression

meta-logical-and-expression:
   meta-inclusive-or-expression
   meta-logical-and-expression && meta-inclusive-or-expression

meta-logical-or-expression:
   meta-logical-and-expression
   meta-logical-or-expression || meta-logical-and-expression

meta-conditional-expression:
   meta-logical-or-expression
   meta-conditional-expression ? meta-conditional-expression : meta-conditional-expression

meta-expression-statement:
   meta-conditional-expression
   meta-logical-or-expression assignment-operator tree-statement

The expression syntax is repeated so that meta-primary-expression resolves meta-scoped-id in the meta-name-space, whereas primary-expression resolves declarator-id in the conventional name-space. The semantics of meta-assignment differ and some inappropriate operators are omitted. A conventional assignment is right associative. Therefore:

\[ a = b = c = d; \]

is equivalent to:

\[ a = (b = (c = d)); \]
A meta-assignment does not associate. The entire right-hand side is analyzed as a tree-statement and is assigned to the left-hand side as a literal syntactic element:

```cpp
class X
{
    void f()
    {
        int x;
        int y;
        y = x;
        auto class_specifier x = class X {};
        auto class_specifier y = class Y {};
        auto y = class X {};
        auto y = $x;
        auto y = x;
        auto y = class X {};
        y = $x;
        $y z;
        $y x;
        $y = x;
        $y = $x;
    }
}
```

assigns b = c = d to a. Of these names, only a is resolved in the meta-name-space. b, c and d are not resolved in any namespace, since they form part of a literal. Eventually when the value of a is used, the name-space in which to resolve b, c and d may become clear.

```cpp
$a; // b = c = d;
// b, c, d in normal name-space.
auto $a; // auto b = c = d;
// b resolved in meta-name-space
// c and d still literals
```

The generality required to parse the tree-statement necessitates a simplification of meta-conditional-expression to exclude assignments and comma-separated lists of meta-expressions.

The meta-expression grammar has the following further incidental differences:

- no casts
- no typeid
- no new
- no delete
- no unary &
- no . * or ->*

Removing explicit casts simplifies the implementation. The remaining constructs are inappropriate for the current language definition.

Impact

The meta-expression-statement and meta-variable-declaration syntaxes exhibit expression/declaration ambiguities. The existing disambiguation rule (§6.8) is extended to resolve an ambiguity in favour of the meta-declaration.

The tree-statement is initially analyzed without knowledge of the expected syntactic type. When assigned to a meta-variable, used as a meta-parameter or returned from a meta-function, a further analysis checks that the value is compatible with the required meta-type. The program is ill-formed if it is incompatible.

A meta-expression-statement replaces and must therefore cover the syntax for a local function variable with an auto prefix. This requires the practical implementation of...
the above grammar to add unary & to support local reference variables, and accept
cv-qualifiers following * to support local pointer variables.

3.2 Built-In Functionality

The built-in functionality is described at length in Appendix E, and so only a very
brief summary is provided here.

3.2.1 Built-in Root Meta-class

auto is the root meta-class for all other meta-classes (including those for
namespaces and built-in types). It has no functionality, but user-defined
functionality can be added and thereby affect all classes.

3.2.2 Built-in Meta-variables

Meta-type-specific meta-variables support access to declarations.

3.2.3 Built-in Meta-functions

Meta-type-specific meta-functions support access to declarations.

3.2.4 std meta-namespace

The std meta-namespace is used as a repository for useful language support
meta-functions.

    static bool std::ambiguous(expression aName)
    static bool std::defined(expression aName)
    static token std::find(expression aName) []
        interrogate the meta-name-space to determine whether a reference to aName is
        ambiguous, defined, or to return a list of all definitions.

    static void std::diagnostic(string aString)
    static void std::error(string aString)
    static void std::warning(string aString)
        generate diagnostic, error and warning messages.

    static string std::get_cpp(string aString)
        resolves a definition within the preprocessor/command line name-space.

    static string std::get_env(string aString)
        resolves a definition within the external environment.

    static string std::date()
    static string std::file()
    static string std::time()
        replace the ANSI C __DATE__, __FILE__ and __TIME__ macros.

    static token std::parse(string aString)
    static token std::parse_tokens(token someTokens[])
    static token std::tokenize(string aString) []
        support character-level substitution and re-entrant analysis.

3.3 Incompatibilities

In principle the FOG grammar is a superset of the C++ grammar. In practice there
are some very minor incompatibilities.
3.3.1 Semantic Errors

Most of the significant FOG enhancements occur by defining a meaning for constructs that are semantic (or syntactic) errors in C++. Therefore a C++ program with semantic errors may be error-free in FOG. This is an inevitable consequence of enhanced semantics.

3.3.2 Transparency

FOG generates multiple output files with a default disposition to an interface and an implementation file per top level class. If FOG is used to preprocess C++ code requiring a more sophisticated file allocation, the file structuring may be lost. file-placement-declarations may be needed to create the required structure, see Appendix F.4.3.

This is an inconvenience that occurs when porting C++ code to FOG. C++ code used via #include or using/utility is not regenerated and so retains its file structure.

3.3.3 auto

FOG meta-constructs are defined by overloading the auto keyword. This causes no incompatibility outside functions, where the keyword is semantically invalid. The keyword is required to disambiguate syntax within a function.

```c++
void f()
{
    int i = 5;     // Assignment of 5 to (integer) i
    auto int j = 5; // As above with redundant auto
    auto e = k = 5; // Assignment of k = 5 to (expression) e
}
```

3.3.4 Incompatible concatenation

Extension of the ANSI C string concatenation policy to characters, identifiers, numbers and strings potentially introduces a major incompatibility for FOG. However the definition of the major syntax elements of C++ (whitespace, punctuation, characters, numbers, strings, identifiers and keywords) and accidental properties of the C++ grammar limit the problems.

An incompatibility arises wherever FOG concatenates but C++ does not. This may occur when characters, identifiers (including keywords), numbers and strings occur without intervening whitespace or punctuation. There are 16 combinations of adjacent characters, identifiers, numbers and strings to consider:

Character-anything only occurs in C++ when a character-literal arises as a literal in a primary-expression and the subsequent operator is not punctuation. In ARM C++ [Ellis90] there were no non-punctuation operators, however the standard [C++98] introduced the alternative tokens (and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq).

Anything-character occurs in C++ when a character-literal arises as a literal in a primary-expression and is preceded by non-punctuation. This occurs for the alternative tokens and following a return, sizeof or throw keyword.

The same possibilities occur for number-anything, anything-number, string-anything and anything-string.

String-string concatenates in FOG, and corresponds to ANSI C behaviour.

"This " "is" ""a" " single" string" // "This is a single string"
Identifier-identifier cannot occur since whitespace or punctuation is required to terminate the first identifier.
For identifier-string and string-identifier there is a further problem that arises in the linkage-specification syntax.

extern "C" size_t f;

must be written in FOG as

extern "C" size_t f;

This is unlikely to cause many problems since few programmers would choose to be so economical with whitespace in these cases.

The FOG concatenation extension therefore introduces minor incompatibilities. Whitespace cannot be omitted around the string-literal of a linkage-specification, following return, sizeof or throw or around alternative tokens, if the omission conflicts with a concatenation interpretation.

3.4 Cpp Replacement

Stroustrup, in The Design and Evolution of C++ [Stroustrup97], identifies elimination of the preprocessor as a major goal for C++, devoting the final chapter to a discussion of its weaknesses, and identifying some remedies that C++ provides. In the final paragraph, Stroustrup writes

"I'd like to see Cpp abolished. However the only realistic and responsible way of doing that is first to make it redundant ...".

In order to justify the claim that FOG makes Cpp redundant, we must briefly review the deficiencies and facilities of Cpp to determine to what extent FOG resolves and replaces them.

3.4.1 Cpp limitations

Cpp supports the definition of and replacement of object-like and function-like macros.

An object-like macro associates an identifier with a replacement sequence of preprocessor tokens.

#define OCTAL_CASES \ 
   '0': case '1': case '2': case '3': case \ 
   '4': case '5': case '6': case '7'

The replacement tokens replace the macro identifier wherever it occurs.

switch (c)
{
   case OCTAL_CASES: /* ... */ break;
   case ALPHABETIC_CASES: /* ... */ break;
   case '%' : /* ... */ break;
}

Substitution occurs at a very low level, offering the programmer considerable flexibility. In the above example, the replacement sequence omits an initial case keyword and a trailing colon token. The missing tokens accompany the instantiation. Readers may form their own opinion as to whether the unusual definition leads to a dangerously obscure or aesthetically pleasing implementation of the switch statement.

A function-like macro associates an identifier and a list of formal parameters with a replacement sequence.

#define MAX(a,b) ((a) > (b) ? (a) : (b))

Invocation of the macro provides the actual arguments that replace the formal parameters in the replacement sequence.
Macro substitution is simple but prone to accidents. Substitution occurs at a lexical level and so ignores any logical structure that may be present in the source code.

The intention that \texttt{MAX} returns an expression from a pair of expression arguments is only realised when the actual usage is appropriate.

The apparently redundant parentheses in the macro definition avoid the surprising evaluation that would otherwise result from the interpretation of

\[
\text{price + MAX(current\_rate, fixed\_rate) * commission}
\]
as

\[
(price + current\_rate) > fixed\_rate) \quad ? \quad current\_rate \quad : \quad (fixed\_rate \quad * \quad commission)
\]

### 3.4.1.1 Unwanted substitution

Perhaps the most serious problem with the preprocessor is that of name capture. All names occur in a single namespace and so every conventional use of a name that is defined as a macro can malfunction. For instance the enumeration

\[
\text{struct Options}
\{
\quad \text{enum \{ LEFT, RIGHT, UP, DOWN, MAX \};}
\}
\]

should operate quite satisfactorily with \texttt{Options::MAX} denoting the number of options. This definition will typically be placed in some include file. However if another include file contains the earlier definition of \texttt{MAX}, any reference to \texttt{Options::MAX} will fail whenever both include files are used. If the macro is defined before the enumeration, a syntax error will spring up in the enumeration.

This form of error is obscure and confusing. It can appear to be intermittent since compilations that do not use both include files succeed. Novice programmers are baffled. Experienced programmers may take a little time to detect the handiwork of the preprocessor.

This problem is resolved in FOG by changing to a policy of invited substitution. The replacement functionality for \texttt{MAX} substitutes only as part of a $ or $ expression.

### 3.4.1.2 Language independence

The independence of macros from the underlying language is resolved in FOG by use of meta-types to constrain the syntax of meta-variable values and of meta-function argument and return values. Parentheses are not required.

When a meta-function is invoked, the argument is represented by a parsed AST and eventually validated against the required syntax. The tree is substituted for each reference within the meta-function, and since the tree has already been parsed there is no possibility of re-interpretation in conjunction with surrounding tokens.

### 3.4.1.3 Side-effects

When a macro such as \texttt{MAX} is invoked with an argument that causes a side effect

\[
c = \text{MAX}(a++, ++b)
\]
one of the arguments is evaluated twice, and so receives a double increment. If the first argument is greater, the result is obtained after one increment has occurred. It is therefore unlikely that the program will function as required.

FOG does not resolve this problem, since FOG passes the syntax tree for \texttt{a++} into the meta-function and instantiates the tree for each reference. FOG does not
'evaluate' the argument once and then pass it. This is not possible because the argument is a syntax tree, not a value. It may be that the syntax happens to correspond to a value as in the \texttt{MAX} case, but not in general.

\texttt{MAX} is a poor example for a syntactic meta-function. \texttt{MAX} is a computational operation and so can and should be implemented by some form of inlined function.

\subsection*{3.4.1.4 Substitution level}

Many implementations of the original Kernighan and Ritchie C preprocessor \cite{Kernighan78} performed character-based substitution. The replacement characters were inserted between the surrounding characters and the resulting character stream was then re-analysed. This offered considerable flexibility, but different implementations varied in their treatment of obscure recursions and encountered difficulties when a composite token such as \texttt{+=} arose as a result of character concatenation.

The ANSI C preprocessor changed to token-based substitution. The preprocessor identifies the tokens and substitution replaces a sequence of tokens. Composite tokens can only arise through explicit use of the \#\# operator.

Although the FOG substitution is syntax-based, character-based substitution is also supported through use of the \texttt{std::parse} built-in meta-function and the \texttt{token} meta-type\textsuperscript{6}. \texttt{token} is the most primitive and generic terminal of the FOG grammar. Every number, string, identifier or piece of punctuation such as \texttt{>>=} is a \texttt{token}.

\begin{verbatim}
auto string OCTAL_CASES[] =
    "'0': case '1': case '2': case '3': case "
    "'4': case '5': case '6': case '7': ";
\end{verbatim}

The required tokens are represented as a concatenation of two strings to fit the available line length. This is then incorporated into its overall context by further string concatenation before \texttt{std::parse} is invoked to convert to a syntax tree.

\begin{verbatim}
$std::parse(
    "switch (c)"
    "{"
    " case "$OCTAL_CASES": /* ... */ break;"
    " case "$ALPHABETIC_CASES": /* ... */ break;"
    " case '%': /* ... */ break;"
    "})"
);
\end{verbatim}

A cleaner solution is available by meta-programming:

\begin{verbatim}
auto statement multi_case(signed lo, signed hi)
{
    auto for (signed i = $lo; i <= hi; ++i)
        case $i: ;
}

switch (i)
{
    $multi_case('0', '7') /*...*/ break;
    $multi_case('a', 'z') /*...*/ break;
    case '%': /* ... */ break;
}
\end{verbatim}

\textsuperscript{6.} Token-based substitution could easily be supported, but it is difficult to conceive of an example that is not better resolved by syntax-based substitution, or more clearly resolved by character-based substitution.
This exploits the non-structured definition of a case statement to treat case 'a':; as a statement that drops through to the next case. The extra semicolon avoids the need for a case meta-type.

```
case '0': ;
......
case '?' : ;
/* ... */ break;
```

### 3.4.1.5 Backslash continuations

The C preprocessor requires all directive lines to comprise exactly one complete line. This restriction is alleviated by the ability to continue a line with a trailing backslash, but results in less readable and difficult to maintain code.

FOG integrates replacement functionality into the main grammar and so the replacement facilities are free format.

### 3.4.2 Concatenation and Stringizing

The character-based substitution of the K&R C preprocessor enabled composite tokens (normally an extended identifier) to be formed by causing the character sequences to abut.

```
begin/**/_and__/**/end // begin_and_end
```

An identifier could be converted to a string by substitution within a macro:

```
#define STRINGIZE(s) "s"
STRINGIZE(text) // "text"
```

In the ANSI C preprocessor, the change to token-based substitution, and the requirement that a comment be replaced by a whitespace character, lost this flexibility necessitating the introduction of the ## operator to request concatenation, and the # operator to support stringizing.

```
#define CONC3(a,b,c) a ## b ## c
CONC3(begin,_and_,end) // begin_and_end
#define STRINGIZE(s) #s
STRINGIZE(text) // "text"
```

In FOG character-based concatenation occurs between characters, numbers, strings, identifiers (including keywords) without intervening whitespace. An empty string or character can be used to provide separation between elements that require concatenation.

```
"begin_and_end"
```

An empty string (or character) can be used as a meta-cast for stringizing. The subsequent text acquires the string (or character) characteristics of the start of the sequence.

```
"text"
```

Alternatively, meta-functions can be defined with similar behaviour to the ANSI C approach

```
auto identifier CONC3(identifier a, identifier b, identifier c) {
    ${a}${b}${c}
}
auto string STRINGIZE(identifier s) {
    $s;
}
$CONC3(begin,_and_,end) // begin_and_end
$STRINGIZE(text) // "text"
```

$ invocations in the body of CONC3 access the formal parameters. The first two invocations use the${} form to ensure evaluation as ${a}${b}${c}, since the default of taking the longest possible interpretation of $a$\$b$\$c$ would evaluate ${a${b${c}}}$.
The stringize meta-function apparently does nothing, however the distinct parameter and return meta-types arrange for an identifier to string conversion, which also does nothing. The stringizing occurs at the point of usage; when a string representation is required, the internal representation is formatted as a valid string.

### 3.4.3 `#define` directive

Preprocessor object-like definitions

```c
#define PI 3.14159
```

and function-like definitions

```c
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

are replaced by meta-variable

`auto double PI = 3.14159;`

and meta-function

```c
auto expression MAX(expression a, expression b) {
    $a > $b ? $a : $b
}
```

definitions in FOG. The FOG definitions have syntactical types that can be checked and avoid the need for protective parentheses.

### 3.4.4 `#include` directive

The historical semantics of the `#include` directive permit an arbitrary sequence of tokens to be incorporated more than once in almost any context. This flexibility is excessive and almost never needed. Most programmers have learnt to tame the directive by placing an include file guard within each include file to inhibit multiple inclusion:

```c
#ifndef FRED_H_INCLUDED
#define FRED_H_INCLUDED
//...
#endif
```

although many have not learnt how much compile time can be saved by inhibiting the include as in

```c
#ifndef FRED_H_INCLUDED
#include "fred.h"
#endif
```

FOG supports a more disciplined form of inclusion like the `#import` of Objective C [Cox86]:

```c
using "fred.h";
```

performs the inclusion provided the file has not already been included. Guards are unnecessary.

### 3.4.5 `#if, #ifdef, #ifndef, #else, #elif, #endif` directives

Preprocessor conditional compilation

```c
#if defined(UNIX)
    static const char *temp_path = "/tmp/";
#else
    static const char *temp_path = "C:\\Temp\";
#endif
```

is replaced by the use of meta-statements
auto bool unix = std::get_cpp(UNIX);
auto if (unix)
    static const char *temp_path = "/tmp/";
else
    static const char *temp_path = "C:\Temp\";
with built-in meta-functions of the std meta-namespace providing support when necessary. (The auto preceding the if is unnecessary if the example occurs outside a function.)

3.4.6 #line directive

The #line directive is not used in source programs generated by human beings. It provides a very simple but useful mechanism for automatic source code generators to ensure that compilers and debuggers refer to the original source lines rather than some scrambled intermediate. #line performs this role adequately and needs no replacement. Once Cpp has been discontinued, an extension that uses a more cryptic free format spelling and releases the # token could be considered. This is examined in Appendix F.1.4.

3.4.7 #error directive

FOG provides the built-in meta-functions

auto void std::diagnostic(string someText)
auto void std::error(string someText)
auto void std::warning(string someText)

to support emission of meta-compile-time messages.

3.4.8 #pragma directive

FOG provides no explicit counterpart for pragmas. However the effects of pragmas may be achieved by vendor-defined built-in meta-functions.

A compilation system that supports intrinsic functions might recognise

#pragma intrinsic(memset)

to request use of the intrinsic rather than function call implementation of memset.

In a compatible meta-compilation system, a built-in meta-function to implement the pragma could be hypothetically declared:

auto void std::intrinsic(identifier anIdenifier);

and invoked as

$std::intrinsic(memset);

Invocation on a meta-compiler without support for the pragma could be dummied out by defining the meta-function to have no functionality:

auto void std::intrinsic(identifier anIdenifier) {}
or to redirect to a different pragma:

auto void std::intrinsic(identifier anIdenifier)
{} $std::use_intrinsics(""$anIdenifier); }

3.5 Summary

We have shown how C++ string concatenation can be usefully generalised by simple lexical changes to phase 6 of the C++ translation. This eliminates the need for Cpp concatenation.

We have shown how an additional form of literal representing a pre-parsed AST may be incorporated into the C++ grammar as an identifier to support the re-use of syntax trees. This eliminates the need for Cpp macros.
We have shown how the `auto` keyword can be re-used to make C++ control and declaration constructs available for meta-programming. This eliminates the need for Cpp conditionalisation.

We have introduced minor syntactic and semantic generalisations to provide greater consistency when multiple declarations are composed together in response to the relaxation of the One Definition Rule. This supports weaving of multiple code contributions together as required for Aspect Oriented Programming.

We have shown how these changes introduce minimal incompatibilities. The changes make Cpp redundant by providing replacement facilities that integrate with the main C++ language. In the next chapter we show how these changes facilitate meta-programming.
4 FOG Semantics

Chapter 3 presented the FOG grammar and discussed many of the minor semantic issues, but deferred the more major ones for resolution in this chapter.

We first describe the compilation stages that transform a FOG source program into C++ files, and distinguish between the potential declarations appearing in the FOG source and the actual declarations that are finally emitted. This leads on to a discussion of the meta-types and meta-objects used to represent these declarations, and the type-constructors used to support iterators and trees.

We then describe alternative models for substitution, justify the choice of syntax-level substitution in FOG, and examine the constraints imposed by supporting context free parsing of arbitrary syntax.

The distinction between the conventional name-space used for C++ declarations and the meta-name-space of meta-declarations is examined. Consideration of the context in which substitution occurs shows the need for more than one substitution operator, and a requirement to use lexically nested scopes rather than structurally nested scopes.

The flexibility that FOG provides for redeclarations is a critical distinction between FOG and C++ that is resolved by composing the multiple contributions to a single composite C++ declaration. The composition policies for different categories of declaration are described.

FOG has a simple meta-class model. Other languages have different models. We review the alternate models and show that the FOG meta-class model is a natural and consistent extension applying the run-time programming perspective to (meta-)compile-time.

User-defined code may contribute to decisions made at (meta-)compile-time through static meta-programming either through direct invocation from source code or through provision of meta-programs. Meta-programming by direct invocation is discussed in Sections 4.2, 4.3 and 4.4. Independent meta-programs are discussed in Section 4.6.

Finally the behaviour of syntax macros is described.

4.1 Meta-compilation stages

The significant processing stages of meta-compilation are shown in the left-hand column of Figure 4.1.

The right-hand column identifies the distinct representations used, with solid arrows indicating the flow of data between the processing stages. As will shortly be explained, the anomalous arrow direct to an actual declaration from a potential declaration is the result of automatic creation of an actual declaration from a potential declaration in a determined context.

Chapter 5 describes the successive stages of lexical, syntactic and semantic analysis that steadily increase the precision of the internal representation of the following trivial program

```
int x;
```

as

- six source characters for `int x`;
- three tokens for `identifier int`, `identifier x` and `punctuation`;
- a syntax tree for the generalised expression `int x`
- a potential declaration for the variable-specifier `int x`
- an actual declaration for the global variable `int ::x`
Actual declarations are emitted to the C++ output files, after construction of the usage dependency graph by the compilation stage as discussed in Chapter 6.

The distinction between potential declarations (in an unresolved context) and actual declarations (in a resolved context) is not necessary for C++, since every declaration has a well-defined context.

FOG supports passing of declarations to meta-functions, and the regeneration of declarations in derived contexts by derivation rules. Potential declarations therefore describe a declaration after semantic analysis. Transformation to an actual declaration occurs as soon as the declaration is associated with an actual context.

Static meta-programming may occur during
- semantic analysis
- meta-construction
- meta-main
- meta-destruction
- compilation

Meta-programming during semantic analysis occurs through interpretation of meta-statements:

```
auto if (generateDebugCode)
    /* debug code */
```

or more generally to service meta-function calls

```
$CachedString::flyweight("pointer");
```

The meta-construction, meta-main and meta-destruction stages are described in Section 4.6. They consist exclusively of application functionality.

The final opportunity for meta-programming occurs during the compilation stage. At this point meta-functions and meta-variables used within the derivation predicates and bodies of declarations are evaluated, and meta-programming must be restricted to the acquisition of information from the pool of declarations. New
declarations cannot be added, although previously declared declarations can be extended by composition and meta-variables updated.

4.1.1 Potential and Actual Declarations

In C++, each (non-template) class, function, type or variable declaration provided as source text is analysed in, and contributes once to, its surrounding context. Templates provide a macro facility that enables certain parametric references to be resolved automatically, but each template reference contributes at most once to its surrounding context.

In FOG, derivation rules and more generally meta-programming may cause one source declaration to make multiple contributions to a variety of contexts. Since there is no longer a one to one relationship between source text declarations and implemented declarations, we need to introduce new terminology.

A declaration appearing as source text is a potential declaration. A potential declaration has the potential to be resolved to an arbitrary number of actual declarations.

A declaration resolved to its eventual context is an actual declaration. Actual declarations correspond to the declarations emitted for subsequent compilation by the C++ compiler.

The two forms of declaration have significant differences. Consider the possible specifiers for an `enum`, which may occur in source text and consequently as a potential declaration for:

- a definition
  
  ```
  enum Enum { ENUMERATOR }
  ```

  corresponding to
  ```
  enum-specifier:
  enum identifieropt { enumerator-listopt }
  ```

- or a to be resolved reference
  ```
  enum Enum
  ```

  corresponding to
  ```
  elaborated-type-specifier: // Part of
  enum : :opt nested-name-specifieropt identifier
  ```

Neither of these can be resolved without knowledge of the prevailing context, whereas an actual declaration has already been resolved and known to be perhaps `enum : :MyNameSpace : :EnumScope : :Enum`.

There is no point in instantiating an actual declaration at the source level, since this can only create a duplicate of the resolved declaration. Conversely, it is meaningless to probe the unresolved context of a potential declaration.

There are therefore necessarily three distinct concepts to be represented by the meta-type system:

- a potential definition
- a (potential) unresolved reference
- an (actual) reference resolved to an actual definition

The distinctions between potential and actual declarations can be seen in the following example, which makes use of the built-in meta-types, `class_specifier`, `declaration`, and `identifier`. Meta-objects with these types store constructs that comply with the `class_specifier`, `declaration`, and `identifier` syntax. The more detailed semantics of meta-types are explained in the next section.

```
class Actual {};
```
auto declaration nest(identifier anId, class_specifier aClass)
{
    class $anId
    {
        $aClass;
    };
}  
auto declaration cached = $nest(Nested, class Potential {});
class ReNested
{
    $cached;
};
The first potential declaration, for class Actual, can be resolved immediately and so results in an actual declaration of class ::Actual.
The meta-function definition similarly results in the actual declaration of the meta-function ::nest. Analysis of the meta-function body creates a potential declaration for the class-specifier (class $anId) using the first formal parameter (anld), for which preliminary semantic analysis can verify compatibility of the identifier meta-type. Syntactic analysis of the tree-literal ($aClass) in the class-specifier encounters the usage of the second formal parameter. For reasons explained in Section 4.2.3, this is presumed to be an identifier which is a degenerate declaration. Again a preliminary semantic analysis can verify the meta-type compatibility of a class_specifier.
Analysis of the invocation of the nest meta-function creates potential declarations for the two arguments: Nested and class Potential {}. The first is an identifier, for which there is no distinction between potential and actual declarations. The second is a class-specifier whose context cannot be resolved, so the potential declaration cannot be converted to an actual. Further analysis verifies that the arguments can be converted to satisfy the identifier and class_specifier syntaxes of the formal parameters.
Interpretation of the meta-function replaces the formal parameters and returns the meta-function-body for execution in the calling context as if the source were:
    auto declaration cached = class Nested
    {
        class Potential {};
    };
Further potential declarations for class Nested and class Nested::Potential are created as the initializer for the potential and actual declarations of ::cached.
A potential and actual declaration for class ::ReNested are then created and the cached variable replaced as if the source were:
    class ReNested
    {
        class Nested
        {
            class Potential {};
        };
    };
so that potential and actual declarations are also created for class ReNested::Nested and class ReNested::Nested::Potential.

4.1.2 Meta-types
Execution of a meta-program involves manipulation of declarations that have some syntactical (meta-)type.

Page 86

29-June-2001
A very simple form of meta-execution could be supported by defining just a single meta-type such as `syntax_element`, which could exhibit polymorphic behaviour for all possible syntax elements. However source syntax checking is normally a rigorous activity and type-less meta-execution is not philosophically consistent with C++, which is (in principle) a strongly typed language.

FOG should therefore support strongly typed meta-execution. The obvious set of types correspond to the grammar productions: `class_name`, `expression`, `statement`, `template_parameter`, etc. The set of types provided by FOG will be described shortly.

Casting

C++ supports type-widening but requires some form of cast to narrow a type.

```cpp
class A {};
class B : public A {};
void f()
{
    B b;
    A *pa = &b;
    B *pb = &b;
    pa = pb; // Widening ok.
    pb = static_cast<B *>(pa); // Narrowing needs a cast.
}
```

This is necessary since it is not generally possible to determine at compile-time that a type-narrowing is safe. Static meta-programs execute at compile-time and so it is always possible to determine safety at compile-time. Precise replication of run-time behaviour at compile-time is therefore unnecessary and undesirable: casts are not available for meta-programming. Assignments are checked dynamically and if possible the assigned value is converted to a value compatible with the target meta-type.

```cpp
auto identifier i = label; // Ok label is an identifier
auto statement s = $i; // Ok identifier to expression-statement
auto expression e = $s; // Ok expression-statement to expression
```

No error occurs on the assignment of a statement to an expression, although not all statements are expressions, because the conversion is performed on the value which is an `expression-statement` for which there is a safe conversion. The permissible conversions are described in Section 4.1.6.

Meta-type as syntax predicate

It is appropriate to regard the meta-type as a syntax predicate, since the polymorphism available at compile-time can be exploited to convert the result of an assignment to the specified meta-type.

In order to keep the FOG grammar context-free, it is necessary to be able to parse the argument of a meta-function, or the initializer of a meta-variable without knowledge of the required syntax (or meta-type). A single syntax must cover all possibilities. This syntax is the `tree-statement` syntax described in Section 3.1.1.6. It satisfies the weakest predicate of the token meta-type, and also the stronger predicates corresponding to the meta-types used to create the AST.

Since each meta-type corresponds to a syntax predicate, a meta-type could be defined for each non-terminal in the FOG grammar.

There are three problems:

Excluding the lexical productions, there are about 150 non-terminals in the C++ grammar (more in FOG).
The non-terminals are not orthogonal. For instance after parsing the source text containing the isolated letter a, what is its syntactic type? It could be an identifier, template-argument, expression, abstract-declarator etc. Choosing the more primitive meta-type would be possible if the grammar was completely context-free and so the productions corresponded to nodes of a tree, unfortunately

\[ a < b > (c) \]

could be a postfix-expression or a declarator, for which postfix-expression is a shared common syntax type in the superset grammar implementation of FOG. However there is no shared syntax type in the C++ (or FOG) grammar. Distinct meta-types are needed to describe potential and actual declarations.

We therefore need extra meta-types to capture the potential/actual distinction, but want fewer of them to reduce the definition and documentation effort. There seems little point introducing meta-types to distinguish between a logical-and-expression and a logical-or-expression. A more pragmatic approach is therefore taken, defining meta-types only for those concepts for which a meta-program may reasonably need to make a distinction.

**Implemented meta-types**

Each meta-type corresponds to a non-terminal, whose production rules define the valid syntax for syntax trees with the meta-type. Many non-terminals are specialisation of others, thus, identifier is a (specialisation of) id-expression which is a primary-expression which is an expression which is a token. This specialisation hierarchy resembles an Object Oriented inheritance hierarchy.

The UML [Booch99] inheritance diagram in Figure 4.3 shows the externally visible general structure and some of the more significant meta-types. Figure 4.2 shows some of the intrinsic meta-types. Intrinsic meta-types define a syntax that can be directly identified and require no distinction between potential and actual declarations. A fuller listing of the intrinsic meta-types is provided in Table 4.1, with specialisation indicated by the partitioning into more specific meta-types across the "intrinsic meta-types" columns.

Figure 4.3 shows some potential and actual meta-types. Meta-objects of a potential meta-type more specialised than specifier are created by the (syntactic and) semantic analysis to describe potential declarations. These Meta-objects are converted to have an actual meta-type more specialised than entity once the context supports the conversion from a potential to an actual declaration. A fuller listing of the potential and actual meta-types is shown in Table 4.2 again using partitioning to denote specialisation.

**Declarations and Statements**

The syntax generalisations in FOG remove the principal distinctions between declarations and statements, however both concepts are widely used and so both declaration and statement meta-types should be provided. Implementation of
these types within a simple inheritance hierarchy requires one to be more specialised than the other. But within a function, a declaration-statement is a specialised statement, whereas in a class a meta-expression-statement is a specialised declaration.

The dilemma is resolved from the external perspective by treating the two names as synonyms for the same meta-type. A subtle, and almost irrelevant, distinction is made that a declaration resolves an ambiguous labeled-statement/anonymous-bit-field to the bit-field, whereas a statement resolves to the label.

**4.1.3 Meta-objects**

Potential and actual declarations and their constituent elements are represented by meta-objects within the meta-compiler, and many of these meta-objects are available for manipulation by meta-programs. Meta-objects are created by the presence of the corresponding declaration, and are managed automatically as part of a hierarchical pool of meta-objects maintained by the meta-compiler. There is no need or facility for explicit creation or deletion of meta-objects; there is no meta-operator new or meta-operator delete. Garbage collection of objects that are not required during the code emission phase occurs automatically.

Each meta-object has a corresponding meta-type, with distinct meta-types for potential and actual declarations. When

```cpp
enum Enum { El, E2Second }
```

is parsed to create a potential declaration, a meta-object of enumSpecifier meta-type is created with two child meta-objects: an identifier of identifier meta-type and a list of enumeratorDefinitions. The list in turn comprises two child objects of enumeratorDefinition meta-type, each of which has a child meta-object to define the name. The unresolved name of the second enumerator is represented by an expression tree to capture the deferred resolution. The root of the tree has the name meta-type. The meta-types (and some values) for the meta-objects defining the potential declaration are shown in Figure 4.4.

When the potential declaration is converted to an actual declaration, the unresolved name is evaluated in its actual context to yield a meta-object of identifier meta-type, and the enumSpecifier and enumeratorDefinition meta-objects are converted to the meta-objects of enum and enumerator meta-type shown in Figure 4.5.

Access to these automatically managed meta-objects occurs by establishing a reference. At the time of creation:
auto enum_specifier potentialDef = enum Enum { E1, E@Second }; $potentialDef;

by subsequent direct reference
auto enum_specifier potentialRef = enum Enum;
auto enum actualRef = enum Enum; // See Section 4.1.6.2

or by indirect reference
auto enumerator enum2a = $Enum::E2;
auto enumerator enum2b = $Enum.enumerators()[1];

4.1.4 Working meta-variables

Temporary meta-objects may be used for numeric calculations and iterators.

class X
{
    auto int i = 0;
    auto for (iterator p = $Enum.enumerators(); p; ++p)
        auto /* ... */ ++i /* ... */
};

The declaration of i has no syntactical indication that i is a temporary variable, so it isn't. The meta-variable X::i is therefore not temporary.

FOG establishes local scopes for meta-programming in the same way that C++ establishes local scopes for normal programming. The iterator is therefore a
### Table 4.1 Intrinsic Meta-types

<table>
<thead>
<tr>
<th>Intrinsic Meta-Type</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>punctuation</strong></td>
<td>example: <code>=</code>, <code>;</code>, <code>..</code>, <code>[]</code>, <code>*</code>, <code>&amp;</code>, <code>.</code></td>
</tr>
<tr>
<td><strong>modifier</strong></td>
<td><code>array_modifier</code> = <code>.. [ ]</code></td>
</tr>
<tr>
<td></td>
<td><code>function_modifier</code> = <code>(int x) throw y</code></td>
</tr>
<tr>
<td></td>
<td><code>pointer_modifier</code> = <code>*T::Q const ..</code></td>
</tr>
<tr>
<td></td>
<td><code>reference_modifier</code> = <code>a ..</code></td>
</tr>
<tr>
<td><strong>reserved</strong></td>
<td><code>case</code> = <code>reserved-word</code></td>
</tr>
<tr>
<td><strong>meta_type</strong></td>
<td><code>function</code> = <code>meta-type</code></td>
</tr>
<tr>
<td></td>
<td><code>class</code> = <code>class-key</code></td>
</tr>
<tr>
<td></td>
<td><code>static</code> = <code>(non-type) decl-specifier</code></td>
</tr>
<tr>
<td><strong>decl_specifier</strong></td>
<td><code>cv_qualifier</code> = <code>volatile</code></td>
</tr>
<tr>
<td></td>
<td><code>identifier</code> = <code>identifier</code></td>
</tr>
<tr>
<td><strong>expression</strong></td>
<td><code>a = b ^ c</code> = <code>assignment-expression</code></td>
</tr>
<tr>
<td></td>
<td><code>5</code> = <code>constant-expression</code></td>
</tr>
<tr>
<td></td>
<td><code>'a'</code> = <code>character-literal</code></td>
</tr>
<tr>
<td></td>
<td><code>4.0</code> = <code>number-literal</code></td>
</tr>
<tr>
<td></td>
<td><code>&quot;string&quot;</code> = <code>string-literal</code></td>
</tr>
<tr>
<td></td>
<td><code>a, b</code> = <code>expression</code></td>
</tr>
<tr>
<td></td>
<td><code>throw (a)</code> = <code>throw-expression</code></td>
</tr>
<tr>
<td></td>
<td><code>int x(5)</code> = <code>mem-initializer</code></td>
</tr>
<tr>
<td></td>
<td><code>label: x;</code> = <code>labeled-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>if (a) ;</code> = <code>selection-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>while (1) ;</code> = <code>iteration-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>goto label;</code> = <code>jump-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>try {} catch ... {}</code> = <code>try-block</code></td>
</tr>
<tr>
<td><strong>statement</strong></td>
<td><code>a = b;</code> = <code>expression-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>int a;</code> = <code>declaration-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>template T&lt;Q&gt; a;</code> = <code>explicit-instantiation</code></td>
</tr>
<tr>
<td></td>
<td><code>asm &quot;;&quot;;</code> = <code>explicit-specialization</code></td>
</tr>
<tr>
<td></td>
<td><code>using namespace X;</code> = <code>asm-definition</code></td>
</tr>
<tr>
<td></td>
<td><code>()</code> = <code>using-directive</code></td>
</tr>
<tr>
<td></td>
<td><code>() [] {}</code> = <code>compound-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>compound-declaration</code> = <code>function-body</code></td>
</tr>
<tr>
<td><strong>handler</strong></td>
<td><code>catch (a) ;</code> = <code>handler</code></td>
</tr>
<tr>
<td><strong>exception_specification</strong></td>
<td><code>throw (a)</code> = <code>exception-specification</code></td>
</tr>
</tbody>
</table>

 temporary meta-variable since it is declared within a for loop. However, so are any other meta-variables declared within the loop. This problem may be circumvented by prefixing the declaration with `This` to start name resolution beyond the local scopes.

```cpp
auto for (iterator p = $Enum.enumerators(); p; ++p)
    auto if (p->value() > 255)
        auto number This.has_big_enum = true;
```

#### 4.1.5 Scalars, Arrays, Lists and Trees

The elemental meta-types and their use for compile-time calculation and syntax validation has been described. Use of scalar types alone is inadequate for meta-programming, since many declarations involve lists of child declarations, and in some cases trees of descendant declarations. Traversal of these structures must be supported.
### Table 4.2 Potential and Actual Meta-types

C++ supports constructed types through pointer-to, array-of, function-returning and record-of type constructors, and perhaps the same type constructors should be available for meta-types.

The C++ pointer system inherited from C is necessary to support uncontrolled memory access and arbitrary memory allocation through the type-less malloc(). This lack of discipline causes many problems through the use of null, dangling or stale pointers and allows memory to leak. C++ introduced operator new and references and thereby alleviated some of the problems. Java supports only references and consequently has no comparable memory access problems. It is not clear that there is any need for genuine dynamic memory allocation at meta-compile time, since FOG allocates and manages meta-objects automatically. A more pragmatic set of type constructors is therefore implemented, exploiting the freedom to define all meta-types and lists of meta-types as specialisations of the token meta-type.
Meta-Array-Of

C++ defines a compound-statement to be a statement, and an expression-list to be an expression. FOG extends this by defining a compound-declaration to be a declaration and a compound-tree-statement to be a tree-statement. There is clearly a significant polymorphism between an element and a list of those elements. It is therefore appropriate to define a meta-type as polymorphic to a list of the same or more specialised meta-type, so that we may use nested lists to handle arbitrary tree structures.

The list has many similarities to a C++ array and so we can safely re-use the nested {} syntax for list initialization and [] for indexing.

```cpp
auto string s = { "a", { "b", { "c", { "d" }}}};
```

defines a tree with a first, second, third and fourth generation descendant, as shown in Figure 4.6.

![Figure 4.6 Example Tree initializer](image)

There is no provision for uninitialized meta-variables and so no need to declare tree shapes independent of initializers. The initializers define the shape: explicitly as in the above example, implicitly when a declaration tree is assigned to a meta-variable.

Meta-programs need to be able to iterate over lists of declarations. A polymorphic iterator meta-type is therefore provided that acts as a pointer to a list.

With this system an iterator could be used as:

```cpp
auto for (iterator i = $bases(); i; ++i)
    $i->do_something();
```

bases() returns a list of base class specifiers. The iterator is initialized by the elements of the iteration domain, but why does iterator i = $bases() establish an iteration over the domain comprising the elements of the list rather than the domain comprising the list as a single element? While it is convenient to treat the list as polymorphic to an element, there must be a predictable mechanism for deciding when and by how much to flatten a tree. The simplest algorithm, which never flattens implicitly, does not support the above example.

An operator is necessary to expose the contents of the list, so that continuing the earlier example
Meta-compilation for C++

FOG Semantics

s[] is the two element list
"a", { "b", { "c", { "d" } } }

s[1][1][1] is the elemental list 'd'

allowing the iteration to be written:

auto for (iterator i = $bases(); i; ++i) // Non-FOG example
The extra [] changes the initialization to the elements of the list ensuring an
iteration over each base class. This is clumsy and prone to errors, since the [] is
too easily omitted.

We therefore extend the system to distinguish exposed and encapsulated lists.
An encapsulated list variable comprises a single element, which may be a list.

auto identifier encapsulatedList = { a, b, c, d };

An exposed list variable identified by a [] declarator suffix comprises the arbitrary
number of elements of a list.

auto identifier exposedList[] = a, b, c, d;

The {} and [] encapsulate and expose symmetrically, so that

exposedList is-the-same-as encapsulatedList[]
encapsulatedList is-the-same-as exposedList

Defining the initialization of an iterator and the return from built-in meta-functions
as exposed lists eliminates the need for the clumsy [] and allows the natural
programming style:

auto for (iterator i = $bases(); i; ++i)
$i->do_something();
or even

auto for (iterator i = $A::bases(), $B::bases(); i; ++i)
$i->do_something();
to achieve iteration over the concatenation of two exposed lists.

The hypothetical built-in declaration for bases is therefore

auto base class::bases() [];

and for iterator initialization:

auto iterator::iterator(token []);
auto void iterator::operator=(token []);

Meta-Pointer-To

This approach supports scalars, arrays and more generally trees, with all names
behaving as references. The limited need for a pointer is handled by the
polymorphic iterator.

Meta-Function-Returning

There is no direct support for pointers to functions, however functions and meta-
functions can be manipulated under the guise of the function or meta_function
meta-types, or more generally as declarations.

auto functionSpecifier add_constant(number fixedValue)
{
    inline double $(unique_name())(double functionArgument)
    {
        return functionArgument + $fixedValue;
    }
}

auto function plusTwo = $add_constant(2);
a = $(plusTwo.name())(b); // a = b + 2;
add_constant is a meta-function that returns the function-specifier (potential declaration) for an inline function which adds fixedValue to the functionArgument. The name of the returned function is determined by a unique name-generating meta function.

A potential declaration for a function that adds 2 is created by the invocation to initialize the plusTwo meta-variable. Since the required meta-type is for an actual function the inline function is created in the current (global) context. The name of the function is then used in the final line.

Meta-Record-Of

User defined meta-types are not supported in FOG. This seems a very natural extension when more substantial compile-time programming is required. A little careful thought is necessary to distinguish (or unify) the conflicting perspectives of user-defined meta-types as types constructed from existing meta-types, and as predicates upon extended syntax.

4.1.6 Meta-type conversions

When a meta-object is used as an intermediate term in a meta-expression, the meta-object may be suitable for direct use:

```cpp
$metaObject.meta_function();
auto int i = $metaObject1 + $metaObject2;
```

Alternatively, a conversion to a more suitable form may be required, which may involve:

- conversion of a potential declaration to an actual one
- resolution of the value of a meta-object
- conversion to a synonym
- conversion to a character, identifier, number or string meta-type
- conversion of an expression to an expression_statement
- conversion of an expression_statement to an expression
- conversion of the break keyword to a statement
- conversion of the continue keyword to a statement
- conversion of the return keyword to a statement

4.1.6.1 Meta-type Synonyms

The declaration and statement meta-types may be interchanged, as may the expression and initializer_clause meta-types.

4.1.6.2 Potential to Actual meta-type conversion

When a potential meta-type occurs in a context where an actual is required, an actual declaration is created from the potential in the prevailing context.

```cpp
auto enum_specifier potentialEnum = enum E; // No conversion
auto enum actualEnum = enum E; // Creates ::E
```

4.1.6.3 Resolution of the value of a meta-object

A tree-literal defines an expression to be resolved in the meta-name-space resulting in a meta-object whose value replaces the tree-literal. The replacement value must be a meta-object that has meaning beyond the meta-level; it must therefore be a character-literal, identifier, number-literal or string-literal, since these represent all possible alternatives for a C++ literal. This value is obtained by invoking the built-in conversion meta-function, operator identifier(), unless the tree-literal forms
part of a concatenation in which case one of `operator character()` , `operator number()` or `operator string()` may be invoked instead.

For the character, identifier, number and string meta-objects these conversion meta-functions perform conversions as described in Section 3.1.1.3.

For named meta-objects, the unqualified name is first expressed as an identifier, and then further converted if necessary. This supports the idiomatic

```
const char *class_name = ""$Scope;
```

rather than the more explicit

```
const char *class_name = "$name();
```

or even more explicit

```
const char *class_name = "$Scope.name();
```

Meta-variables, meta-iterators and meta-functions have no run-time object and so the appropriate conversion operator is applied to their value.

The value of a meta-function is the function body.

The value of a meta-iterator is a boolean meta-object, whose value is true while the iterator is valid. This supports:

```
auto for (iterator p = $functions(); p; ++p)
    /* ... */
```

### 4.2 Substitution

Substitution is triggered when a reference such as

```
$name
```

is detected that satisfies the *tree-literal* syntax described in Section 3.1.1.6. The reference clearly cannot be fully resolved until the definition is available, which requires that the semantic processing of the definition has completed.

The C preprocessor has an independent definition syntax using `#define` that can easily be fully analysed before processing continues. In FOG, the definition syntax is integrated with the rest of the language, where alternative substitution semantics impose distinct requirements on the overall processing.

#### 4.2.1 Substitution levels

Substitution of a segment of source code by a replacement is often called macro processing. A variety of models and classifications of macro processing are discussed in Section 2.1.2. The most appropriate for discussing substitution in FOG is the distinction between character-, token-, and syntax-level substitution.

These alternate levels will be considered while attempting to implement the following example meta-program, in which a loop populates an array with the value of the meta-variable used as the loop counter.

```
auto for (unsigned i = 0; i < 10; ++i)
    static const char *digits[] = { "$i
```

The example involves repeated interleaved definition and reference of a meta-variable. The statement should be equivalent to the following C++ declaration, for which the loop provides the values which are automatically comma-separated as FOG emits the composed list.

```
static const char *digits[] =
    { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" };
```

In each iteration, the identifier `i` must be resolved to the appropriate meta-variable, and evaluation must use the prevailing value of the meta-variable. (In
most examples, such as this one, \( i \) resolves to the same meta-variable on each iteration, however this cannot be assumed.

While maintenance of the loop counter is often restricted to the loop header, it is unsafe to assume that the loop body is clean:

```c
auto for (unsigned i = 0; i < 10; ++i)
{
    auto if (i == 5)
        auto i += 2;
    static const char *digits[] = {
        "$i
```
4.2.1.2 Token-level substitution

Token-level substitution performs substitution after lexical analysis has completed the conversion of source text into a token stream. This improves efficiency and provides for a more predictable environment but loses flexibility.

Operation at the token-level prevents the formation of composite punctuation tokens such as `+=`. This is probably beneficial. However, losing the ability to create extended identifiers and the ability to convert character sequences to strings is restrictive. The ANSI C preprocessor uses this substitution model and introduces the `#` and `##` operators to provide a slightly clumsy way around the restrictions.

Token-level substitution prohibits syntactic analysis of the replacement tokens in isolation, since adjacent tokens may influence the meaning; syntactic analysis must be repeated for each replacement and error diagnostics associated with instantiation rather than definition.

Token-level substitution simplifies the example only slightly, requiring the preliminary parse to resolve:

```c
auto for (unsigned i = 0; test-tokens; step-tokens) body-tokens
```

The special scanner now caches token sequences rather than character sequences, but is still vulnerable to unexpected replacement behaviour that involves the structuring tokens.

Re-entrant invocation of syntactic and semantic processing is again necessary to interpret the cached token sequences during syntactic analysis of the loop.

A form of this token-level substitution was used by the multi-pass grammar implementation of FOG, which performs syntax-driven re-entrant analysis. The difficulty of defining the token `{` (for use in a replacement) rather than the punctuation `{` (for defining program structure) was at one point assisted by introducing the additional lexemes `\{` and `\}`). This fudge was then replaced by an identifier to punctuation conversion fudge in the phase 7 tokenization grammar. Neither fudge is required in the next approach:

4.2.1.3 Syntax-level substitution

Syntax-level substitution uses a syntax tree from an earlier syntax analysis as the replacement. The replacement is therefore constrained to satisfy an explicit syntactical type, rather than the much weaker constraints of sequence-of-character or sequence-of-token.

Syntax level substitution
- further improves efficiency since no source code experiences repeated lexical or syntactic analysis
- supports improved error diagnosis since the definition of a replacement can be checked syntactically
- avoids the syntactical interpretation changing in response to its instantiation context (the parenthesis problem of Section 3.4.1).

The informal restriction on the occurrence of structuring tokens is removed, or rather becomes intrinsic to the substitution model; it is difficult to define structuring tokens as replacements, since partial syntax does not satisfy syntactical requirements.

Direct syntactic analysis of the entire example is supported by incorporating an AST node for `$i$. This node is then resolved and evaluated during each iteration of the loop. With syntax-level substitution it is possible to complete syntactic analysis of the entire program before any semantic analysis starts. Re-entrancy occurs, but only within the confines of the semantic analysis.
The superset grammar implementation of FOG uses syntax-level substitution and avoids all context dependencies. Substitution below syntax-level is supported by deferred character-level substitution:

```cpp
  auto string braced_else = "} else";
  $std::parse("if (a) { b; " $braced_else " c;}
    // if (a) { b; } else c;
```

Although this supports dubious concatenation, the prevailing syntax type must always be syntactically valid. The mismatching `}` preceding the `else` is only mismatched while represented as a string. Conversion to a useful syntactic element must wait until string concatenation provides valid source for `std::parse` to convert to a generic syntax element: in this case a `selection-statement`.

For the C preprocessor, there is no looping and so references always occur after their definitions.

```cpp
#define SOME_THING

... SOME_THING ...
```

There is no flexibility in the location of a resolved definition. Replacements occur directly, with complexity arising only for formal parameters of function-like macros, for which replacement text is affected by the later actual argument.

```cpp
#define DO_SOME_THING(ResolveLater) ... ResolveLater ...

... DO_SOME_THING(WITH_THIS) ...
```

The same is not true for FOG.

While syntax-level substitution is efficient and predictable, it inhibits character-level and token-level substitution.

In practice, many substitutions would satisfy the constraint that an invariant definition lexically precede a reference from a determined context. So it is tempting to allow such backward reference substitutions in which replacement characters or tokens could be substituted while the source was at character or token level. However this would be a special behaviour that could change unexpectedly as a program evolved. Provision of `std::parse` to allow an explicit recursion from semantic level back to the lexical level seems to satisfy the potential requirements for lower level substitution without introducing irregular behaviour. Other requirements are satisfied by the concatenation mechanism.

### 4.2.1.4 Semantic-level substitution

The three preceding levels of substitution correspond to different degrees of validity for the definition of a replacement:

A character-level replacement is lexically indeterminate:
- an arbitrary sequence of valid characters

A token-level replacement is lexically valid but syntactically indeterminate:
- an arbitrary sequence of valid tokens

A syntax-level replacement is syntactically valid but semantically indeterminate:
- an arbitrary tree satisfying a valid syntax

A fourth semantic-level substitution is possible using a replacement that is semantically as well as syntactically valid. This level is not possible in FOG, since semantic analysis requires type information which may come from the surrounding context. The replacement cannot be semantically analysed in isolation. Overall semantic validity is determined where the replacement is used, although some checks can be made at the definition site.
4.2.2 Syntactic Polymorphism

Replacement of syntactically consistent trees allows some rather dubious programs to be written:

```c++
auto expression decl0 = i;
auto declaration decl1 = int i;
auto for (unsigned j = 0; j < 2; ++j)
  $decl$j = k;
```

On the first iteration `decl0` is selected and so the loop body is an expression:

```c++
i = k;
```

On the second iteration using `decl1`, the loop body is a declaration:

```c++
int i = k;
```

The variation in syntactical type is legal, since there is no requirement for uniformity.

The declaration form is only legal in FOG because `int i` satisfies the generalised naming that resolves `int i` as a name before assigning `k` to it, resulting in the syntax tree shown in Figure 4.7.

![Figure 4.7 int i = k](image)

This is not completely consistent with the exposition of the C++ grammar in the standard, for which `i = k` is resolved as an `init-declarator`, before the `int` prefix is applied as the `decl-specifier-seq` of a `simple-declaration`. However, since C++ ascribes no semantics to partial declarations, FOG does not create an incompatibility by defining them.

The ordering in the FOG grammar is necessary to support context free syntactic analysis. The FOG grammar should therefore be expressed in a way more closely resembling the practical implementation in Appendix C rather than the changed C++ grammar of Appendix A.

4.2.3 FOG substitution

The C++ reference model (§2.1) provides for each of the minor translation phases to be performed in sequence. However, the many responsibilities of phase 7 (the main compilation phase) are just bundled together, reflecting the apparent need for syntactic and semantic analyses to be tightly coupled.

In FOG, resolution of a `tree-literal` is detected during the phase 6 (string concatenation) processing but requires the use of phase 7 syntactic processing to identify the `tree-expression` syntax and any meta-function arguments. FOG originally used syntax-driven parsing to resolve each argument according to its known syntax. This imposed tight semantic coupling and required deferred analysis in contexts where the syntax was initially unknown. FOG is now context-free and so no syntactical knowledge or deferral is required.

1. The right to left evaluation of nested $-expressions is explained in Section 3.1.1.6.
The early translation stages naturally operate in a demand-driven fashion with the phase 7 syntactical analysis, in the main grammar, making repeated requests to a phase 6 procedure to provide the next token for syntactic analysis. When phase 6 detects a tree-literal, phase 7 processing is activated re-entrantly to acquire and analyze the source tokens of the tree-literal. These are returned to the interrupted phase 7 processing as a tree-literal token, which describes the unresolved tree-expression. Recognition of tree-literals may occur re-entrantly to arbitrary depth.

FOG therefore introduces a need for re-entrant invocation of phase 7 syntactic processing, but achieves an overall simplification through the separation of syntactic and semantic processing. All syntactic processing can be completed before any semantic processing starts.

This is important since, in general, semantic information is not available. When the tree-literal is encountered in the meta-function body of:

```
auto declaration defer(class clientClass)
{
    ${$(clientClass)::deferred($Scope));
}
```

it is only known that clientClass identifies a class. It is not known which class, so the existence or signature of the deferred meta-function cannot be determined, and the required syntax type of the parameter is unresolvable. The invocation context is unknown, so the syntactical type of the $Scope argument is also unknown.

Syntax-driven parsing is not possible when the syntactical type of either a formal parameter or an actual argument cannot be determined. In the example, neither can be known. The FOG grammar must therefore be context-free.

The example is resolved by initially parsing only for a generic syntax element. Eventually the syntax element is used in a deterministic context where semantic interpretation ensures compliance with the syntax types.

tree-literal definition

Definition of a tree-literal in one context, generally involves names that are resolved in another context.

```
auto expression e = a(*b);
```

The initializer a(*b) is therefore parsed against the generic tree-statement syntax, which accepts almost any C++ sentence, including many that are not expressions. Association of the initializer with the meta-variable performs only a weak semantic validation to verify that the initializer could satisfy the syntax of an expression. It does not matter that the initializer could satisfy more than one syntax: a function call or a function parameter with a redundantly parenthesised parameter.

When the meta-variable is used in a determined context:

```
class X
{
    typedef int b;
    class $e; // class a(*int);
};
```

two errors can be detected. First that the value of e does not satisfy a strong semantic check of the syntax of an expression, and secondly that this particular expression is not a valid name for a class.

The first of these errors can be subverted by use of a syntactically weaker meta-type.

---

2. Semantic processing of syntax macros must occur at the syntactic level.
auto token f = $e;
class $f;

The original initializer is checked to verify that it could satisfy the syntax of a token (which everything satisfies). When class $f$ is checked, the strong check then detects that this particular token is not a valid name for a class.

tree-literal syntactical analysis

The deferred invocation example

```cpp
auto declaration defer(class clientClass)
{
    ${${clientClass}::deferred($Scope)};
}
```

showed that the syntactic type of tree-literals is not necessarily known during syntactic analysis of sentences involving tree-literals. This would appear to preclude syntactic analysis of the meta-function body at the definition site. Even if the syntax type could be known, its use introduces a tight coupling between the semantic analysis that determines syntax types and the syntactic analysis that uses these types.

Revising the C++ grammar to incorporate indeterminate contributions is not possible. The result would be totally ambiguous:

```cpp
$a $b $c $d;
```

could be a namespace-alias-definition:

```cpp
namespace x = y;
```

amongst very many alternatives.

A useful compromise leaves the C++ grammar almost untouched, but supports most practical replacements. The compromise makes the simple assumption that each occurrence of a tree-literal represents an identifier. Syntactical analysis and the grammar is therefore only affected by the categorisation of a tree-literal as an identifier.

This assumption imposes minor limitations. tree-literals cannot be used to source keywords or punctuation where these define program structure. The use of $a and $c for the namespace-alias-definition example is therefore impossible.

The assumption is much less restrictive than it might appear. It does not require the tree-literal to be an identifier, merely to be used where an identifier could be used.

identifier is a degenerate sentence for many syntaxes:

- a type name (since there is no type/non-type discrimination)
- any form of expression (identifier is a primary-expression)
- a parameter-declaration(-clause) (identifier is a type)
- a decl-specifier(-seq) (identifier is a type)

identifier ; is a degenerate sentence for most other syntaxes:

- any form of statement (identifier ; is a degenerate expression-statement)
- any form of declaration (identifier ; is a degenerate simple-declaration)
It is only for very narrow syntaxes involving just reserved words or punctuation that a substitution cannot be allowed, for example:

- `access-specifier`
- `class-key`
- `cv-qualifier`
- `function-specifier`
- `operator`
- `ptr-operator`
- `storage-class-specifier`
- `unary-operator`

These limitations do not prevent definition and usage of these syntactic types, however they do prevent parameterisation of syntax that uses them. For instance, a `class/struct/union` tag cannot easily be used directly:

```cpp
auto class_key ClassKey = struct; // Ok
$ClassKey Class { /* ... */ }; // Illegal
```

Some of these restrictions can be worked around by using a wider syntactical type such as `decl-specifier` rather than `cv-qualifier`. Others could be worked around by relaxing the assumption that all `tree-literals` are `identifiers` so that a functional cast can specify the built-in meta-type.

```cpp
$class_key($ClassKey) Class { /* ... */ };
```

However it seems perverse to cast something to its own type, and even more perverse to do so without validation. It seems that some variant of a `$-expression` is needed that uses the semantic knowledge where the programmer can guarantee that the type satisfies 'appropriate' constraints on being 'adequately' resolvable. Choosing a satisfactory syntax and resolving 'appropriate' and 'adequate' is a matter for further research.

Restrictions involving punctuation or reserved words acting as punctuation cannot sensibly avoided. There is no point introducing single-valued 'casts' to reserved words or punctuation, when the reserved words or punctuation could be more easily used directly:

```cpp
$'namespace'($a) $b $'='($c) $d; // Silly non-FOG example
```

### 4.3 Name Resolution

In C++, there are three categories of name-space (as distinct from `namespace`).

- the hierarchical name-space of (run-time) program declarations
  - template-names
  - type-names
  - non-type-names
- the per-function name-space of labels available for use by `gotos`.
- the C preprocessor macro definition name-space

FOG adds

- the hierarchical meta-name-space of (compile-time) program declarations.

FOG removes the distinction between template, type and non-type names at the meta-programming level, since meta-programming may occur before such distinctions are possible. The distinction is preserved where C++ semantics apply.

The C++ name-spaces are used for resolving names of conventional and extended C++ declarations, and for the scopes of meta-names.

The meta-name-space is explicitly used when resolving `tree-expressions`
$\text{Scope} \quad \texttt{bases()[0].is\_virtual()}$

and implicitly by the left-hand side of meta-expressions (within meta-statements)

```cpp
auto if (meta\_variable) /* ... */;
```

Resolution of a tree-expression involves four phases
- location of an object in the run-time name-space
- location of the meta-object that describes it
- evaluation of expression operators that use meta-objects
- conversion of the resultant meta-object to suit the invoking context

A meta-object for use in tree-expression is identified by a meta-scoped-id

```
meta-id:
  id
  meta-type
  auto
meta-nested-id:
  meta-id
  ~ meta-id
  meta-id :: meta-nested-id
meta-scoped-id:
  ::opt meta-nested-id
```

The names are resolved in the conventional name-space, with the minor generalisation that meta-classes such as unsigned int or auto, which have no conventional classes are treated as having empty classes.

```
\begin{itemize}
  \item «instance meta-meta-object»
  \item «class meta-object»
  \item «meta-class»
    \item X::Y
  \item meta-member-names
  \item «meta-function meta-object»
    \item X::Y::g
  \item «describedBy»
  \item «describedBy»
  \item «describedBy»
  \item meta-meta-level
\end{itemize}
```

```
\begin{itemize}
  \item «instance meta-meta-object»
  \item «class meta-object»
  \item «meta-class»
    \item X::Y
  \item meta-member-names
  \item «function meta-object»
    \item X::Y::f
  \item «describedBy»
  \item «describedBy»
  \item «describedBy»
  \item meta-level
\end{itemize}
```

Figure 4.8 Meta-function and function meta-object distinction

Thus, considering the meta-objects in Figure 4.8, the tree-literal

```
$X::Y::f.is\_static()$
```

is resolved by successively locating X, Y and f using the conventional name-space to identify the «function meta-object» that describes the «function» X::Y::f. Invocation of is\_static() upon this «function meta-object» tests whether the «function» X::Y::f has a static qualifier.

Whereas in:

```
$X::Y.g.is\_static()$
```
x, y are successively located using the conventional name-space to identify the «class meta-object» that describes the «class» X: :Y (not «instance», since X: :Y is a class-name not an instance-name). Selection of the g member selects the child «meta-function meta-object» named g. Invocation of is_static() upon this «meta-function meta-object» tests whether the «meta-function» X: :Y::g has a static qualifier.

These two mechanisms can coexist without conflict, since the «function meta-object» describing a «function» is not a «meta-function». This is shown in Figure 4.8 and discussed in greater detail in Section 4.5. The meta-objects of the members of a «class» are not members of their corresponding «meta-class». Therefore :: always performs resolution of references in the class/namespace hierarchy, and . or -> are resolved within the prevailing name-space.

Note that this has the corollary that when defining a meta-variable or meta-function, either :: or . may be used for the final scoping:

```cpp
auto int Class.meta_variable = 0; // Correct
auto int Class::meta_variable = 0; // Also correct
```

Note also that, as might be expected,

```cpp
$v.is\_static()
```

is a short form of

```cpp
$the\_current\_context^3.v.is\_static()
```

so that v is located in the meta-object describing the current context. The meta-object describing v is not the start of the resolution.

The meta-name-space is used for meta-class, meta-function and meta-variable names. It is not used for meta-function arguments or the right-hand side of a meta-assignment, which, in the absence of further $ or @ operators, represents a literal AST to be interpreted in the usage context.

Search for a name is always restricted to the expected name-space, however it should be noted that template parameters are present in both name-spaces.

A $ or @ prefix is necessary to access the meta-name-space where the run-time name-space would normally be used.

### 4.3.1 Search Locations

C++ has complicated rules (§3.4) for determining when and where names should be resolved:

A name may be resolved within

- the enclosing local block hierarchy
- the local object
- a parameter list
- the local class
- the base class hierarchy
- the enclosing (structural) class hierarchy
- the enclosing (structural) namespace hierarchy

---

3. the-current-context is This, and so $This.v.is\_static() may be used. However, it is still necessary to use the meta-object describing the current context to locate This, since This is defined only in the meta-name-space.
FOG adds further complications. The resolution of a name may be available within
- the meta-name-space
- the enclosing lexical hierarchy
- the local block hierarchy
- the root or derived scope of an actual declaration
- the definition or invocation context of a meta-function

The C++ name resolution rules are complicated, and so arbitrary extension to cover new possibilities would cause considerable confusion. Resolution of names within different name-spaces in conventional contexts is therefore signalled explicitly by the @ or $ operators.

4.3.2 Meta-name-space contents

Resolution of a name within the conventional run-time name-space proceeds according to the rules defined in the C++ standard. Resolution within the meta-name-space is triggered by the use of a $ or @ prefix to establish a tree-literal, or by the use of a meta-scoped-id on the left-hand side of a meta-expression.

Resolution of a name within the compile-time meta-name-space proceeds according to similar principles as run-time resolution, but without the context-dependent visibility of type-names. All names in a meta-name-space context are visible to all searches of that context. The search for a name proceeds in stages, with different candidate sets of locations considered in each stage. The search terminates at the end of the first search stage in which a definition is found, or when all stages have been completed. It is an error for more than one distinct resolution to be found in a search stage.

The search stages are in order:

Local scope

The search for a meta-name in a meta-program may find:
- meta-variables declared in a local scope established by a loop or compound-statement.

Local object-scope

The search for a meta-name in a meta-object may find (using a function as an example):
- meta-functions declared in an object-statements-clause
- meta-variables declared in an object-statements-clause
- the built-in meta-variables of the function meta-type
  - This resolving to the meta-object of the function
- the built-in meta-functions of the function meta-type
Meta-class

The search for a meta-name in a class meta-object occurs either through use of the class, or through failure to resolve the name in a local-object scope. The search for a meta-name in a class meta-object may find:

- meta-functions declared in a class-specifier
- meta-variables declared in a class-specifier
- the built-in meta-variables of the class meta-type
  - This resolving to the (meta-)class
  - Scope resolving to the (meta-)class
  - OuterScope resolving to the less nested (meta-)class
- the built-in meta-functions of the class meta-type
- the formal template parameters.

Search of a class meta-object does not find:

- the class name or base class names
- nested types (classes, enums or typedefs)
- member functions
- member variables
- enumerators

or their meta-objects.

Base Meta-classes

If a meta-name is not found directly in a meta-class, the search proceeds recursively by searching each base meta-class and recurses for each of the base classes, stopping individual searches when a name is found or after the root base meta class has been searched. Repeated resolutions with the same value are discarded. Ambiguous resolutions are an error.

Less nested base meta-classes

If a meta-name is not found in a meta-class or its base meta-class, the search is repeated successively for the meta-classes of each hierarchically less nested class stopping once resolved, or when the namespace scope is reached.

Meta-namespace

The search for a meta-name in a meta-namespace may find:

- meta-functions declared in a namespace-body
- meta-variables declared in a namespace-body
- the built-in meta-variables of the namespace meta-type
  - This resolving to the (meta-)namespace name
  - Scope resolving to the (meta-)namespace name
  - OuterScope resolving to the (meta-)namespace name
  - Namespace resolving to the (meta-)namespace name
  - OuterNamespace resolving to the less nested (meta-)namespace name
- the built-in meta-functions of the namespace meta-type
Less nested meta-namespaces

If a meta-name is not found in the namespace, the search is repeated successively for each structurally less nested namespace stopping once resolved, or after the unnested global namespace has been searched.

The built-in functions and variables for the meta-types are summarised in Appendix E.

4.3.3 The Substituted Value

When a substitution requested by a $ or $-expression is resolved, the expression is resolved in the invoking context to identify the meta-object whose value replaces the expression.

\$i[1]

The meta-object must necessarily have already been defined, at which point any $-expressions that reference the defining context are resolved.

auto token i[] = { "$Scope, "$Super }

Once the meta-object has identified any $-expressions referring to formal arguments are replaced. This involves not only the explicit parameters of a meta-function, but also the implicit Static, Dynamic and template parameter names for meta-functions or meta-variables.

Each replacement takes the form of an already parsed syntax tree, albeit with residual $ and $-expressions which are then resolved after substitution into the invoking context. Execution of a meta-function body therefore occurs within the invoking context, since:

Multi-$ expressions are resolved when the meta-function is defined (see Section 4.3.6).

Single-$ expressions are replaced as the meta-function body is substituted within the invoking context.

$-expressions are resolved, and meta-statements executed each time the expression is analysed within the invocation context.

The following example

auto statement nextDay(string dayText)
{
static const char *dayTable[] = { $dayText };
auto dayCount++;
}

auto int dayCount = 0;
$nextDay("Sunday");
$nextDay("Monday");
$nextDay("Tuesday");
$nextDay("Wednesday");
$nextDay("Thursday");
$nextDay("Friday");
$nextDay("Saturday");
static const int maxDay = $dayCount;

results in the equivalent C++ code:

static const char *dayTable[] =
{ "Sunday", "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday" };
static const int maxDay = 7;

showing how the dayCount++ expression is executed seven times within the invoking context. The array initializer is similarly executed seven times, with the
initializer resolved from the formal argument; the redeclarations compose as described in Section 4.4.7 to give the composite array declaration.

4.3.4 Derived context resolution

A declaration specified using a derivation rule may need to use names referring to the root scope or the derived class. The following example (elaborated in Section 7.1.4) defines a clone method for a RootClass and a derivation rule to implement the protocol automatically for all derived concrete classes.

```cpp
public virtual $Scope *RootClass::clone() const = 0 {
    derived(!Scope.is_pure())
    { new @(Scope){*this}; }
};
```

The body of the clone method must refer to the derived class, but the return type must use the root scope (on pre-standard compilers).

The two requirements are resolved by the different substitution operators.

A $-expression is resolved within the defining context of a declaration. This is RootClass in the above example.

An ©-expression is resolved as late as possible, which is generally within the actual context of a declaration. This is the derived class in the above example. 'As late as possible' may arise at three distinct times, depending upon the context of the ©-expression:

©-expression in scope

An ©-expression forming part of the scope of a declaration is resolved when the declaration is interpreted in a determined context in order to install the declaration in the determined scope. This applies to each ©-expression in:

```cpp
class ©Outer {
    int ©Inner::i;
};
```

Since this form of usage that creates an actual declaration directly, there is no distinction between a (single $) $-expression and an ©-expression. However a distinction does arise when the conversion to actual is deferred; the ©-expression is resolved in the actual context, whereas a $-expression is resolved in the defining context.

```cpp
auto identifier Outer = "Outside";
auto identifier Inner = "Inside";
auto class_specifier cachedClass1 = class $Outer {
    int $Inner::i;
};

auto class_specifier cachedClass2 = class ©Outer {
    int ©Inner::i;
};
```

This defines the meta-variable cachedClass1 with the value

```cpp
class Outside { int Inside::i; };
```

and the meta-variable cachedClass2 with the value

```cpp
class ©Outer { int ©Inner::i; }
```

Immediate usage in which the invoking context is the same as the defining context $cachedClass2;
results in

```cpp
class Outside { int Inside::i; }
```
as would also be the result for $cachedClass1. However $cachedClass2 can be invoked from an alternate context:

```cpp
class AnotherContext
{
  auto identifier Outer = "Without";
  auto identifier Inner = "Within";
  $cachedClass1; // class Outside { ... }
  $cachedClass2; // class Without { ... }
};
```

with the result that the deferred resolution of $cachedClass2 uses the distinct invocation context.

**@-expression in name**

An @-expression occurring as part of the naming of a declaration is resolved when the signature is resolved to install a declaration in its actual scope. Resolution of the @-expression may then respond to any change of scope caused by derivation rules, or meta-programming.

The naming comprises the name, types and parameter names, but not default initializers. (Parameter names do not form part of the unique signature of a declaration.) This therefore applies to each @-expression in:

```cpp
class X
{
  auto string prefix = "p";
  @Scope *accept_@{Scope}@Scope& @{prefix}@Scope)
    :{ derived(true) { return &@{prefix}@Scope; } };
};
class Y : public X
{
  auto prefix = "q";
};
```

which consequentially generates the functions:

```cpp
X *X::accept_X(X& pX) { return &pX; }
Y *Y::accept_Y(Y& qY) { return &qY; }
```

The lexical concatenation of accept_ and @{Scope} although syntactically analysed as an identifier remains unresolved until the declaration is installed in its actual scope.

**@-expression in body**

An @-expression occurring as part of the body of a declaration is resolved when the body is compiled in the final compilation phase prior to emission. This differs only slightly from the earlier example, by deferring resolution till all meta-programming for all declarations has completed, except for side effects caused by other @-expressions also in declaration bodies.
The body of a declaration comprises
- the body of a function
- the handler of a function-try-block
- the initializer of a ctor-initializer
- the parameters of exception-specification
- the initializer of a variable
- the default-initializer of a function argument
- the predicate of a derivation rule

```cpp
static int X::member_variables = @variables().size();
static bool X::f(int i = @default_init()) {
    derived(@predicate()) { return i == @threshold(); }
};
```

Thus:

```cpp
class HeapAllocated {
    protected virtual ~@{Scope}() :{ derived(true) {} },-
};
```

enforces protected access on the destructor throughout a class hierarchy. The derivation rule ensures regeneration in all derived classes, and ~@{Scope} specifies the correct name in each derived class.

Note that:

```cpp
class Base {
    @{Scope}(const $Scope&) :{ derived(true) { /* ... */ } }; 
};
class Derived : public Base {};
```

provides a copy constructor in the root scope where @{Scope} and $Scope are the same class, and so generates:

```cpp
Base::Base(const Base&) { /* ... */ }
```

but an ordinary constructor in derived classes, where the failure to defer resolution results in:

```cpp
Derived::Derived(const Base&) { /* ... */ }
```

The final $Scope must be replaced by @{Scope} to specify a copy constructor throughout the class hierarchy.

### 4.3.5 Lexical scope resolution

In C++, there is little need to use lexical nesting for name resolution since structural nesting is substantially the same.

```cpp
class Outer // ::Outer
{
    public:
        /*...*/ name /*...*/; // ::Outer::name
    class Inner // ::Outer::Inner
    { /*...*/ name /*...*/ // resolved in structural scope.
    };

    Resolution of name searches in ::Outer::Inner, then ::Outer (and finally ::).
```
The FOG generalisation to support interleaved declarations and arbitrary scopings means that a lexically enclosing scope is not necessarily a structurally enclosing scope.

namespace Domain {
    class Outer
    {
        public:
            /* . . . */ name /* . . . */;
        class ${Namespace}::Sibling
        {
            /* . . . */ $name /* . . . */
        };
    };
}

Resolution of $name searches in Domain::Sibling and Domain and ::, but not in Domain::Outer, so the definition is only visible when explicitly qualified as $Outer::name. The lack of visibility introduces significant problems for a metafunction such as:

    auto declaration create_sibling()
    {
        /* . . . */ name /* . . . */;
        public class ${Namespace}::Sibling
        {
            /* . . . */ $name /* . . . */
        };
    }

The meta-function establishes some declarations, such as name, in its invocation context. These should be visible to other declarations in lexically nested contexts. The surrounding context is available as $Scope, which could be used to explicitly qualify the reference, except that the reference occurs within a lexical nesting that redefines Scope. Caching the value of Scope in a temporary meta-variable across the lexical boundary is possible but hardly elegant.

A solution is provided by extending the $ operator.

    $x resolves x in the prevailing context.
    $$x resolves x in the surrounding lexical context.
    $$$x resolves x in the surrounding surrounding lexical context.
    etc.

The meta-function can therefore be written in re-usable fashion as

    auto declaration create_sibling()
    {
        /* . . . */ name /* . . . */;
        public class ${Namespace}::Sibling
        {
            /* . . . */ $$name /* . . . */
        };
    }

4.3.6 Defining or invoking resolution

Name references within C++ functions normally occur as if resolved within the run-time execution context. With the exception of virtual functions, these resolutions actually occur at compile time, and therefore make use of the defining context which is known at compile-time.
Name references within FOG meta-functions occur when the meta-function defined in a (definition) context is invoked from another (invocation) context. Both contexts are known at compile-time, and since meta-programming is manipulating declarations, resolution within the invocation context as well as the definition context can be useful.

The semantics of meta-function execution is that formal parameters are replaced and then the entire body of the meta-function is returned for interpretation within the invocation context (see Section 4.3.3). This ensures that the invocation context can be manipulated but prevents the definition of a meta-function reacting to its defining context. The following example attempts to conditionalise the behaviour of a meta-function in accordance with a `DEBUG` command line variable:

```cpp
auto bool debug = $std::get_cpp("DEBUG");
auto declaration f()
{
    if ($debug) // Not the way to do it
        /* ... */
    /* ... */
}
class X
{
    auto declaration debug() { /* ... */ }
    $f();
};
```

Resolution of `$debug` occurs within the invocation context and so the conditionalisation is resolved by the non-zero body of the meta-function `X::debug`.

This problem is well known to Lisp programmers who call it the functional argument or FUNARG problem. A segment of code with free symbols is passed to a context where the free symbols may obtain an unexpected resolution.

Although this is a different problem to the lexical scoping problem, it is amenable to the same solution. Defining a meta-function declaration as establishing a nested lexical scope, causes the rewritten meta-function:

```cpp
auto declaration f()
{
    if ($$debug)
        /* ... */
    /* ... */
}
```

to perform the resolution of `debug` in the surrounding lexical context ensuring the intended resolution to `::debug`.

### 4.3.7 Multi-$expression resolution

The generalisation of `$`-expressions is logical and predictable. The following example has nested meta-functions defined within nested classes. The outer meta-function `A::B::f()` is invoked within `D::E` to define the inner meta-function `D::E::A::g()`, which is invoked within `F::G`.

```cpp
//6
class A
{//5
class B
{//4
    auto declaration f() // A::B::f
        /* ... */
}//3
class A
{//2
```
auto declaration g()
{ //1
   /*reference*/
} ;
}
};
class D
{
   class E
   {
      $A::B::f();
   } ;
};
class F
{
   class G
   {
      $D::E::A::g();
   } ;
};

Considering alternative invocations at /*reference*/, for which Scope is a convenient built-in meta-variable to demonstrate the context in which name resolution occurs:

$Scope resolves within //1, which is the meta-function body that replaces the invocation as $D::E::A::g() from ::F::G. It therefore resolves to ::F::G.

$\$$scope resolves within //2 which is nested scope within meta-function body that replaces the invocation as $A::B::f() within ::D::E. It therefore resolves to ::D::E.

$\$$\$$scope resolves within //3, which is the meta-function body that replaces the invocation as $A::B::f() within ::D::E. It therefore resolves to ::D::E.

$\$$\$$\$$scope resolves within //4, which is a nested scope and resolves to ::A::B.

$\$$\$$\$$\$$scope resolves within //5, which is a scope and resolves to ::A.

$\$$\$$\$$\$$\$$scope resolves within //6 which is the global namespace ::.

$\$$\$$\$$\$$\$$\$$scope is an error since there is no surrounding context for the global namespace.

And for completeness:

Scope is an identifier.

@Scope resolves to ::F::G, or a class derived from ::F::G, if the reference occurs within a declaration that is regenerated by a derivation rule or meta-program.

This is clearly not a complete set of all contexts, since neither D nor F appear although they constitute lexically surrounding scopes of contexts that do appear. However the existence of D and F are implementation details that cannot be known to the author of the meta-functions. A more elaborate 2-dimensional scheme supporting access to the third surrounding lexical context of the second surrounding meta-function invocation would therefore be inappropriate.

The relevant lexical context is readily identified by counting back through braces surrounding the /*reference*/ invocation expression. It is independent of the subsequent usage. Each additional $ causes resolution in a less nested context: the invocation context for a meta-function body or the internal context of a class body.
Nested lexical contexts are established:
• between ( and ) of a class (or namespace) body
• between ( and ) of a meta-function body
• between :{ and } of an object-statements-clause

Nested lexical contexts are not established for:
• { and } or :{ and } within :{ and }
• structural nesting established by ::
• compound statements
• compound declarations
• filespaces (see Appendix F.4.4)
• initializer clauses

4.3.8 Transferred lexical scope for object-statements-clauses

When meta-function or class bodies nest within an object-statements-clause, the object-statements-clause establishes a lexical context. The function bodies do not establish a further context.

```cpp
auto statement f()
:{ // Outer grouping :{} is a lexical scope
derived(true)
:{ // Grouping :{} is not a lexical scope
text(entry
{ /* ... */
};
);
}
};
```

4.3.9 No lexical scope for :: nesting

It might seem that :: should establish a nested lexical scope so that there is no difference between the following pair of meta-functions:

```cpp
auto identifier A::B::f() { $$Scope; }
class A
{
class B
{
    auto identifier g() { $$Scope; }
};
};
```

However this would introduce an inconsistency for

```cpp
auto $X A::B::h() /* ... */
```

which by analogy with the C++ declaration

```cpp
X A::B::h() /* ... */
```

would suggest that $X be resolved in its surrounding (global) context rather than the nested A::B context. There is no strong reason why $f() and $g() should behave in the same way. C++ consistency requires that they do not. Therefore :: does not establish a lexical scope.

4.3.10 No Lexical scope for initializers and arguments

Not establishing lexical contexts for initializer clauses seems obvious since in

```cpp
int scalingMatrix[2][2] = ( ( $Scale, 0 ), ( 0, $Scale ) );
```
the braces are a grouping operator, they do not create a lexical context in which
declarations could be differently resolved.

It is less clear whether the = and ; of a variable initializer or the ( or , and , or )
surrounding a meta-function argument should establish a nested lexical scope.

At first sight it is clear that in

```cpp
class X
{
    auto identifier p = $Scope;
    $f($Scope);
};
```

$Scope should be resolved as X using the prevailing context. However, the
semantics of meta-function execution return the body of a meta-function for
interpretation in the invoking context. Perhaps the initializers and arguments
should also be interpreted within their usage context. It is then necessary to use
$$Scope to ensure that the above example has the obvious behaviour.

Considering a more complicated meta-function call in which a `class-specifier` is
passed as a meta-function parameter to create the class `To::X`.

```cpp
auto declaration f(class_specifier c)
{
    class ::To
    {
        $c;
    };

class Call::From
{
    $f(class X
    {
        ... $Scope ...
        ... $$Scope ....
    });
};
```

which expands to

```cpp
class Call::From
{
    class ::To
    {
        class X
        {
            ... $Scope ...
            ... $$Scope ....
        };
    };
};
```

$Scope should clearly resolve to `::To::X`, rather than `Call::From` (which is not
visible from `::To::X`), or `Call::From::X` (which may not ever exist).

$$Scope resolves to a surrounding context, but is it `Call::From` (that surrounding
the definition of the argument) or `::To` (that surrounding the instantiation
as `::To::X`)?

Either resolution is tenable, however resolution as `To` in the invoking context
involves the use of a surrounding context that is not lexically identified where a
meta-variable is initialized or where a meta-function argument specified. It does
not seem necessary to make this scope easily accessible. This invisibility of
remote contexts mirrors the invisibility of the remote lexical scopes for `D` and `F` in
the nested class and meta-function example of Section 4.3.7. Therefore
initializers and meta-function arguments do not establish a nested lexical scope: resolution of $-expressions has the obvious behaviour.

Although the remote surrounding lexical context is not directly accessible, it can be accessed indirectly. When $x$ is structurally as well as lexically nested, $\text{Scope}$ can be resolved in the surrounding structural scope by $\text{OuterScope}$. Even when $x$ is not structurally nested, access is possible by creating a dummy lexical and structural nesting:

```cpp
class From {
    $f$(auto class ExtraNesting {
        class ::X {
            ...
            ...
            $\text{Scope}$ ...
            ...
            $\text{OuterScope}$ ....
        };
    });
};
```

The structurally and lexically nested $\text{ExtraNesting}$ meta-class establishes an additional lexical context surrounding $x$, which can be reached using $$, and from which $\text{OuterScope}$ can traverse to the surrounding structural (and lexical) context.

### 4.3.11 Formal parameters

Formal parameters are visible throughout a meta-function-body, including any nested lexical contexts, so it is unnecessary to use additional $'$s to access formal parameters from within nested lexical contexts.

```cpp
auto declaration X::f(identifier outerParameter) {
    $\text{resolved\_in\_invocation\_context}$;
    $$\text{reaches\_out\_to\_defining\_context}$;
    auto declaration h(identifier innerParameter) {
        class $\text{outerParameter}$ {
            $\text{innerParameter}$; // one $ is enough
            $\text{outerParameter}$; // one $ is enough
        };
    }
}
```

The semantics of meta-function execution, that resolves tree-literals primarily in the invocation context, makes the definition context less accessible and requires the use of a multi-$-expression. This is not adequate for all forms of access. For instance in the above example, definition as $X::f$ rather than $\text{class X ( ... f )}$; prevents access to $X$ using a built-in meta-variable. If the meta-function is invoked in a derived class, neither form of definition supports access to the derived definition class name. Two built-in formal parameters are therefore supplied to circumvent the problems.

The Static built-in formal parameter resolves to the static definition scope of the meta-function.

The Dynamic built-in formal parameter resolves to the dynamic definition scope of the meta-function.
auto class MetaBase
{
    auto static declaration static_mf()
    {
        public static char *static_dynamic = "\$Dynamic;"
        public static char *static_static = "\$Static;"
        public static char *static_scope = "\$Scope;"
    }
    auto declaration nonstatic_mf()
    {
        public static char *nonstatic_dynamic = "\$Dynamic;"
        public static char *nonstatic_static = "\$Static;"
        public static char *nonstatic_scope = "\$Scope;"
    }
};

auto class DerivedMetaBase : auto MetaBase{};

class Invoking
{
    DerivedMetaBase::static_mf();
    DerivedMetaBase::nonstatic_mf();
};

Within the invocation of DerivedMetaBase::static_mf, $Dynamic resolves to MetaBase, whereas within the invocation of DerivedMetaBase::nonstatic_mf, $Dynamic resolves to DerivedMetaBase. In both cases $Static resolves to MetaBase and $Scope to Invoking.

Formal template parameters are treated as formal parameters supporting use of the definition parameterisation in the invocation context.

These additional built-in parameters may hide names in the invocation context, necessitating the use of $This.$Static to resolve Static in the invocation context rather than as a built-in formal.

4.3.12 Meta-function and substitution semantics

Declarations within meta-function bodies are analysed in three contexts

- when the meta-function is defined
  - to resolve (multi-)$-expressions that use formal parameters
  - to resolve multi-$-expressions in the definition context
- each time the meta-function is invoked
  - to resolve residual $-expressions in the invocation context
- when any generated declarations are compiled
  - to resolve $-expressions in the actual context

Formal parameters may hide declarations in nested classes, and may be hidden by formal parameters of nested meta-functions. A formal parameter occluded in this way may be accessed by using sufficient $'s to reach out to a lexical context in which the occluding formal is not visible.

auto declaration g(identifier id)
{
    auto declaration h(identifier id)
    {
        $j($$id, $id);  // j(outer-id, inner-id)
    }
}

It is an error if a $-expression reaches out to an external context where it cannot be resolved.
The usage of the formal parameter names is identified during analysis in the defining context. The formal parameter names are therefore not visible while resolving $-expressions in the invocation context.

```cpp
auto declaration k(identifier aParameter) {
  ${nestedInvocation};
  ${nestedDefinition};
}
```

Therefore, if in the above example $nestedInvocation has the value aParameter, the subsequent access to aParameter is resolved in the invocation context, ignoring the formal parameter. Furthermore, since $nestedDefinition reaches out to the external context, it is resolved in the defining context and so if nestedDefinition has the value aParameter, the $-expression is resolved to the formal parameter.

Upon invocation of a meta-function, all accesses to formal parameters within the meta-function body are replaced by their corresponding actual arguments before the entire body and residual $-expressions are interpreted as part of the invocation context.

### 4.4 Composition

The C++ One Definition Rule (§3.2) permits only one appearance of each declaration, and requires the declaration to have the same meaning in all compilation sessions in which it is used. A few exceptions to the rule permit repeated forward references and typedefs.

Violations of the rule within a single compilation should be trapped by the compiler. Violations between compilation sessions may go undetected and lead to unpredictable program behaviour. Some of these inter-session violations are diagnosed by the practice of using name mangling for function declarations.

Most opportunities for inter-session violations are eliminated by the practice of placing interfaces within include files that are shared between compilation sessions. As a result a program malfunction due to violation of the ODR arises mainly through the undue enthusiasm of some incremental compilers.

The ODR in combination with the hierarchical nature of C++ class declarations prevents declarations from more than one class being interleaved. This is a severe impediment to the implementation of patterns or Aspect Oriented Programming and so FOG relaxes the ODR with respect to FOG source code, requiring only that it be possible to satisfy the ODR after translation has been completed.

Multiple declarations are permitted and the contributions from each are combined hierarchically to form composite declarations. If the contributions are incompatible, the inconsistency is diagnosed and the resulting behaviour is unpredictable.

Composition is performed for the actual declarations of classes (including namespaces, structs and unions), enums, arrays, variables, functions, meta-variables and meta-functions.

When any potential declaration is associated with a determined context, an existing actual declaration is first located. If one is found, the new potential declaration is composed with the existing actual declaration. When searching for such a declaration, the search for the declared name is restricted to the specified scope, which is located conventionally.

Thus a declaration for A::B::E creates E within the scope visible as B within the scope visible as A. It may compose within an existing E. It hides, rather than composes with, an E that is visible in (by inheritance) but not part of A::B. This provides consistency with the single definition in C++.
This implies that \( A::B::A::f \) is legal and probably refers to \( A::f \) since the more nested \( A \) is visible as the less nested \( A \). The unlikely alternative with an occluding nested class is demonstrated by:

```cpp
class A;
public class A::B;
private class A::B::A;
```

in which case \( A::B::A::f \) and \( A::f \) would be distinct entities.

Names must be made visible before they are used. In particular scopes must be forward declared as in the cascade of declarations leading to \( A::B::A \). Direct declaration of a nested scope without its less nested scopes could be interpreted as an implied forward declaration, but it leaves \textit{class/struct/union} distinctions and nested access constraints unclear. It also requires assumptions to be made about the nature of the intervening names. These assumptions may be invalidated by typedefs or additional base classes.

### 4.4.1 Class composition

Most class composition occurs hierarchically: the class grows to accommodate distinct member declarations, or to compose repeated member declarations.

#### 4.4.1.1 Nested contexts

Nested class-specifiers, namespace-declarations and linkage-specifications compose hierarchically.

Compound declarations are not nested contexts. They are just a syntactical grouping of multiple declarations as a single declaration. Declarations within a compound-declaration are therefore composed individually ignoring any compound structure.

Nested filespaces, described in Appendix F.4.4, are not nested contexts. The filespace just associates file-placement with its enclosed declarations. Declarations within a filespace are therefore composed without reference to the filespace beyond retention of their required placement. It is an error for composed declarations to have conflicting placement requirements.

#### 4.4.1.2 Base classes

A base-specifier-list comprises an ordered list of base-specifiers. Composition occurs as one or more lists of potential base-specifiers (each being a base_specifier meta-object) are transformed into the single composed list of actual base-specifier declarations (each being a base meta-object). Additional potential base-specifiers whose class-name is already on the list compose with the existing actual. Additional potential base-specifiers for new class-names are converted to actuals and appended to the list.

Composition of a base-specifier involves merging the access-specifier and the virtual keywords. An error arises if a conflict arises such as private with protected, or virtual with \texttt{virtual}. A conflict does not arise when merging a specified access with an unspecified access or virtual with unspecified virtual.

```cpp
class A : B, C, B, \texttt{virtual} D {};
class A : E, protected B, F, private D, public E {};
```

composes to:

```cpp
class A : protected B, C, private \texttt{virtual} D, public E, F {};
```

The use of unspecified access is deprecated in C++ and defaults to \texttt{private}. This remains the case in FOG, but only after composition, which may provide a specified access.
4.4.1.3 Miscellaneous declarations

**accessibility-specifier**

An *accessibility-specifier* changes defaults for subsequent declarations, within the prevailing context. An *accessibility-specifier* does not affect and is not composed with other contributions to the same context.

**meta-expressions and meta-control-declarations**

Meta-programs are interpreted directly and so there is nothing to compose.

**include-declaration**

An *include-declaration* is interpreted directly and so there is nothing to compose.

**syntax-macro-definition** (Section 4.7)

Syntax macros with matching argument lists compose in the same way as meta-functions. Syntax macros with distinct argument list are overloaded and so do not compose.

**using-directive**

**namespace-alias-definition**

**file-dependency-declaration** (see Appendix F.4.5)

**explicit-instantiation**

**explicit-specialization**

Multiple declarations are gathered together. Duplicates are eliminated.

**asm-definition**

Multiple declarations are gathered together. Duplicates are preserved.

**file-placement-declaration** (see Appendix F.4.3)

Multiple declarations are gathered together. Only one distinct location may be specified for each interface and implementation file.

4.4.2 Object statement composition

Object-statement-scopes provide a limited form of scope at meta-compile time for functions and variables. Composition of declarations proceeds in the same way as composition for equivalent concepts in classes.

The presence of derivation rules supports multiple bodies in a potential declaration. Composition of these bodies is deferred until the compile compilation stage at which point only those bodies that are enabled by derivation predicates are retained.

4.4.3 Enum composition

In C++, an *enum-specifier* comprises a list of *enumerator-definitions* each of which is a name-value pair. The value part may be omitted, in which case it assumes the value zero for the first enumerator, or the preceding value plus 1 for subsequent values.
enum Enums
{
    ZERO, // Implicitly 0
    TWO = 2, // Explicitly 2
    THREE // Implicitly TWO + 1
};

In FOG, composition occurs as one or more potential declarations comprising meta-objects of enum_specifier meta-type and lists of meta-objects of enumerator_definition meta-type are converted to an actual declaration of enum meta-type and an actual list of enumerator meta-type.

As each additional potential enum-specifier is composed, each additional potential enumerator-definition is appended to the list of actual enumerators. The missing enumerator value is resolved as zero for the first enumerator or one plus the value of the most recent addition to the list for all subsequent values.

enum Enums // Composing with above
{
    FOUR, // Implicitly THREE + 1
    FIVE, // Implicitly FOUR + 1
    TWO = 2, // Explicitly 2
    THREE, // Implicitly TWO + 1
    ONE = TWO - 1 // Explicitly TWO - 1
};

Enumerator names may be repeated provided the value associated with the enumerator is the same for each repetition. Repeated names are discarded so that a meta-program traversal of the enumerators and the final code emission iterates over the domain

    { ZERO = 0, TWO = 2, THREE = 3, FOUR = 4, FIVE = 5, ONE = 1 }

The "same value" involves a direct comparison of numeric values, whose evaluation may use the already resolved enumerators. There is no support for deferred evaluation and comparison of Abstract Syntax Trees.

### 4.4.4 Construct composition

Composition of constructs first identifies the construct (function, typedef or variable) to be composed using its unique signature, and then composes the remaining parts of the construct.

The unique signature of a typedef or variable involves the scope, template arguments and the name of the typedef or variable.

```cpp
static const int X<A>::v = 5; // Signature ::X<A>::v
typedef int (*PFunc)() const; // Signature ::PFunc
```

The unique signature of a function additionally involves the function arguments and those cv-qualifiers that resolve overloads.

```cpp
virtual inline int f(const size_t& p = 5, const int q) volatile // Signature ::f(const unsigned int &, int) volatile
```

The unique signature does not include other decl-specifiers such as static, virtual or inline, the type-specifier-seq, or function parameter names or default function arguments. Type names using typedefs are resolved. Redundant cv-qualifiers for by-value arguments are ignored (§13.1-3).

The type-specifier-seq, e.g. function return type, is not part of the unique signature, since overloading of functions (or variables or typedefs) on the type is not

---

4. Exclusion of static from the signature is possible since the standard specifically excludes overloading static and non-static member functions (§13.1-2).
supported. When multiple contributions are composed, each type-specifier-seq must refer to the same type to avoid an error.

The friend, typedef and using decl-specifiers are not composed. friend and typedef distinguish between different categories of construct. They must be present for each declaration of a name.

using, in the context of the re-using-declaration described in Section 3.1.4.4, indicates that only part of the declaration is provided, and that the remainder is obtained from the re-used declarations. Composition of using in the context of a using-directive has been described in Section 4.4.1.3.

The private, protected and public access-specifiers are composed using a four-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. Conflicting explicit values are an error.

A missing value for a new declaration specified within class braces automatically acquires the prevailing access for declarations in its scope. Repeated declarations within class braces, or any declaration outside class braces retain unspecified access until composed with a declaration that has a defined access.

It is an error for a declaration forming part of a class to have no access-specifier from any source.

access-specifiers for declarations forming part of a namespace-body or linkage-specification are discarded.

The static and !static decl-specifiers are composed using a three-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. Conflicting explicit values are an error.

The explicit, export, extern, mutable and register decl-specifiers are composed using a two-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. There are no conflicting values to cause an error.

4.4.5 Value composition

Values may need to be composed as initializers for variables, dimensions of arrays, or default initializers for function arguments. Successful composition verifies that all alternative contributions have the same or no value, with the specific value being used rather than the no value. In the case of trees of values this policy is applied recursively to build a composite tree, by overlaying the lists at each level to form the longer list with consistent values. At any stage composition of a value with a different value or list of values is an error.

Conflict is determined by a syntax tree comparison after any tree-literals and constant values have been resolved. No symbolic interpretation of the syntax tree is performed, and so a redundant pair of parentheses or the interchange of binary operands is sufficient to cause a composition failure.

4.4.6 Variable composition

Declarations of (non-array) variables are composed by composing the residual decl-specifiers as outlined above in Section 4.4.4, and initializers as described in Section 4.4.5. Any conflict is an error.

4.4.7 Array composition

Declarations of array variables are composed by composing the residual decl-specifiers as outlined above in Section 4.4.4, and dimensions as described in Section 4.4.5. Composition of array initializers is performed by filling up a multi-
dimensional array from \([0][0]\ldots\), in the same way that a normal C++ initializer defines its initializers. Each composition continues where the previous one left off; it does not restart from \([0][0]\ldots\).

The gcc [Stallman98] indexed initializer syntax (Section 3.1.4.2) may be used to define a specific placement for an initialization value. As a result that it, and subsequent values in adjacent locations, may provide multiple initializers for the same array index. Those are composed provided the values are consistent as described in Section 4.4.5. Missing initializers are given a zero value when the equivalent C++ declaration is emitted for the composite initializer.

The default incremental composition is most useful within the idiomatic loop

```cpp
class X
{
  public static const char *variableNames[];
  auto for (iterator p = $variables(); p; ++p)
    using variableNames = ( "\$p->name() " );
    using variableNames = ( 0 );
};
```

The first line fully declares an array variable but without a dimension or any initializers. The loop iterates over the member variables of class X. The body of the loop contains a re-using-declaration to re-use the array declaration with a single initializer comprising the concatenation of the empty string and the member variable name. Each iteration composes the additional initializer with the existing declaration, so that the array gradually builds up a list of strings. Finally the last line adds a null terminator to the list.

Re-use of the declaration avoids a potentially conflicting redefinition, but is not readily recognisable to C++ programmers. The loop can be specified more explicitly, saving a line:

```cpp
auto for (iterator p = $variables(); p; ++p)
  public static const char *variableNames[] = { "\$p->name() " };
public static const char *variableNames[] = { 0 };
```

Indexed initializers are useful for applications such as automatically creating an array of debug text strings from an enumerator.

```cpp
auto declaration EnumTextArray(name textsName, enum enumDecl)
{
  auto for (iterator p = $enumDecl.enumerators(); p; ++p)
    using $textsName = { [$p->value] "\$p->name() " };
}
```

```cpp
class EnumClass
{
  public enum Enums { A, E = 4, F, B = 1 };
};
static const char *enum_names[] = { "A", "B", 0, 0, "E", "F" };
```

### 4.4.8 Function composition

Function composition involves composition of residual `decl-specifiers`, parameter names, default arguments and function bodies. Composition of non-function-specifier
decl-specifiers proceeds as above. The composition policies specific to functions are:

The inline, !inline, inline/interface or inline/implementa
dation decl-specifiers are composed using a five-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. Conflicting explicit values are an error, except that composition of plain inline with the more specific inline/interface or inline/implementa composition composes to preserve the more specific value.

The virtual, !virtual and virtual/pure decl-specifiers are composed using a four-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. Conflicting explicit values are an error. For the above algorithm virtual/pure may be explicitly specified as virtual/pure or by use of virtual and a subsequent = 0 pure-specifier. The existence of a corresponding virtual function in a base class is ignored during composition, however during the compilation phase the virtual attribute is propagated to derived functions, resulting in an error message for a conflicting requirement for a !virtual function derived from a virtual function, and a warning message for a virtual function derived from a non-virtual function.

Function parameter types do not compose, since distinct types represent distinct overloaded functions.

Function parameter names compose using a two-valued algorithm. Matching names compose to preserve the name. An omitted name composes with an explicit name to preserve the explicit name. Conflicting names are currently an error. This is a necessary constraint imposed by the potential for re-use of parameter names by multiple function bodies or derived functions. A less restrictive implementation should tolerate a local respecification of parameter names in function bodies and automatic renaming to those in the first declaration.

Default function arguments compose as described in Section 4.4.5.

Constructor initializers compose by gathering all initializers together. Multiple initializations for the same member variable compose as described in Section 4.4.5, however since the initializers form part of the function body, they are guarded by derivation predicates. Disabled initializations are ignored. When the constructor is emitted, the constructor initializers are emitted in constructor initialization order using explicit values, where available, with implicit values from member variable initializers as defaults.

Exceptions

Composition of exception-specifications and function-try-blocks goes well beyond the considered policies for FOG, and so is an area for further work. It would appear that exception-specifications for a particular function should just be gathered together with duplicates discarded. It seems that exception-specifications should propagate up the inheritance hierarchy to extend non-empty exception-specifications, so that the exception-specification of a derived virtual function should never be wider than its inherited exception-specification. This may lead to errors when inheriting from library classes, whose specifications cannot be changed, but these errors diagnose a problem rather than imposing a restriction. Function-try-blocks should probably just be concatenated on a per-handled type basis, with handlers organised to ensure handlers for more derived types precede those for less derived types. It may be necessary to extend the syntax for a default member-variable initializer to wrap a try-block around it.

Body

The overall function body is formed from the concatenation of contributions to five named segments within which contributions are independently composed.
return-type function-name(function-arguments) cv-qualifiers  
exception-specification
{
  entry-segment-contribution
pre-segment-contribution
body-segment-contribution
post-segment-contribution
exit-segment-contribution
}
Contributions to each segment are concatenated and by default, contributions are made to the body segment, so that

```c++
void f() { i++; }
void f() { i--; }
```
composes to

```c++
void f()
{
i++;
i--;
}
```
More explicit control of contributions requires the use of an object-statements-clause in which the extra annotation syntax does not cause conflicts. The entry and exit segments are useful for establishing a function framework:

```c++
class Manager
{
  public Manager() {}  
  public bool do_it()
  {
    entry { bool exitStatus = true; };
    exit { return exitStatus; };
  }
};
```
Other sources modules may contribute code. An independent module could specify:

```c++
class Manager
{
  private MyContext& _context = MyContext::make(*this);
  using do_it { if (!_context.do_it()) exitStatus = false; }
};
```
Composition of the framework established by entry and exit segments together with the extra body results in the composed function:

```c++
bool Manager::do_it()
{
  bool exitStatus = true;
  if (!_context.do_it())
    exitStatus = false;
  return exitStatus;
}
```
The framework can be extended by any number of independent contributions. The safety of such extensions is dependent upon their orthogonality. If the contributions interact, then the meta-compilation source files must be carefully sequenced to ensure the intended result.

As well as extending the do_it function, an additional member variable was specified with a default initializer. The composed class therefore looks like:
class Manager
{
  private:
    MyContext& _context;
  public:
    Manager();
    bool do_it();
};

The default initializer is automatically supplied for all non-copy constructors:
Manager::Manager()
  _context(MyContext::make(*this))

The five segments provide sufficient flexibility for many applications, but are not
adequate for all. For instance, wrapping an if around a function body:
void f()
{
  pre { if(...) { } // Not valid in FOG
  post { } } // Not valid in FOG
};

requires partial syntax that violates the syntactical requirement for statements.
There is also a parsing ambiguity between { and } as partial syntax and as syntax
structuring. (An earlier version of FOG used \{ and \} for partial syntax and
allowed any list of tokens as a composed contribution.)

The structure of a function body is defined with two levels of indirection through
meta-variables:

auto statement function::value[] = { @function_structure };
auto static const statement token::function_structure[] =
  { @entry; @pre; @body; @post; @exit; };
auto statement function::entry[] = {};
auto statement function::pre[] = {};
auto statement function::body[] = {};
auto statement function::post[] = {};
auto statement function::exit[] = {};

The segment-name/body syntax
derived(is_root{)) entry { bool exitStatus = false; }
is therefore syntactic sugar for:
derived(is_root{)) auto entry += { bool exitStatus = false; }
in which the internal meta-expression-statement of the form
auto list-name += tree-statement
is used to append the requisite code to the list using the built-in operator+= which
is valid for lists.
An if may therefore be wrapped around the existing body without violating syntax
constraints by:
void f()
{
  auto body = { if(...) @body; };
};

Braces are optional around @body since it is treated as a single statement. The
use of @ rather than $ is very important to defer evaluation until the compilation
phase. A $ would be evaluated when the potential declaration is associated with
an actual function, allowing further potential declarations or meta-programs to
extend the body following, rather than within, the if.
The global structure of functions cannot be changed by redefining `token::function_structure`, since this is a `const` meta-variable.

The structure of an individual function can be changed by defining `function_structure` within an `object-statements-clause`. Code segments may be added and removed, but the syntactic sugar for the five built-in segments is unaffected; user-defined segments can only be extended using an expression operator such as `+=`.

Statements can be appended to a segment using `+=` or replaced by just using `=`, as in the examples above.

Declaration of a meta-variable as `const` precludes multiple assignments, and so may be used to provide some protection against unforeseen activities by interfering meta-programs. Thus

```cpp
auto const statement body[] = {};  
```

(re)defines the body segment to be empty, detects an error if the existing content is anything other than the built-in empty default, and detects an error when any subsequent attempt is made to change the content.

As semantic analysis and meta-programming proceeds, the meta-objects for each function build a list of object-statements with their associated derivation predicates. $$\varphi$$-expressions in these object-statements remain unevaluated.

During the compilation stage, classes are compiled in least derived first order, and individual members of each class are compiled in an unspecified order. The list of object-statements is scanned in the order in which potential declarations contributed to actual declarations, with the further inherited lists of object-statements applied in destruction order: a base-class can therefore wrap code around its derived implementations predictably.

**Historical Note**

The multi-pass implementation of FOG was able to, but did not, parse function bodies which were therefore treated as an arbitrary list of brace-delimited tokens. Code composition was supported by arbitrary concatenation of such lists giving total flexibility and anarchy since there was no syntactical constraint upon the composition, and no discovery of error until a subsequent compilation failed.

The superset implementation parses function bodies and imposes syntactical consistency for each composition. These constraints provide much needed integrity and respond to some of the hazards outlined by [Ossher98] and impose some of the discipline discussed by [Mulet95].

### 4.4.9 Meta-variable composition

Meta-variables are working variables for use at meta-compile-time. Any redeclaration of a meta-variable must therefore have the same meta-type and current value, which in the case of a `const` meta-variable must be the initial value.

Redeclaration of a `const` meta-variable with the same value is permitted, although reassignment of a `const` meta-variable with the same value is an error.

### 4.4.10 Meta-function composition

Meta-functions compose in the same way as functions. By default, contributions are gathered into the `body` list of declarations. When a meta-function is invoked, the body returned for incorporation in the invoking context comprises the hierarchical composition of segment contributions with the formal parameters replaced by actual arguments.

Declaration of a `const` meta-function can be used to inhibit a redefinition:
auto const declaration f()

:\{
    auto const statement body[] = { /* ... */ };
\};

4.5 Meta-classes

Conventional programming involves programs that operate on application entities. Meta-programming involves programs that operate on program entities: the class, function and variable declarations that define a program.

In the same way that newcomers to Object-Oriented are easily confused by loose usage of the terms class, instance and object, newcomers to meta-programming are easily confused by loose usage of the term meta-class. One confusion arises because, in Object-Oriented Programming, the phrase is-a denotes an inheritance relationship, but in meta-programming is-a can alternatively be used to denote an instantiation relationship. Further confusion arises because the object models available to the programmer do not correspond to the underlying abstraction.

The exact one to one relationship between classes and meta-classes in FOG is easy for the programmer to appreciate and natural to use, but apparently in conflict with some of the more traditional perspectives of meta-classes and meta-meta-classes. We will therefore describe the very pure object model exemplified by ObjVlisp [Cointe87], before describing the C++ model and the enhancements provided by FOG in order to justify the FOG model.

A very simple three-class hierarchy is shown in the central column of Figure 4.9 in which class X inherits from class Y, and class Y inherits from class Z. Three Instances of each class are shown stacked to its right with the top instance named respectively anX, aY and aZ.

Figure 4.9 Pure OO object model

An Object Oriented program performs computation as a result of the interaction of its object instances at run-time, and in the simplest object model, instances have meaning only as object instances at run-time; classes exist solely as an abstraction at compile time.

A more sophisticated object model enables the run-time objects to make use of class information, and in a pure object model, this information is provided by a run-time object for each class. Each such class object provides a description of the instances of its class.

Every object must be an instance of some class, so it is necessary to define meta-classes that are instantiated as the class objects. The meta-classes are labelled...
_X, _Y and _Z in the figure. (If the same description applies to each of X, Y and Z, then _Z alone is sufficient to describe all classes.)

The pure object model requires a run-time object to describe each of these meta-classes, and so there are corresponding meta-class objects to describe each class. A potentially infinite recursion is avoided by ensuring that the instance of _Z is also a valid description of _Z. Every box in Figure 4.9 corresponds to a run-time object; vertical arrows denote an inheritance relationship; horizontal arrows denote an instance-of or described-by relationship from right to left or a describes relationship from left to right.

The pure object model just described is that of ObjVlisp [Cointe87]. Few other languages comply to it. Smalltalk provides a similar but more restrictive model; the meta-classes of meta-classes are not visible [Briot89]. CLOS and SOM provide considerably more generality. The inheritance relationships of meta-classes and classes need not correspond. This leads to a number of significant compatibility issues [Graube89], whose resolution seems to create further problems [Danforth94], [Forman94].

Meta-classes were originally introduced for languages such as CLOS to assist in the construction of instances whose layout was entirely defined at run-time. More efficient languages such as C++ or Eiffel define object layouts at compile-time and compiler writers have no need to provide meta-classes. The available facilities in C++ are limited.

The C++ components corresponding to Figure 4.9 are shown in Figure 4.10. The instances in the right hand column comprise

- a contiguous piece of memory for the member variables
- a hidden pointer to the instance description often called vptr, which has been arbitrarily labelled -rtti-

Each class (and meta-class) 'object' comprises a potentially contiguous area of memory containing the compiler generated run-time type information as an instance of std::type_info, and generally discontiguous areas of memory for the static member variables and functions.

Most of the behaviour of std::type_info is implementation defined, but it will typically comprise

- a pointer to the class description (-rtti-)
- a list of base-class instance descriptions (-bases-)
- a class name
- a dispatch table for virtual functions (-vtable-).

Very limited functionality is exposed for the std::type_info class, whose sole instance both describes and is an instance of the std::type_info class. This is the only meta-class in ISO C++, although a more substantive facility was proposed by [Buschmann92] during the standardisation process.

The class object of any suitable type T is returned by the typeid(T) operator. The solitary meta-class object is returned by typeid(typeid(T)) or just typeid(std::type_info).

Class variables

The symmetry of Figure 4.9 and the presentation in Figure 4.10 implies that the class variables (static member variables) form a logical part of their corresponding class object. This follows postulate 6 of [Cointe87] that the class variables of an instance are the instance variables of its meta-class. However, in common with many programming languages, C++ does not require the programmer to distinguish between use of class variables and instance variables. This programming convenience leads to some confusion between class and instance
Figure 4.10  C++ object model (Memory Perspective)
means, and undermines the pure perspective of the meta-class structure. The class variables are logically shared parts of each instance. The alternative detailed presentation of Figure 4.11 and its simplified presentation in Figure 4.12 is more appropriate. All visible names now appear in one rather than two columns. At the source level, a member of a class or instance is accessed by name. The access is resolved by consulting the instance description to convert the name into the address of a variable or function. In C++, this conversion process is almost
entirely performed at compile-time; those few conversions that cannot be completely resolved at compile-time are partially resolved as fixed indexes that index the \texttt{-vtable-} at run-time. The \texttt{-vtable-} is all that remains of the more general name to address conversion table required by less compiled languages.

Seen from this perspective, class variables and instance variables differ in their access policy, but share the same name-space. Each class variable exists with a 1:1 relationship to a class, and is accessible by name from that class, and derived classes. However, there is no reason for class variables to be grouped as an object, and in C++ they are not. It is in fact impossible to group class variables as a contiguous object, since shared class variables cannot be adjacent for more than two distinct derived classes.

When class variables are accessible with respect to instances, there is no reason for them to be accessible with respect to their classes as well, and again in C++ they are not. However, the symmetrical interpretation, in which class variables are the instance variables of their meta-classes, requires that they are accessible with respect to instances and classes and so blurs the distinction between meta-levels [Maes87].

Therefore names of class variables (and functions) are visible in the instance description (the conventional name-space for access with respect to instances), but are omitted from the class description (the meta-name-space for access relative to instance meta-objects (classes)).

**Compile-time object model**

The run-time object model of Figure 4.11 is redrawn without the clutter and without inheritance in Figure 4.12.

![Figure 4.12 C++ Run-time object model](image)

Programming normally operates using the multiple instance objects of each class shown in the right-hand column. Static and non-static member functions and variables share the same name-space. The hidden \texttt{-rtti-} pointer identifies the run-time type information description of the instance. Invocation of \texttt{typeid()}  

5. An instance description is a description of the names visible in an instance (on Figure 4.11 and Figure 4.12. A "class description" misleadingly refers to the wrong meta-level.)
shifts the programming perspective to the meta-level, where only the members of the `typeid` instance describing the class are visible. A further invocation of `typeid()` shifts to the meta-meta-level and since `typeid` is its own meta-class, only its members remain visible.

The corresponding compile-time programming model for FOG is shown in Figure 4.13.

A meta-level instance at compile-time (Figure 4.13) has instance variables and functions, as well as class variables and functions in just the same way as an instance at run-time (Figure 4.12) has instance (member) variables and functions and class (static member) variables and functions. The class meta-object members are referred to as (non-static) meta-members and static meta-members by direct analogy with (non-static) members and static members.

Each class declaration forms part of a class meta-object that also comprises meta-variables and a description of the run-time instance objects. The very right-hand 'level' corresponds to the normal run-time perspective, in which a small amount of the instance description maintained by the class meta-object is provided as the run-time-type-information of the class object. This level does not exist at compile-time.

Meta-programming occurs at the meta-level where the description of objects (member names and types) is available as well as the meta-members of the meta-class. The `-ctti-` counterpart of `-rtti-` identifies the describing meta-object. The class description describing an instance meta-object contains the full mapping from names to built-in and user-defined meta-functions and meta-variables, rather than the highly optimised `-vtable-` in the run-time type information.

Instance meta-objects are instances of `class` or `class_specifier` meta-types. Class and meta-class meta-objects are instances of the `meta_type` meta-type.

In the same way that `typeid()` shifts the perspective to the left in the run-time diagram, the `meta_type()` built-in member function shifts the compile-time
perspective to the left. Whereas the typeid of a run-time class is type_info, the meta-type of a compile-time meta-object is meta_type.

The meta-class

The compile-time representation of a class comprises two meta-objects, one class meta-object to describe the class, and one instance meta-object to describe run-time instances. Since these two meta-objects have matching inheritance and always exist as a pair, it is convenient to regard the pair of meta-objects as a single object, which may be safely but loosely referred to as the meta-class.

Other meta-types

Figure 4.13 is drawn for the meta-objects describing classes and their instances. The same diagram applies for all meta-objects, replacing ‘instance’ by the meta-object category. However, only variables, functions, meta-variables and meta-functions have their own user-defined meta-functions and meta-variables.

A function meta-object, of meta-type function or function_specifier, contains a function description (the parameters). The corresponding function meta-meta-object contains the mapping of all meta-names applicable to the function including any meta-variables defined within an object-statements-clause. The function body is indirectly defined via the value meta-variable.

Similarly, but taking care to use distinct terminology, a meta-function meta-object, of meta-type meta_function or meta_function_specifier, contains a meta-function description (the parameters). The corresponding meta-function meta-meta-object contains the mapping of all meta-names applicable to the meta-function including any meta-variables defined within an object-statement-clause. The meta-function body is indirectly defined via the value meta-variable.

The distinction between meta-functions and function meta-objects is shown in Figure 4.8 on page 104.

Meta-inheritance

C++ supports instance objects and, to a limited extent, class objects at run-time. FOG extends C++ to support class objects and, to a limited extent, meta-class objects at compile time. FOG provides a meta-class for every class and built-in type. The inheritance of meta-classes mirrors that of the class hierarchy, so that the base meta-classes of every meta-class are the meta-classes of the base classes of the corresponding class. Every meta-class without any other base meta-classes automatically inherits (virtually) from the built-in meta-class ::auto. Additional meta-inheritance may be specified by using the auto keyword as an access-specifier. The inheritance and meta-inheritance for
class Base {};
class Derived : public Base, auto char {};
is shown in Figure 4.14. With the exception of meta-inheritance from ::auto which is always virtual, meta-inheritance is defined by the virtual keyword. A virtual meta-class appearing more than once in the meta-inheritance hierarchy only contributes one set of meta-variables.

4.6 Meta-programming

The description of compilation stages in Section 4.1 identified the different contexts in which meta-programming can occur.

A parasitic form of meta-programming occurs through augmented behaviour during semantic analysis and compilation as tree-literals are resolved. However, even allowing for the extra flexibility provided by derivation rules, this behaviour constrains the actions of meta-programming to be closely correlated with the
corresponding timing of compiler activity. This is adequate for simple elaboration of declarations, but a poor foundation for a meta-program where the programmer, rather than the meta-compilation system should determine the sequencing.

FOG therefore provides additional compilation stages in which the programmer has greater control. The meta-construction provides an opportunity for algorithms that operate on meta-classes to be activated in a base-functionality first order analogous to construction, with meta-destruction providing a similar opportunity in base-functionality last order analogous to destruction. In addition the meta-main stage provides the programmer with no sequencing assistance or constraints.

**meta-construction**

During the meta-construction stage, the meta-constructor of each class (including built-in types) is invoked once in a least-derived first order. Programs may supply code for execution during this stage by declaring a meta-constructor:

```c++
auto MyClass::MyClass()
{
    /* meta-program */
}
```

This code is composed with any other definitions of the same meta-constructor. Such composition includes inherited contributions, which are executed first in accordance with the normal principles of constructors. Since everything inherits from the *auto* class, definition of a meta-constructor for *auto* provides a mechanism for executing a meta-program in all classes.

**meta-main**

The meta-main stage consists of execution of all definitions of:

```c++
auto void main()
{
    /* meta-program */
}
```

in the order in which the declarations are encountered.

**meta-destruction**

The meta-destruction stage mirrors the meta-construction stage. The meta-destructor for each class is executed once, again in least-derived first order, but
with inherited contributions executed after local contributions in accordance with destructor principles.

**Integrity**

It has been recognised [Chiba96] that the operation of two apparently independent meta-programs on the same program can lead to poorly defined behaviour. For instance, a problem can arise even with two very simple meta programs which

- add a diagnostic print-out to every function to create a call-trace
- add a check invariant() and invocation from each non-const function

Whether a diagnostic print-out is added to the check invariant() routine depends upon the application order. Whether such a print-out should be added is a subtle user preference.

The distinction is fairly trivial in this example. The distinction is critical for applications that involve synchronization or persistence, since functions or variables added by meta-program A after the class structure has been analysed by meta-program B may not be subjected to a re-analysis by meta-program B.

The problem is largely ignored in practical reflective systems. It is assumed that the meta-programmer will coordinate multiple meta-programs. The theoretical problem is addressed by the reflective tower [Smith84], in which each level\(^6\) of reflection defines a new language for the level above that hides the language of the level below. The rather impractical need for a distinct representation of each object at each level is described by [Chiba96].

The three stages offered by FOG perhaps represent a pragmatic compromise and symmetry with run-time concepts of static-construction, main-program and static-destruction. During the meta-construction stage, actual declarations are in a highly unstable state, since further meta-programming may provide additional declarations. It is therefore unwise to place any code in a meta-constructor that browses child declarations. Meta-construction code should consist only of definitions. Browsing meta-programming should be implemented in meta-destructors, and meta-programs should avoid creating new declarations during meta-destruction in order to support consistent behaviour by other meta-destructors. Meta-destructors should only elaborate and compose with existing declarations. The meta-main stage is not strictly necessary, however it avoids the need for relatively arbitrary meta-programs to be constrained by the invocation mechanisms of meta-construction or meta-destruction.

The non-trivial examples in Chapter 7 make extensive use of the meta-construction and meta-destruction phases to realise each example. It would appear that the two traversals of the tree of program declarations are insufficient to support multiple meta-programs.

It is essential for all of one meta-program to execute before any of the next, since use of just the meta-construction and meta-destruction phases requires interleaved meta-program execution as the inheritance hierarchy is descended.

In principle, the problem of many meta-programs can be directly resolved by multiple meta-main programs, each of which perform a hierarchical traversal starting with an iteration over ::all_classes(). However this requires the source declarations to be presented in the correct order. A slightly less direct approach could support registration of activities as one or more lists of meta-functions during meta-construction. These lists could then be serviced by meta-

---

6. A level of reflection, counting the number of layers of language elaboration provided by meta-programming, should not be confused with a meta-level, counting the number of levels of description for instance, meta-instance/class, meta-meta-instance/meta-class, ...
main. This indirection provides some opportunity for programmed prioritising, rather than source file sequencing, to determine the behaviour.

Identifying a more direct mechanism for specifying multiple meta-programs and their sequencing dependencies is an area for further work. This could tie in to consideration of user-defined meta-types, for which the current use of meta-constructors and meta-destructors for compilation stages rather than meta-type maintenance could be embarrassing.

4.7 Syntax macros

The superset parsing approach described in Chapter 5 isolates the syntactic and semantic analysis stages. This supports an implementation in which all syntactic processing completes before any semantic processing starts. However, enforcing this isolation prohibits any syntax dependency on semantics, and unfortunately prevents the definition of syntax macros, since resolving the definition of a syntax macro is a semantic activity.

It is important to distinguish the C++ syntax macro problem from the equivalent Lisp problem. Lisp has a very disciplined lexical structure, in which the program tree structure is represented by parentheses in the source. It is therefore easy for a Lisp preprocessor to manipulate its syntax trees, since they can easily be identified. There is no equivalent underlying lexical structure in C++, not even \{ \} are predictable,

\[
\begin{align*}
\text{namespace } X \{ \} & \quad // \text{completes a } \{ \} \text{ construct} \\
\text{struct } X \{ \} a; & \quad // \text{; completes a } \{ \} \text{ construct} \\
\text{if (a) } \{ \} & \quad // \text{; sometimes completes a } \{ \} \\
\text{if (a) } \{ \text{ else if (b); else; } \} & \quad // \text{; construct except when ...} \\
\text{do } \{ \text{ while(); } \} & \quad // \text{; may be mid-construct} \\
\text{for ( ; ; ) } \{ \} & \quad // \text{; may be in a strange place}
\end{align*}
\]

Identifying a C++ construct cannot be reliably performed by a simple preprocessor, unless that preprocessor has such a substantial understanding of C++ syntax, and indeed semantics, that it cannot really be regarded as a simple preprocessor. Section 5.4 describes how early attempts to implement FOG in preprocessor style failed through lack of adequate language comprehension.

In Lisp, a syntax macro can be defined and exploited by a preprocessor. In C++, definition and exploitation of the syntax macro must be integrated with the language.

A syntax macro supports (or rather gives the illusion of supporting) user-defined language extensions\(^7\). Such an extension could in its general form support arbitrary additional syntax so that:

\[
\text{with counter from 1 to 100 step 5 in } \{ \text{ sum += } $\text{counter; } \}
\]

could be recognised as an alternate way of writing

\[
\text{for (int counter = 1; counter < 100; counter += 5)} \{ \text{ sum += } $\text{counter; } \}
\]

More practically syntax macros should support what appear to be extra \textit{decl-specifiers}

\[
\textit{synchronized} \text{ class MyClass} \\
\{ \\
\textit{persistent} \text{ int } _\text{count}; \\
\}
\]

so that programmers appear to use an extended language, although the \textit{synchronized} and \textit{persistent} extensions are realised by meta-programming in

\(^7\) The extensions are shown by italicizing the type-writer font.
a standard language. The example is clearly more readable than the functionally equivalent:

```
synchronized(class MyClass
{
  synchronized(int _count);
});
```

In both cases application meta-functions are invoked to support the concepts of multi-process synchronisation or data-base persistence. The syntax macro approach has the advantage of offering a much more acceptable programming interface.

Implementation of a syntax macro requires the trigger word (`persistent`) to be recognised in order to perform the appropriate syntactical parse.

Recognition of the trigger word in an unconstrained context requires the trigger word to become a new reserved word, introducing the problems of conflicting usage and unwanted replacement associated with the C preprocessor.

Alternatively, recognition of the trigger word within a restricted syntactical context imposes the implementation problem of executing yacc at compile-time to generate an updated syntax analyser, and the practical problem of enabling the application programmer to understand the shift-reduce conflicts associated with a proposed macro. Resolution of these conflicts is unlikely to be portable. The complexity of generating and diagnosing an updated analyser at compile-time are inappropriate for a language with as difficult a syntax as C++.

A potential solution to the problem of conflicting name capture lies in the use of the C++ name hierarchy. A syntax macro could be defined within a namespace or class and would only be a reserved word within that namespace or class. However this approach has two problems:

A reserved word does not necessarily occur lexically within its class:

```
persistent int MyClass::_status;
```

would have to be written:

```
class MyClass
{
  persistent int _status;
};
```

This is inelegant but could perhaps be tolerated.

The class may be indeterminate:

```
$do_something(class MyClass
{
  persistent int _status;
});
```

Since the context of MyClass is undetermined, it cannot be known whether `persistent` is a scope-dependent reserved word and so syntactic analysis is not possible.

This is also inelegant and could perhaps also be tolerated, but the first example indicates that scope-specific syntax macros do not extend C++ comfortably. The second example shows an incompatibility with other FOG concepts.

A syntax macro must therefore be scope-independent, and so have the same status as any other reserved word. Syntax macros should be restricted to applications where the benefit of the cleaner invocation far outweighs the hazards of the introduction of a global name.

Definition of a syntax macro should integrate with the rest of the language, and necessarily occurs at global scope. However, usage of a syntax macro requires
semantic analysis of its definition to have been completed. Therefore preserving
the implementation option of completing syntactical analysis before starting
semantic analysis requires semantic analysis of `syntax-macro-definitions` to occur
during syntax analysis. Syntax macro definitions therefore have a distinctive
syntax to facilitate this special treatment.

Although definition of a syntax macro can be regarded as a preprocessing activity,
its exploitation occurs in conjunction with the subsequent compilation activities.

```
syntax-macro-definition:
  explicit auto meta-type identifier ( syntax-macro-parameter-list_opt ) exposed-tree_opt
  compound-tree-statement

syntax-macro-parameter-list:
  syntax-macro-parameter
  syntax-macro-parameter-list , syntax-macro-parameter

syntax-macro-parameter:
  meta-type identifier exposed-tree_opt
  identifier
  reserved-word
  punctuation
```

The further overloading of `explicit` and `auto` is unpleasant and only slightly
mnemonic. Introducing a new reserved word such as `syntax` would be better.

Declaration of a syntax-macro declares the `identifier` to be a reserved trigger word
for a sentence that should satisfy the `meta-type` syntax. The syntax to be accepted
by the parser comprises this trigger word followed by the sequence of `syntax-macro-
parameters`, which comprise expected syntax elements and further words or
punctuation which are temporarily reserved between recognition of the trigger
word and detection of the end of the syntax.

It is a slightly surprising but fortunate accident that this syntax supports
specification of any combination of intervening punctuation including `,` or `)`. Thus
the syntax-macro to intercept and pack a fractional coordinate such as

```
pt(0.5, -0.7)
```

could be specified as

```
explicit auto expression ptT (
  assignment_expression8 x
  ,
  expression y
)

T (int(32768 * $x) <= 16) | (int(32768 * $y) & 0xFFFF); T
```

using an overline to distinguish tokens defining the structure of the definition from
those parameterising the definition.

The above definition makes `pt` a reserved word throughout the rest of the
program. Once the reserved word is recognised, its literal arguments are given a
temporarily reserved status. These are the punctuation comma and `,`, following `pt`
(or from, to, step, in following with in the earlier example). The temporarily
reserved status ensures that separators are treated as separators, giving a well-
defined, if not necessarily flexible behaviour. Premature recognition of the comma
as a separator in

```
pt(pow(1,2), 0)
```

will give a potentially confusing error diagnostics from arithmetic on "pow(1", even
if no earlier confusion was caused by the trailing ", 0)". Syntax macros are
perhaps best restricted to single arguments that comply with a C++ construct.

---

8. `assignment_expression` rather than `expression` is necessary to ensure that
`pt(0,0)` is not treated as `pt((0,0) "missing")`, since `0,0` is an expression.
Overloading is permitted, subject to the constraint that the set of temporarily reserved tokens is the union of all syntaxes triggered by the overloaded trigger word. In the following example with single arguments, there are no separator tokens, so there are no temporarily reserved tokens to cause confusion.

```cpp
explicit auto variable_specifier persistent(variable_specifier v) {
  $v;
  /* additional meta-programming using $v */
}

explicit auto class_specifier persistent(class_specifier c) {
  $c;
  /* additional meta-programming using $c */
}
```

Syntax-macro parameters are identified one at a time. For each parameter, a one-token lookahead is used to see whether the explicit identifier, reserved-word or punctuation requirement of a parameter can be satisfied. If satisfied, the lookahead is discarded, overload alternatives without explicit requirements are discarded and the scan continues looking for the next parameter. When no explicit requirement can be satisfied and a meta-typed parameter is required, a recursive syntactical analysis is invoked to locate a generic syntax element. Overload alternatives that the generic element satisfies are retained. Eventually:

- no alternatives remain: a syntax error has been detected
- one alternative remains and it requires no further parameters: the syntax-macro arguments have been successfully identified
- more than one alternative remains: an ambiguous invocation has been detected

The temporarily reserved words are restored to their previous status after processing of the syntax macro. This may still be a reserved status since syntax macros can be invoked recursively.

Care should be exercised in the use of { } and ; as specific punctuation since FOG uses these for recovery from syntax errors, which may be hampered by unconventional usage of these tokens.

A syntax-macro is functionally the same as a meta-function. It differs only in its invocation mechanism. A meta-function invocation has a trigger operator ($ or @) and an expression identifying the meta-function name followed by parenthesised comma-separated generic syntax elements. A syntax macro has a trigger word identifying the macro followed by a sequence of user-defined punctuation, (temporarily) reserved words and generic syntax elements.

The foregoing description has been implemented in so far as FOG is able to perform the syntactic analysis of syntax macro definitions, including the above example. Meta-functions have also been implemented, so there is a relatively small gap to bridge to activate the alternative invocation mechanism, particularly for the simple case of single argument macros. Resolution of overloads and multiple separators is not particularly difficult, however it is clear that the lack of an underlying lexical C++ structure imposes severe limits upon the useful complexity of multi-argument syntax macros. Implementing this and assessing the utility is an area for further work.

### 4.8 Summary

We have described the FOG compilation activities and the transformation of source declarations to potential declarations, at which level meta-programming and composition can operate before conversion to the actual declarations that are
emitted as C++ declarations. We have shown how meta-types act as a syntax predicate and justify the polymorphic treatment of a single meta-type with a tree of the same meta-type, supporting meta-programming over useful program structures.

Alternative models for macro substitution have been examined and the choice of syntax level substitution in FOG justified. The rationale for treatment of a tree-literal as an identifier has been given.

The C++ name-spaces have been described. The new meta-name-space has been related to the run-time name-space, with usage within auto statements or tree-expressions. A multi-$ invocation has been provided to resolve lexical scoping problems and the FUNARG problem. The need for the built-in meta-function parameters Dynamic, Static has also been given.

Composition rules have been provided to define the behaviour once the C++ One Definition Rule has been relaxed so that it applies only at the output from FOG. Some of the hazards of ill-disciplined function body composition have been addressed by the use of token lists.

The general concepts of meta-classes have been reviewed in order to establish the C++ perspective and show how FOG provides a consistent compile-time generalisation of meta-classes and other meta-types.

The need for flexible meta-programming has been motivated and solutions provided by meta-construction, meta-main and meta-destruction compilation stages.

Finally, the difficulties of implementing syntax macros in C++ have been considered, and a proposal described that works reasonably, at least for single argument macros.
5 Parsing

This chapter deals with the practical problems encountered during the
development of the parser for FOG, and the reasons for the solutions adopted.
The problems and solutions, while motivated by the needs of FOG, are almost
type of C++. This chapter therefore concentrates on the C++
perspective, so that readers only interested in the analysis and techniques for
analysis of C++ syntax and the novel C++ parsing approach may read this chapter
in isolation.

Readers particularly concerned about C++ language details may care to browse
the on-line penultimate working draft [C++96] of [C++98] at
http://www.maths.warwick.ac.uk/c++/pub/wp/html/cd2
or print off a copy of Annex A from

Overview

Traditional approaches to parsing C++ make use of semantic (type-name and
template-name) knowledge during the lexical and syntactical analysis stages. Meta-programming in FOG can involve manipulation of declarations before type
and template information is available. A traditional approach to parsing C++ will
therefore not work for FOG, so a new superset approach is necessary.

The superset approach involves identifying a larger language than C++ that can
be parsed without semantic knowledge. The traditional declaration/expression
ambiguities are unambiguous in the larger language, and so their resolution can
be deferred until the semantic analysis that performs the narrower C++ analysis.
This approach is then able to operate without type-name information

A lack of template-name information is more disruptive, and so another new
approach is required. This involves a potentially brute force search of all the
template/non-template alternative parses until a syntactically consistent parse is
found. This search is shown to incur only minor costs in practice, and a simple
implementation of the search is provided by introducing back-tracking without
modifying yacc/bison. The rare discrepancy between a syntactically consistent
and the semantically correct interpretation is able to be deferred until the C++
semantic analysis, at which point minor corrections produce the required parse.

The combination of these two new approaches enables the different parsing
stages to be isolated. Lexical analysis is performed using a very simple lex
grammar, that makes use of no syntactical or semantic knowledge. Syntax
analysis is performed with a yacc grammar that is smaller and closer to the
language standard than traditional grammars. Since the grammar does not use
semantic knowledge, it does not suffer from the problems of potentially infinite
lookahead to resolve types that normally arise. The final semantic analysis occurs
within the context of the Abstract Syntax Tree that represents the entire program,
so the analysis may be coded in a natural style, rather than within the tight
confines of parser action routines that have only limited context and must not
provoke shift-reduce conflicts.

A further innovation extends the regular expression notation so that complex
productions from the C++ grammar can be analysed and the ambiguities deduced.

Chapter Summary

The traditional technology, structure and terminology of a compiler are briefly
outlined in order to provide some grounding for readers not well-versed in compiler
fundamentals. The dragon book [Aho86] is the recognised authority.

29-June-2001
Typical approaches to parsing C++ are discussed, the choice of parsers available as a basis for the FOG parser is reviewed and then the evolution of FOG from a very simple to fully fledged parser is described.

The potential and actual ambiguities encountered by a C++ parser are analysed, by way of demonstrating an extended form of regular expression notation that supports reasoning about grammars.

The syntax generalisations of the superset grammar are described and the new notation is then used to justify the soundness of the superset parsing approach, which solves the need for type information. The use of back-tracking in bison is then described to solve the need for template information.

A few details of the code structure are provided to demonstrate the high degree of isolation between the parsing stages.

Finally some size metrics are produced to compare the new C++ parsing approaches against other approaches and estimate the extra cost of the FOG enhancements.

### 5.1 Terminology

Figure 5.1 is based on Figure 1.9 of the dragon book [Aho86]. It shows the typical components of an application that translates a source program into a target program.

Successive stages of analysis extract the meaning of the source program, enabling an intermediate representation to be built, optimised and then converted to the required target program. A symbol table maintains information to be shared between stages. The error handler supports generation of error messages in as helpful a fashion as possible.

The analysis is broken into three stages. The lexical analysis identifies and validates individual lexemes (words), the syntax analysis identifies and validates grammatical constructs (sentences), and the semantic analysis validates the meaning of each construct within a wider context.

Lexeme is the normal term for the product of lexical analysis. The same concept is more commonly referred to as token in the context of syntax analysis. Syntax
presents the greater technical challenge and so the term *token* is preferred in this thesis.

In C and C++, the preprocessor provides additional translation. It may be implemented as a separate program, or as an additional stage between lexical analysis and syntax analysis. The C++ standard treats the preprocessor as an extra stage and uses the term preprocessor token to describe a lexeme that passes from lexical analysis to preprocessor, and the term token to describe a lexeme passed from preprocessor to syntax analyser.

Lexical and syntactical analysis were once the hardest parts of a compiler to write, however with the advent of standard lexer and parser generator tools, these stages are now relatively easily automated.

The *lex* program converts a grammar specification in the form of a number of regular expressions into a state machine or DFA (Deterministic Finite Automaton) that accepts characters one by one from a source file and emits a lexeme for each analysed word. *lex* [Lesk75] is the standard tool. *flex* [Levine90] is a more polished version distributed as part of the GNU tool set.

The *yacc* program converts a grammar specification in the form of BNF rules into an LALR(1) parser that demands lexemes one by one from the lexer and invokes action routines as each parsing rule is satisfied. *yacc* [Johnson75] is the standard tool. *bison* [Levine90] is a more polished version distributed as part of the GNU tool set.

The *flex++* [Coetmeur93b] and *bison++* [Coetmeur93a] variants were used for FOG. The variants encapsulate the generated lexer or parser as a C++ class, and so readily support multiple and re-entrant lexers and parsers. FOG uses one lexer grammar, four (tiny) parser grammars for ANSI C preprocessing and one (huge) parser grammar for extended C++ parsing. Deferred character-level substitution by std::parse allows lexer and parser to be re-entered during meta-compilation.

The operation of the state machine generated by an LALR parser is extremely simple comprising just four actions for each possible next token.

- accept the token as the termination of a sentence in the grammar
- reject the token as inconsistent with any sentence of the grammar
- shift to another state, deferring any decision
- reduce following recognition of a rule

Parser generators allow application code to be supplied for execution when a rule is recognised. This code will typically create a data structure that describes the information that has just been parsed. In compiler applications, these data structures are highly recursive and well represented by a tree structure. The structure is called an Abstract Syntax Tree.

In the following very simple *yacc* grammar, there are 7 rules leading to 4 productions. Each (production or reduction) rule has a non-terminal at its left-hand side and may use terminals or non-terminals on its right-hand-side. *Identifier* and '* ' are terminals.

```yacc
term: Identifier { $$ = create_identifier_node($1); } |
    Number   { $$ = create_number_node($1); }
product: term { $$ = $1; }
    | product '*' term { $$ = create_multiply_node($1, $3); }
```

1. Parsing algorithms are categorised as LL “Left-to-right scanning of input, Lightmost derivation”, LR “Left-to-right scanning of input, Right-most derivation in reverse”, and LALR(k) “k-token Look-Ahead, Left-to-right scanning of input, Right-most derivation in reverse”. Handwritten parsers are typically LL. LR parsers are more powerful than LL, but need to be machine generated. Standard tools such as *yacc* and *bison* pursue the more compact LALR(1) approach.
expression:  product
  | expression '+' product

grammar:  expression '

| expression ';

There are two rules for the production of a term. One from an Identifier and one from a Number. A create_xxx routine is associated with each rule using the special $n variables to access inputs, and $$ to propagate a result.

The source sentence

\[ a + b * 5 ; \]

is parsed to create the AST

![Figure 5.2 Abstract Syntax Tree](image.png)

Shifts occur to advance to states that reflect a partial parse following each of +, b and *. Reductions occur as each rule is recognised

- one reduction after the a
- one reduction after the b
- three reductions after 5, for each of the rules
  - Number to term
  - product * term to product
  - expression + product to expression.

Each of these reductions activates the application code that creates the AST nodes. Once the external textual representation has been converted to an internal AST form, the program can be manipulated by compilation code to perform whatever checking, correction, rearrangement or optimisation is necessary to perform the translation.

An unambiguous grammar provides for only one possible parse tree for a given input, although a potentially infinite amount of lookahead may be required to distinguish between alternative partial trees for partial inputs. Conflicts arise from an attempt to distinguish alternatives prematurely. A reduce-reduce conflict arises if there are two alternate simplifications available. A shift-reduce conflict arises between an elaboration and a simplification of context.

A parser generator has a policy by which conflicts are resolved to produce a deterministic, although not necessarily useful parser. The default resolution of conflicts may be controlled by the programmer using the concept of precedence. The name reflects its original use to resolve precedence problems with arithmetic operators. The concept is of more general use. A special %prec non-terminal may be used to specify resolution of a conflict.

Translation programs are frequently presented with source files containing errors. As many of these errors as possible should be detected and diagnosed in a helpful fashion. It is rarely acceptable for a translator to just stop and report that a "parse error" has been encountered somewhere. The analysis must therefore continue after an error has occurred. In support of this philosophy, yacc suspends analysis and generates a special error token when an error is encountered. A carefully written grammar can make use of error to control resumption of the analysis.
5.2 Approaches to C++ Parsing

An LALR(1) shift-reduce parser generates a table driven parser for an unambiguous context-free grammar, subject to the requirement for detection of the right-most edge of a grammar production with 1 token of lookahead. The C++ grammar is ambiguous, context-dependent, and potentially requires infinite lookahead to resolve some ambiguities:

```
int(x), y, *const z;  // int x; int y; int *const z;
```

Is a comma-separated list of declarations in which the first is redundantly parenthesised, whereas changing the final list element:

```
int(x), y, new int;  // ((int(x)), (y), (new int));
```

gives a list of expressions, the first two of which are redundant, and the third causes a memory leak.

Other ambiguities are resolved by the language definition:

```
int(x), y, z = 0;  // int x; int y; int z = 0;
```

This could be an expression too, but isn’t. It may not be possible to determine the meaning until a potentially infinite amount of further source text has been analysed.

An LALR parser is not an obvious match to these requirements. However the alternatives are worse. A (bottom-up) LALR parser is faster and more compact than an LR parser, and able to handle all grammars that could be handled by a simpler (top-down) LL parser, and so the most widely used parsers are based on LALR(1).

In order to satisfy the constraints of an LALR(1) parser, the ambiguities, context dependence, and lookahead problems of C++ must be resolved.

The dragon book [Aho86] recognises that the boundaries between lexical, syntactic and semantic analysis are not clear cut.

Traditional C++ approaches seek a correct high resolution parse. As a result, the boundary between syntactic and semantic analysis has to be shifted to exploit semantic information during syntactic analysis by the parser and to leak semantic information through to the lexer. Use of semantic information during syntactic analysis requires very tight coupling to ensure that scope context is honoured and that changes of name visibility in mid-statement are correct. [Roskind81] provides a particularly unpleasant example where a change of classification midway through an apparent declaration leads to a contradiction, that is only resolvable as an expression.

The two variants of the new parsing approach described in this chapter do not move the boundary. The parser proceeds without full semantic knowledge and produces a result that is syntactically consistent, but sometimes semantically incorrect. An additional pass is therefore added to the semantic analysis to correct the inaccuracies of the syntactic analysis.

The advantages of this approach are:

- elimination of type (and template) tagging
- much simpler grammar that more closely follows the standard
- clear separation of syntactic and semantic processing
- conversion of syntactic ambiguities to semantic ambiguities
  - disentangles resolution from grammar implementation
- conversion of syntactic errors to semantic errors
  - avoids some losses of synchronization
  - provides more opportunities for helpful error diagnostics
Meta-compilation for C++

Parsing

- probably very slightly smaller code size
- more functionality to be coded
- simpler context for the code

The disadvantages of this approach are:
- introduction of new ambiguities
- an additional semantic correction pass
- additional semantic validity checking
- probably very slightly slower
- more functionality to be invoked
- more functionality to be executed
- simpler context for the code

Two variants of the approach are described. The previous implementation (available on the net in the v1 subdirectory) uses multiple passes to resolve ambiguities. The current implementation (available on the net in the v2 subdirectory) uses a superset grammar approach enabling operation without type or template information, as is necessary to support meta-programming consistently (Section 4.2.3).

The context-dependencies of C++ are described in detail in Appendix F.2.1.

5.3 Alternatives

Before developing the FOG parser, a brief review of the available alternatives was made. These alternatives are described in this section, along with two others that were not known at the time. Unfortunately, the developers of commercial C++ compilers do not make their parsers freely available in the public domain, and so the many proprietary implementations cannot be considered. However, comparison of the public domain approaches is sufficient to shed useful light on the difficulties.

5.3.1 Roskind grammar

A yacc-able C++ 2.1 grammar was made available by Jim Roskind [Roskind91]. This grammar dates from 1991 and has not been updated to handle C++ facilities such as templates or exceptions foreshadowed by the ARM [Ellis90], or to incorporate concepts such as bool and namespace added during standardisation. The paper accompanying the grammar provides a very insightful discussion into the source of the parsing problems and some rather pathological examples, whose correct interpretation is debatable.

The grammar resolves context dependence by a "lex hack", so that the lexer classifies identifiers as either IDENTIFIER or TYPEDEFname. The paper notes the need for another such hack to resolve template names.

The grammar is no more than a grammar. There is no action code to react to successfully analysed constructs, and only dummy hooks at the places where symbol table maintenance must be performed. The grammar code has no error recovery.

Ambiguities are resolved but are not removed from the grammar code. Some ambiguities are eliminated by rewriting parts of the grammar. Others are carefully analysed to ensure that the default ambiguity resolution policy of the parser generator chooses the required alternative.

Some potential lookahead problems are resolved by structuring the grammar code to recurse on the right-hand side, or by flattening out, in each case deferring reductions until more context has been seen. Other lookahead problems are resolved using the disambiguation policies of the previous paragraph. Further
problems could have been resolved by more flattening, but were perceived not to merit resolution while there was a possibility that the C++ grammar could change. The grammar has a total of 24 shift-reduce and 18 reduce-reduce conflicts, originating from 11 ambiguities. (Since conflicts occur between states, an ambiguity results in more than one conflict if the ambiguity affects more than one state transition.)

5.3.2 gcc

The GNU C compiler [Stallman98] has evolved to handle Objective C and C++. The compiler continues to improve, and is close to the C++ standard, but currently (version 2.8.0) experiences significant problems with template instantiation, because of the lack of a compilation database. These problems would not affect the use of gcc as a foundation for FOG.

The gcc compiler is portable to a very large number of operating systems, on which a build process normally involves compiling gcc through the local compiler, then recompiling gcc using the potentially better optimisations of gcc. The need to bootstrap through the local compiler requires extensive conditionaisation so that the gcc sources avoid the defects of all known compilers. gcc source code is therefore harder to read than it might be.

gcc source code is necessarily written in C, and so lacks the modularization and polymorphism that can be achieved using classes and Object Orientation in a language such as C++. The internal data structures of gcc comprise a tree node that is a union of all possible expressions, operators, names, declarations, statements, files etc.

The gcc compiler is a complete compiler including good error recovery and diagnosis, and full symbol table maintenance. gcc is recognised to be of production quality.

The lexer is hand coded and makes a seven way categorisation of identifiers to disambiguate the subsequent parser. However the subsequent parser requires 109 %prec directives to resolve 704 conflicts explicitly, leaving 5 shift-reduce and 38 reduce-reduce conflicts to be resolved automatically.

5.3.3 CPPP

A C++ parser was developed at Brown University, as a general purpose tool for which a variety of applications were foreseen [Reiss95]. This parser has steadily evolved, however the most recent version available on the net is version 1.82 from 1996. It would appear that development stopped before facilities such as namespace or bool were implemented.

CPPP achieves a higher degree of decoupling between lexer, parser and database than gcc, and has a grammar that closely resembles the published C++ grammar. CPPP comprises three stages: a lexer, a lookahead parser, and a main parser. The lookahead parser recognises potentially ambiguous constructs and invokes custom parsing routines to look sufficiently far ahead to resolve the ambiguity. Additional tokens are inserted into the token stream so that the subsequent parser proceeds without ambiguity.

The P++ developers [Singhal96] report extending CPPP successfully, but only after resolving a fair number of bugs. CPPP was also used for Iguana [Gowing96].

The grammar has only one unresolved shift-reduce conflict, but uses precedence extensively (22 %prec) to suppress a further 410 conflicts. The high number of conflicts is misleading. Most are the result of flattening the expression syntax and using grammar precedence to implement arithmetic precedence. This results in a faster parser since an expression term is reduced just once, rather than once at each of the ten binary operator precedence levels of the C++ grammar.
5.3.4 PCCTS

The Purdue Compiler Construction Tool Set provides alternative versions of lex and yacc called DLG and ANTLR. ANTLR is an LL(k) parser generator. Use of LL principles provides the freedom and (for practical grammars) the necessity of incorporating semantic resolution within the parse. A C++ grammar is available for use with PCCTS [Lilley97], but is heavily disclaimed as initial and experimental. A significantly customised version of ANTLR is required by the grammar.

5.3.5 C++ to F-code translator

The work described in this thesis concerns meta-compilation and the FOG implementation. The inspiration for FOG arose from research aimed at the development of a compiler implementing optimisations appropriate to DSP processors. This compiler was to use F-code [Muchnick93] as its intermediate representation. Some work was performed on a C++ to F-code translator as part of this earlier research. At that time, there was no knowledge of the CPPP or PCCTS grammar and so there was an implementation choice between

- the out-of-date Roskind grammar (42 conflicts)
- the tightly coupled gcc grammar (747 conflicts)
- a custom solution

The author had previously extended gcc to perform automated documentation generation for C++ code. This had merely required a late read-only traversal of the internal data structures. The difficulties of debugging with union nodes, the lack of clear documentation on the semantics of each node, and the enormous 250,000 line size of the code indicated that wholesale extension of gcc could lead to considerable problems.

A simpler alternative of just reusing the gcc parser grammar was examined. Examination of the parser showed that the parser, lexer and program data base were too closely coupled, making separate re-use of the grammar alone impractical.

The seemingly large number of unresolved conflicts, out-of-date character and obscure coding of declarators in the Roskind grammar discouraged its use. Recognition that the introduction of tree-literals would impact the heart of the grammar indicated that a clean grammar should be the starting point. A custom solution seemed the only alternative.

5.4 FOG parsing

The original aims of FOG concerned facilities for

- automatic insertion of repetitious code into class declarations
- elimination of redundant source text
- support for algorithm-centric modularization

The first aim is satisfied by derivation rules.

The others correspond to what is now known as weaving in the Aspect Oriented Programming world.

FOG/1 - superficial guided parse

It was perceived that these goals could be satisfied by a very simple parser assisted by extra keywords (guides). The first implementation therefore explicitly annotated C++ source adding new reserved words such as constructor, function, variable and type for the guides. Lines that needed special treatment could be easily identified and other lines copied from input to output without interpretation.
constructor Class::Class(int aSize) : _size(aSize) {};

This approach imposed a language incompatibility. For a legal C++ source file to be acceptable to FOG/1, it was necessary to manually add the guides to every declaration.

A second problem arose as to what syntax to use following the guide. The C++ declaration syntax is complicated and highly recursive. There are better syntaxes than the C++ syntax [Werther96], but they look very out of place in a C++ program. The C++ declaration syntax was preserved, and the problem of recursion of function signatures was partially solved by treating the function signature as unparsed text to be copied through to the output. This hid the problem for most functions but could not cope with pointers to functions, where the name is buried in, rather than preceding, punctuation.

    variable int (*v)();
    function int (*f())();

**FOG/2 - pragmatic guided parse**

Resolution of the function signature problem mandated an accurate parse of the signature. This problem had already been solved as part of the C++ to F-code translator and so the relevant part of the grammar was re-used. C++ function signatures comprise parameter types, names and optional default values. The default value was initially left unparsed, with the text copied unchanged to the output.

    function void f(int a = unparsed_text);

This usually works, but requires recognition of the ) or , that terminates the initializer. The initializer is an expression and so is subject to the use of templates. A simple parse of the initializer in

    template <bool T1, int T2> class B;
    function void f(int a = B < c, 5>);

may identify the comma-terminated B < c as the first initializer, before misinterpreting the residue. It requires the knowledge that B in the unparsed text is a template to correctly resolve the instantiation. Reliable parsing of declarations requires reliable parsing of expressions too.

**FOG/3 - pragmatic full parse**

At this point it was becoming clear that many of the complexities of C++ parsing could not be avoided in FOG, and it seemed likely that every attempt to avoid a complexity would introduce a deficiency. This is a far from unique discovery. There are a number of commercial development tools that have taken short-cuts to parsing C++, with the result that interesting C++ programs are misinterpreted. The class browser of Microsoft Visual C++ is just one example.

Another large segment of the C++ to F-code grammar was therefore added to FOG, so that FOG contained most of the C++ grammar. The traditional C++ declaration / expression ambiguities did not (yet) arise, because expressions occurred only in the limited context of initializers.

The need for guides was a barrier to porting existing C++ code to exploit FOG. Some of the first code to be ported from C++ to run through FOG was some compiler code. This code made use of the guides as function names. Requiring that the member function type() be renamed did not seem sensible.

With the increasingly accurate C++ parsing in FOG, it was no longer essential to have the extra guides. Parsing was no longer made easy by their use.

Once the extra guide keywords had been eliminated, there were few fundamental differences between FOG and C++ syntax. Each of these was challenged and
eliminated, with the result that the FOG grammar is an almost pure\(^2\) superset of C++.

$\text{-parsing}$

Tree-literals were originally envisaged as being resolved by a preprocessor before the real code emerged. A tree-literal was therefore recognised very early and replaced. Replacement could occur anywhere, including within strings. The semantics of the replacement were purely lexical, and so the replaced text could contribute partial strings and partial reserved words. This was flexible, powerful, undisciplined and awkward to implement. The ability to handle partial tokens required an unpleasant ability to recurse earlier lexing stages. The implementation resulted in a very complicated lexer, that had to maintain a stack of states according to how far through a string / character / number it was when a replacement started. It was far from clear that the implementation would behave correctly under perverse usage.

Recognition that the ANSI C string concatenation could be generalised to concatenation of adjacent textual tokens revealed that there was no need for substitution within strings:

\begin{quote}
"built at $\text{time}$ on $\text{date}\)"  // Not FOG
\end{quote}

It could happen almost as easily between strings:

\begin{quote}
"built at "$\text{time}" on "$\text{date}\)"  // FOG
\end{quote}

This then enabled the lexer to be more disciplined; preprocessor tokens could be identified first, then substitution could occur token by token, with only a minor complexity in retokenizing the result of a concatenation. Retokenisation was eventually discarded as unnecessary (Section 3.1.1.4).

**Ambiguity resolution**

Removal of the guides requires the syntaxes for typedefs, variables and functions to coexist. Ambiguities arise:

\begin{verbatim}
T ( A );
\end{verbatim}

could be a constructor for \(T\) with an unnamed parameter of type \(A\). Or it could be a variable of type \(T\) with the redundantly parenthesised name \(A\). This problem is traditionally resolved by ensuring that the lexer has the semantic information available to classify \(T\) and \(A\) as \texttt{class-name} or \texttt{type-name} or \texttt{identifier}. In C++, this classification requires accurate scope context and symbol table maintenance to ensure that an \texttt{identifier} is classified as a \texttt{type-name} at the declaration point.

Type information was not being used in the FOG parse and it seemed desirable to continue not using type information. The few ambiguities that appeared as a result were resolved by back-tracking, which is described in Section 5.8.

Development of the C++ to F-code translator had revealed how difficult resolving the expression / declaration ambiguity was. The grammar grew unpleasantly large as productions were elaborated to create sub-productions without ambiguities. Back-tracking was introduced to support sequential rather than concurrent consideration of alternatives.

Since FOG was then only analysing declarations and meta-statements, the major C++ ambiguities did not arise, only minor problems and implementation inconveniences. It was not necessary to perform semantic correction of syntactic errors, the parse-declarations-first policy disambiguated adequately.

---

2. Some very minor exceptions are listed in Section 3.3.
This summarises the evolution leading up to the previous implementation of FOG which uses back-tracking to resolve all ambiguities. Semantic leakage is limited to template names; type information is not used.

**Superset**

Consideration of the performance overheads associated with marking and back-tracking for every meta-statement, and of the validity of not using type information led to the more efficient superset parsing approach described in this chapter. The grammar for this approach has been implemented, cross-checked for completeness against the C++ standard, and processed by both *bison* and *yacc* to show lack of fundamental ambiguity, and successfully used to parse C++ programs. The *bison* report file has been used to verify correct resolution of the 24 conflicts that result from the 4 residual C++ ambiguities, 1 introduced ambiguity, and 5 implementation artefacts. These ambiguities are summarised in the comment header of Appendix B and described in Appendix F.2.

The ambiguities and semantic corrections resulting from the lack of type information apply to this superset parsing approach, and to a lesser extent to the multi-pass parsing approach.

The back-tracking to perform a binary tree search to resolve the template name ambiguities applies to both approaches.

### 5.5 Analysis of the C++ Grammar

The parsing approach described in this thesis deviates from accepted practice. It is therefore necessary to justify that the approach is sound. In order to do this we must first understand the standard problems and then identify any new problems before showing how the new approach resolves them. This requires a fairly detailed examination of some aspects of the C++ grammar and the introduction of a notation that supports reasoning about that grammar.

#### 5.5.1 Notation

The dragon book [Aho86] describes two different notations for defining languages.

**Regular Expressions**

The simple regular expression notation supports description of classes of character sequences and is the basis for the *lex* lexer generator. An identifier can be defined by

```
[A-Za-z][0-9A-Za-z]*
```

The expression starts with a character from the class containing the alphabetic characters or underscore and continues with an arbitrary number of repetitions of the second class that adds the numeric characters. This is a compact notation but is unable to express recursion and so cannot describe the language of matched nested braces ({}, {{}}, {{{}}} etc.). The inability to specify matching delimiters prevents the use of regular expressions to define most (if not all) programming languages; the use of matched parentheses to enforce arithmetic precedence is almost universal.

**Context-Free Grammars**

A (Context-Free) Grammar (CFG) is used to describe a more complicated language and is the basis of the *yacc* parser generator. A grammar is defined by rules that operate on the terminals and non-terminals of the language. Terminals correspond to the input tokens, non-terminals appear as the left-hand side of production rules involving terminals and/or non-terminals on their right-hand side. The brace language may be defined using ( and ) as terminals and *braces* and
grammar as non-terminals. grammar is the distinguished non-terminal that defines the language.

\[
\begin{align*}
\text{grammar:} & \quad \text{braces} \\
& \quad | \quad \{ \text{grammar} \} \\
\text{braces:} & \quad \{ \}
\end{align*}
\]

The grammar is specified using a Backus-Naur Form. Alternative rules producing the same non-terminals are separated by \( | \). When a clear multi-line formatting policy is used as in the C++ standard, the \( | \) may be omitted. Rules are variously referred to as production rules, or reduction rules.

The availability of intermediate non-terminals gives Context-Free Grammars much greater power than regular expressions. However the requirement to use multiple rules and the transformation of repetition into recursion makes it difficult to reason about the grammar.

### Extended Regular Expression notation

C++ comprises two relatively independent subgrammars, one to define expressions and another to define declarations. It is well known that there are sentences such as:

\[
\text{int (var);} 
\]

that are ambiguous. It could be a functional cast of the variable \( \text{var} \) to an integer value or a declaration of the redundantly parenthesised \( \text{var} \) as an integer. Identifying these ambiguities in the grammar is difficult because a few hundred inter-related rules are not a convenient representation for logical reasoning.

We need to be able to substitute one rule in another in order to derive the rules that identify each non-terminal with respect to terminals, or relatively fundamental non-terminals. We will therefore extend regular expressions so that C++ syntax can be represented. We can then represent an expression as one extended regular expression, a declaration as another, and identify the ambiguities by comparing their terms.

Both regular expressions and context-free grammars can describe the concatenation of lexically adjacent elements, but only context-free grammars support a complex ordering through nesting of non-terminals. We therefore introduce \( \odot \), a functional operator to support more arbitrary ordering in regular expressions.

The \( \odot \) operator is always applied to a specific argument and so associates from right to left. This of course differs from the composition operation in functional languages, since we are interested in successive application, not in function composition.

The \( \odot \) operator has higher precedence than lexical concatenation. Thus:

\[
\alpha Z \odot \gamma 
\]

represents \( \alpha \) concatenated with the application of \( Z \) to \( \gamma \). If \( Z \) denotes application of braces, and \( \alpha \) and \( \gamma \) are the identifiers \( a \) and \( g \), the above expression denotes the sentence

\[
a \{ g \};
\]

The nested braces language, using \( \varepsilon \) as the empty set of sentences, is

- \( Z^* \varepsilon \) \quad \{ \}
- \( Z^*Z^* \varepsilon \) \quad \{ \{ \} \}
- \( Z^*Z^*Z^* \varepsilon \) \quad \{ \{ \{ \} \} \}
- \( Z^*Z^*Z^*Z^* \varepsilon \) \quad \{ \{ \{ \} \} \}
- etc.

which we abbreviate to
More practically, we represent the pointer prefix of a declarator by $P_d(\zeta)$ and the array suffix of a declarator by $A_d(\eta)$, where $\zeta$ is a free symbol denoting a ptr-operator, $\eta$ is a free symbol denoting an assignment-expression, and $?$ is the standard regular expression operator denoting 0 or 1 of.

$$P_d(x::y::*\text{const})\cdot A_d(5)\cdot v$$

therefore denotes

$$x::y::*\text{const } v [5]$$

The subscripts in these names form part of a compact naming policy: $P$ for pointer, $A$ for array, $P_d$ for a pointer valid for use in a declarator, $A_e$ for an array valid for use in an expression.

Since the parenthesised parameterization is not usually significant in ambiguity reasoning, we can refer more simply to

$$P_d^*A_d^*\theta_d$$

where $\theta_d$ is any name valid in a declarator.

The full notation is summarised in the following tables

<table>
<thead>
<tr>
<th>(Non-)Terminal</th>
<th>Notation</th>
<th>Description or example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free (untyped) symbols</td>
<td>$\alpha$</td>
<td></td>
</tr>
<tr>
<td>Built-in type (one word)</td>
<td>$\beta$</td>
<td>int</td>
</tr>
<tr>
<td>Declaration</td>
<td>$\delta$</td>
<td>simple-declaration</td>
</tr>
<tr>
<td>The empty set</td>
<td>$\epsilon$</td>
<td></td>
</tr>
<tr>
<td>Pointer Type</td>
<td>$\zeta$</td>
<td>ptr-operator</td>
</tr>
<tr>
<td>Assignment-expression</td>
<td>$\eta$</td>
<td>assignment-expression</td>
</tr>
<tr>
<td>Name in declaration</td>
<td>$\theta_d$</td>
<td>declarator-id</td>
</tr>
<tr>
<td>Name in expression</td>
<td>$\theta_e$</td>
<td>id-expression plus a bit$^0$</td>
</tr>
<tr>
<td>Name other than local destructor</td>
<td>$\theta$</td>
<td></td>
</tr>
<tr>
<td>Constant-expression</td>
<td>$\kappa$</td>
<td>constant-expression</td>
</tr>
<tr>
<td>Character, Number or String</td>
<td>$\lambda$</td>
<td>literal</td>
</tr>
<tr>
<td>Parameter-Declaration</td>
<td>$\pi$</td>
<td>parameter-declaration</td>
</tr>
<tr>
<td>Parameter-Declaration and Assignment-Expression</td>
<td>$\rho$</td>
<td>$\rho = \eta \land \pi$</td>
</tr>
<tr>
<td>Type-name in declaration</td>
<td>$\sigma_d$</td>
<td>size_t</td>
</tr>
<tr>
<td>Simple-type-specifier</td>
<td>$\sigma_e$</td>
<td>simple-type-specifier</td>
</tr>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>type-id</td>
</tr>
<tr>
<td>Generalised Assignment-Expression</td>
<td>$\chi$</td>
<td>$\chi \supset \eta$</td>
</tr>
<tr>
<td>Generalised Parameter-Declaration or Assignment-Expression</td>
<td>$\omega$</td>
<td>$\omega \supset \eta \cup \pi$</td>
</tr>
</tbody>
</table>

Table 5.1 Terminals and Non-Terminals

a. See Section 5.5.2.1.

There are no operators that apply to more than one operand, although operators such as $P_d$ may take additional parenthesised parameters.

The infix operator $O_j$ applies to one operand that denotes two (or three) terms, each independently of the same form as the one operand. Thus, a particular
Operator | Notation | Description
--- | --- | ---
Independent instances | \(\alpha_1, \alpha_2, \alpha_3, \alpha_4\) | 
Zero or one | \(Z^*\alpha\) | \(\alpha\) or \(Z\alpha\)
Zero or more | \(Z^+\alpha\) | \(\alpha\) or \(Z\alpha\) or \(Z^*Z\alpha\) or \(Z^*Z^*Z\alpha\) or etc..
One or more | \(Z^*\alpha\) | \(Z\alpha\) or \(Z^*Z\alpha\) or \(Z^*Z^*Z\alpha\) or \(Z^*Z^*Z^*Z\alpha\) or etc..
Exactly one of | \{\(\alpha_1\) \(\alpha_2\)\} | \(\alpha_1\) or \(\alpha_2\)
Comma separated list of | L\(\alpha\) | \(0_L\alpha\) is \(\in\) \(1_L\alpha\) is \(\alpha\) \(2_L\alpha\) is \(\alpha_1, \alpha_2\) \(3_L\alpha\) is \(\alpha_1, \alpha_2, \alpha_3\) etc. \(4_L\alpha\) \(5_L\alpha\) \(6_L\alpha\) \(7_L\alpha\) and \(8_L\alpha\) have at least 0 / 1 elements
Covers/Contains | \(\alpha_1 \subseteq \alpha_2\) | Every sentence of \(\alpha_1\) is contained in \(\alpha_2\)
Strictly Covers / Contains | \(\alpha_1 \subset \alpha_2\) | Every sentence of \(\alpha_1\) is contained in \(\alpha_2\), and some sentence of \(\alpha_2\) is not contained in \(\alpha_1\)
Intersection | \(\alpha_1 \cap \alpha_2\) | Sentences common to \(\alpha_1\) and \(\alpha_2\)
Union | \(\alpha_1 \cup \alpha_2\) | Sentences of \(\alpha_1\) or \(\alpha_2\)

Table 5.2 Mathematical Operators

Lexical presentation of the tertiary \(O_\eta\) is "test ? 5 / 8 : 6", since each of "test", "5 / 8" and "6" are assignment-expressions.

The list operator \(L\) uses a pre-superscript rather than a post-superscript to highlight the distinction between the repetition of the operand many times and multiple application of the operator. The operand is repeated with no constraint between the operands. Thus \(2_L\alpha\) covers the 4 possibilities of \(Z\) present or absent independently for each operand whereas \(Z^*Z^*L\alpha\) covers only the 2 possibilities of \(Z\) jointly present or absent.

Overloading of lexical tokens and other simple lexical properties lead to the properties shown in Table 5.4.

5.5.2 C++ Grammar Properties

This notation will now be applied to analyze the declaration and expression syntaxes. The analysis is a little lengthy and ignores a number of peripheral syntaxes that do not contribute to ambiguities. Thus ..., which is unique to a parameter-declaration-clause, and sizeof(), which is unique to an expression are excluded from the analysis.

5.5.2.1 Names

Before we can analyse the grammar, and show that the new approach is compatible, we must first understand how names are used in the standard grammar.

When a variable is declared, it complies to a syntax that simplifies a little to:

\[\text{decl-specifier-seq declarator = initializer ;}\]

The decl-specifier-seq comprises the miscellaneous qualifiers such as static or extern and a type-specifier-seq that provides the type such as const unsigned int (but no pointers). The declarator comprises a name in the form of a declarator-id potentially wrapped up with pointer prefixes, function and array suffixes and clarifying parentheses.

When a variable is used in an expression, it complies to the id-expression syntax.
Table 5.3 Grammatical Operators

a. The = {} form of *initializer* is incorporated directly into a generalised O₁ to slightly reduce the number of distinct declarator / expression operators.

The *id-expression* and *destructor-id* are therefore the main name concepts, that have subtle but significant differences. Understanding the distinctions from the C++ grammar is quite hard, so the same information is presented in graphical form in Figure 5.3. The diagram is a kind of Venn diagram in which different forms of name are arranged in four columns and thirteen rows. Shaded areas indicate the coverage of each production.

Four columns represent the possible scope nestings of a name:
- unscoped name (e.g. `name`)
- nested name (e.g. `Nested::name`)
- global scope nested name (e.g. `::Nested::name`)
- global name (e.g. `::name`)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Notation</th>
<th>Lexical Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array declarator suffix</td>
<td>A_d(k)*α</td>
<td>α [ ]</td>
</tr>
<tr>
<td>Array expression suffix</td>
<td>A_e(<em>L</em>η)*α</td>
<td>α [ η ]</td>
</tr>
<tr>
<td>Array for declarator and expression</td>
<td>A_de(k)*α</td>
<td>α [ k ]</td>
</tr>
<tr>
<td>Array for declarator or expression</td>
<td>A_t(L*η)*α</td>
<td>α [ η ]</td>
</tr>
<tr>
<td>Parenthesis Brackets</td>
<td>B*α</td>
<td>( α )</td>
</tr>
<tr>
<td>Cast expression</td>
<td>C(τ)*α</td>
<td>( τ ) α</td>
</tr>
<tr>
<td>decl-specifier prefix</td>
<td>D*α</td>
<td>extern α</td>
</tr>
<tr>
<td>exception-specification suffix</td>
<td>E*α</td>
<td>α throw ( )</td>
</tr>
<tr>
<td>Function declarator suffix</td>
<td>E^*V^<em>F_d(L</em>π)*α</td>
<td>α ( π₁, π₂ ) const throw ( )</td>
</tr>
<tr>
<td>Function call suffix</td>
<td>F_d(L*η)*α</td>
<td>α ( η₁, η₂ )</td>
</tr>
<tr>
<td>Function call and declarator</td>
<td>F_de(L*p)*α</td>
<td>α ( p₁, p₂ )</td>
</tr>
<tr>
<td>Function call or declarator</td>
<td>E^*V^<em>F(L</em>π)*α</td>
<td>α ( ω₁, ω₂ ) const throw ( )</td>
</tr>
<tr>
<td>Assigned initializer</td>
<td>I(η)*α</td>
<td>α = η</td>
</tr>
<tr>
<td>Constructed initializer</td>
<td>J(L*π)*α</td>
<td>α ( η₁, η₂ )</td>
</tr>
<tr>
<td>(Non-pointer) Prefix operator</td>
<td>O₂*α</td>
<td>++ α</td>
</tr>
<tr>
<td>Infix operator</td>
<td>O₁*α</td>
<td>α₁ / α₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td>α = ( 5 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>α₁ = α₂ ; α₃</td>
</tr>
<tr>
<td>Suffix operator</td>
<td>O₃*α</td>
<td>α ++</td>
</tr>
<tr>
<td>ptr-operator</td>
<td>P_d(z)*α</td>
<td>* volatile α</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp; α</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Class::* const α</td>
</tr>
<tr>
<td>Pointer for expression</td>
<td>P_d(z)*α</td>
<td>* α</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp; α</td>
</tr>
<tr>
<td>Unified Pointer operator</td>
<td>P(z)*α</td>
<td>*cv-qualifier-seq opt α</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp; α</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D*opt::cv-qualifier-seq opt α</td>
</tr>
<tr>
<td>type-specifier prefix</td>
<td>T*α</td>
<td>int α</td>
</tr>
<tr>
<td>cv-qualifier-seq suffix</td>
<td>V*α</td>
<td>α const</td>
</tr>
</tbody>
</table>
Figure 5.3  C++ Names

Thirteen rows represent each of the different categories of name. The top 8 rows correspond to type-names:

- ...decl-specifier, keywords such as virtual, static and friend
- elaborated-type-specifier, an enum or class reference
- class-specifier, a class definition
- enum-specifier, an enum definition
- cv-qualifier, const or volatile
- β, a single word built-in type such as int
- template-id (e.g. FixedSizeArray < 4 >)
- identifier (e.g. MyType)

The bottom five rows denote non-type names:

- template-id (e.g. sort < int >)
- identifier (e.g. my_variable)
- operator-function-id (e.g. operator++)
- conversion-function-id (e.g. operator MyClass **)  
- destructor name (e.g. -MyClass)
<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_d \subset \delta_e \subset T^* \subset D^*$</td>
<td>Application type-names</td>
</tr>
<tr>
<td>$\beta \subset \delta_e \subset T^* \subset D^*$</td>
<td>Built-in type-names</td>
</tr>
<tr>
<td>$\sigma_d \subset \theta_d$</td>
<td>Declarator type-names</td>
</tr>
<tr>
<td>$\theta_e \subset \theta_d$</td>
<td>Non-type-names</td>
</tr>
<tr>
<td>$V^* \subset T^*$</td>
<td>Non-type-names</td>
</tr>
<tr>
<td>$A_{de} \cdot \alpha \subset A_{de} \cdot \alpha$</td>
<td>Declarator type-names</td>
</tr>
<tr>
<td>$A \cdot \alpha \supset A_{de} \cdot \alpha$</td>
<td>Declarator type-names</td>
</tr>
<tr>
<td>$F_{de} \cdot \alpha \subset F_{de} \cdot \alpha$</td>
<td>Declarator type-names</td>
</tr>
<tr>
<td>$E_{de}^* F_\alpha \supset F_\alpha$</td>
<td>Declarator type-names</td>
</tr>
<tr>
<td>$I \cdot \alpha \subset O_{de} \cdot \alpha$</td>
<td>Declarator type-names</td>
</tr>
<tr>
<td>$P_{de} \cdot \alpha \subset P_{de} \cdot \alpha$</td>
<td>Declarator type-names</td>
</tr>
<tr>
<td>$P \cdot \alpha \supset P_{de} \cdot \alpha$</td>
<td>Declarator type-names</td>
</tr>
<tr>
<td>$B^* \cdot \tau = C(\tau) \cdot \epsilon = E_{de}^* (\tau) \cdot \epsilon$</td>
<td>Parenthesis / cast null / abstract function</td>
</tr>
<tr>
<td>$B^* \cdot \tau = F_{de}^* (\tau) \cdot \epsilon$</td>
<td>Parenthesis / abstract function</td>
</tr>
<tr>
<td>$B^* \cdot \tau = F_{de} (\tau) \cdot \epsilon$</td>
<td>Parenthesis / null-call</td>
</tr>
<tr>
<td>$B^* \cdot \tau = F_{de}^* (\tau) \cdot \epsilon$</td>
<td>Parenthesis / null-call or function</td>
</tr>
</tbody>
</table>

Table 5.4 Properties

Figure 5.3 shows that qualified-id grammar production covers all categories of nested non-type-name, and that a declarator-id covers all possible non-type-names and user-definable type-names.

The irregular shape of the $\theta_e$ contributions to a primary-expression is the source of many difficulties in implementing the parser grammar. Reduce-reduce conflicts arise from a need to commit to a declarator-id or primary-expression before sufficient lookahead context has been examined. Part of this is just a trap for the unwary implementor. Since $\theta_e \subset \theta_d$, no conflict need arise. It is just the unhelpful way the grammar is written that is a problem.

Name differences between declarator and expression

A type-name is not generally valid in an expression. However, a specific variant of function call in the syntax for postfix-expression, supports use of a type-name as the function name, and serves to invoke a constructor or functional cast.

Omission of an unscoped destructor name from id-expression resolves the ambiguity between the one's complement operator and a destructor for

```
~non_class_name & 7;
```

and gives the correct interpretation (§5.3.1-9) of
Meta-compilation for C++

Parsing

-ClassName();

Omission of a global `conversion-function-id` from `id-expression` is semantically correct, but represents a needless syntactic complexity, since an ambiguity resolution of `something-ending-in-:: conversion-function-id` exploiting the syntactic exclusion requires `something-ending-in-::` to be meaningful. The only construct ending in `::` is a `nested-name-specifier`, whose presence contradicts the presence of a global name.

Omission of a global destructor name from `id-expression` is similarly semantically correct, but syntactically redundant.

The same argument applies to a global `template-id`, however in this case, it would appear that the omission is an error. Given:

```cpp
template <class T> void sort(T *anArray, size_t arraySize);
```

Refusal to permit the hopefully redundant `::` in

```cpp
p = &::sort<int>;
```

seems unreasonable.

**Conclusion**

The global name exclusions from a `primary-expression` can be ignored syntactically. A subsequent semantic check may yield a helpful diagnostic. Only the omission of the local destructor need be honoured.

Once the distinction between `type-name` and `identifier` is removed, the local destructor exclusion is the sole difference between a `declarator-id` and a name in a `primary-expression`. The generalised name coverage shown as 0 is used for both purposes in Section 5.7. This covers some syntactically meaningless names in an expression, but misses out the local destructor from a `declarator-id`. The omission will be covered by a complement expression and must be repaired semantically.

### 5.5.2.2 Declarators, Declarations and Type Identifiers

The C++ grammar defines

```cpp
declarator:
  direct-declarator
  ptr-operator declarator
direct-declarator:
  declarator-id
  direct-declarator ( parameter-declaration-clause )
  direct-declarator ( parameter-declaration-clause )
  direct-declarator ( parameter-declaration-clause )
  direct-declarator ( parameter-declaration-clause )
  cv-qualifier-seqopt exception-specificationopt
declarator ( declarator )
```

It is the two level recursion between these productions that makes them difficult to understand. Two levels are required because a CFG cannot express both prefix and suffix elaboration in the same production unambiguously. Considering:

```cpp
affixed-production:
  terminal
  prefix affixed-production
  affixed-production suffix
```

It is unclear whether the prefix or suffix production rule is reduced first in:

```cpp
prefix terminal suffix
```

In the extended regular expression notation outlined above, functional operators are used for lexical prefixes, suffixes and their combinations. The prefix-suffix ordering ambiguity is therefore removed and all forms of lexical decoration can be expressed uniformly. We may express the `declarator` syntax as
Parsing

Meta-compilation for C++

declarator = \{direct-declarator\}

\{P_d \cdot \text{declarator}\}

direct-declarator = \begin{cases} \theta_d \\
E \cdot V \cdot F_d \cdot \text{direct-declarator} \\
A_d \cdot \text{direct-declarator} \\
B \cdot \text{declarator}
\end{cases}

The recursion in the first choice can be simplified:

declarator = P_d^* \cdot \text{direct-declarator}

and then substituted in the second to give

direct-declarator = \begin{cases} E \cdot V \cdot F_d^* \\
A_d \\
B \cdot P_d^*
\end{cases} \cdot \theta_d

declarator = P_d^* \cdot \begin{cases} E \cdot V \cdot F_d^* \\
A_d \\
B \cdot P_d^*
\end{cases} \cdot \theta_d

Applying the same analysis to other parts of the C++ grammar we find that

abstract-declarator = P_d^* \cdot \begin{cases} \begin{cases} E \cdot V \cdot F_d^* \\
A_d \\
B \cdot P_d^*
\end{cases} \cdot \theta_d \\
\end{cases} \cdot \varepsilon

type-id = \tau = T^+ \cdot P_d^* \cdot \begin{cases} \begin{cases} E \cdot V \cdot F_d^* \\
A_d \\
B \cdot P_d^*
\end{cases} \end{cases} \cdot \varepsilon

new-type-id = T^+ \cdot P_d^* \cdot A_d^* \cdot \varepsilon

init-declarator = \{I\} \cdot \begin{cases} \begin{cases} E \cdot V \cdot F_d^* \\
A_d \\
B \cdot P_d^*
\end{cases} \end{cases} \cdot \theta_d

29-June-2001

Page 161
\[ \text{init-declarator-list} = +L \cdot \left\{ I \right\} \cdot P_d^{*} \cdot \left\{ \begin{array}{c} E^2 \cdot V^2 \cdot F_d^* \\ A_d \\ B \cdot P_d^{*} \end{array} \right\} \cdot \theta_d \]

\[ \text{parameter-declaration} = \pi \text{ where} \]

\[ \pi = D^+ \cdot I^? \cdot P_d^{*} \cdot \left\{ \begin{array}{c} E^2 \cdot V^2 \cdot F_d^* \\ A_d \\ B \cdot P_d^{*} \end{array} \right\} \cdot \left\{ \begin{array}{c} \theta_d \\ \varepsilon \end{array} \right\} \]

(The usage of \( \theta_d \) in a parameter-declaration is syntactically correct although semantic constraints allow only an identifier.)

Ignoring the optional ellipsis which is not a source ambiguity:

\[ \text{parameter-declaration-clause} = B \cdot L \cdot \pi \]

Generalising slightly by ignoring the constraint that a decl-specifier-seq\(_{opt}\) applies only to the first element of an init-declarator-list:

\[ \text{simple-declaration} = \delta = D^+ \cdot L \cdot \left\{ I \right\} \cdot P_d^{*} \cdot \left\{ \begin{array}{c} E^2 \cdot V^2 \cdot F_d^* \\ A_d \\ B \cdot P_d^{*} \end{array} \right\} \cdot \theta_d \]

### 5.5.2.3 Expressions

Analysis of an expression omitting uniquely prefixed terms such as new or const_cast that may be conveniently considered to be part of \( \lambda \) leads to

\[ \text{primary-expression} = \left\{ \begin{array}{c} \theta_e \\ \lambda \\ B \cdot \text{expression} \end{array} \right\} \]

\[ \text{postfix-expression} = \left\{ \begin{array}{c} \text{primary-expression} \\ A_e \cdot \text{postfix-expression} \\ F_e \cdot \text{postfix-expression} \\ F_e \cdot \sigma_e \\ O_s \cdot \text{postfix-expression} \end{array} \right\} = \left\{ \begin{array}{c} A_e^* \\ F_e \\ F_e \cdot \sigma_e \\ O_s \\ B \cdot \text{expression} \end{array} \right\} \]

While analysing prefix operations, we lump the more mundane operators such as ++ under \( O_p \), but keep the potentially ambiguous pointer operators * and & as \( P_e \).
unary-expression = \{ O_p, P_e \} \cdot cast-expression

\{ unary-expression \} \cdot cast-expression = C^* \cdot unary-expression

= C^* \cdot \{ \{ O_p \} \cdot C^* \} \cdot cast-expression

Simplifying the nested arbitrary choice, and substituting the postfix-expression

\{ unary-expression \} \cdot cast-expression = \begin{cases} O_p^* \cdot \{ A_e \}^* & \text{if } \theta_e \\ P_e^* \cdot \{ F_e \}^* & \text{if } \lambda \\ C \cdot cast-expression & \text{if } F_e \cdot \sigma_e \\ B \cdot expression & \end{cases}

The ten levels of precedence for binary operators, tertiary operator and assignment are not significant to this analysis. All such operators are represented by $O_i$ and we may write

assignment-expression = \eta = O_i^* \cdot \{ O_p \}^* \cdot \{ A_e \}^* \cdot \{ \theta_e \} \\ P_e^* \cdot \{ F_e \}^* \cdot \{ \lambda \} \\ C \cdot cast-expression \cdot \{ F_e \cdot \sigma_e \} \\ B \cdot expression

expression = \begin{cases} +L \cdot O_i^* \cdot \{ O_p \}^* \cdot \{ A_e \}^* \cdot \{ \theta_e \} \\ P_e^* \cdot \{ F_e \}^* \cdot \{ \lambda \} \\ C \cdot cast-expression \cdot \{ F_e \cdot \sigma_e \} \\ B \cdot expression \end{cases}

expression-list = \begin{cases} +L \cdot O_i^* \cdot \{ O_p \}^* \cdot \{ A_e \}^* \cdot \{ \theta_e \} \\ P_e^* \cdot \{ F_e \}^* \cdot \{ \lambda \} \\ C \cdot cast-expression \cdot \{ F_e \cdot \sigma_e \} \\ B \cdot expression \end{cases}

The doubly nested one or more choices permit arbitrary ordering.
5.5.3 C++ Ambiguities (using type information)

We have derived relatively simple formulae for the major declaration and expression constructs. The utility of this representation will be shown by deriving formulae that describe the conventional C++ ambiguities, when type information is available. This derivation can then be revisited to assess the consequences of parsing without type information.

The major ambiguities occur between and within declarations and expressions, since these syntaxes lack unique keywords. Ambiguities in statement syntax are isolated, since a leading if isolates the syntax from all other syntax (but not from the dangling else problem).

We concentrate on declarations and expressions since these are the sources of problems:

- as many other implementors have already discovered
- as any attempt to implement a C++ grammar with yacc detects

5.5.3.1 Declaration / Declaration Ambiguity

A declaration / declaration ambiguity arises when

\[
\text{simple-declaration} = \delta = D^* \cdot L \cdot \left\{ I \right\}^* \cdot P_d^* \cdot \left[ \begin{array}{c}
E^* \cdot V^* \cdot F_d^* \\
A_d \\
B \cdot P_d^*
\end{array} \right] \cdot \theta_d
\]
is open to more than one interpretation. Multiple interpretations are possible because there is lexical overlap between the different operators.

\[ D^\theta_d \text{ and } D^\theta_d \text{ or } D^\theta_d \text{ or } D^\theta_d \]

A declarator-id (\(\theta_d\)) or ptr-operator (\(P_d\)) may start with a :: which may be ambiguous with respect to the last name in a declar-specifier (D).

```cpp
Class ::Scope::p
Class ::Scope::* p // A pointer to member
```

This ambiguity is not explicitly resolved in the standard, but a resolution may be inferred from the requirement to maximise the length of a declar-specifier-seq (§7.1-2). The resolution guarantees a semantic error.

```cpp
Class::Scope::p // Error name but no type
Class::Scope::* p // Error illegal ptr-operator
```

\[ A^\theta_d \text{ or } \theta_d \]

An array declarator may add a [] suffix to a name.

The names operator new[] and operator delete[] end in [], and the names operator new and operator delete exist. It is therefore unclear whether

```cpp
int operator new[];
```

does not declare an array or a scalar. Since neither alternative is semantically valid, the syntactic problem is academic.

\[ F^\theta_d \text{ or } \theta_d \]

A function declarator may add a () suffix to a name.

The name operator() ends in (), but there is no name operator so there is no ambiguity.

\[ F^\theta_d (\text{int})\text{ or } D^\theta_d \text{ or } D^\theta_d \text{ or } D^\theta_d \]

The overloaded usage of parentheses leads to an ambiguity between

- \(F^\theta_d (\text{int})\text{ - a single argument constructor declaration}
- \(D^\theta_d \text{ - a redundantly parenthesised variable declaration}

for:

```cpp
T(a)
```

Trimming \(\delta\) to match these two forms

\[
\delta_c = \begin{bmatrix}
1^J & \{ E^2 \cdot V^2 \cdot F_d \}^* \\
A_d & \end{bmatrix} \cdot F_d(\pi_c) \cdot \theta_d
\]

\[
\delta_v = D \cdot \begin{bmatrix}
1^J & \{ E^2 \cdot V^2 \cdot F_d \}^* \\
A_d & \end{bmatrix} \cdot B \cdot P_d^* \cdot \begin{bmatrix}
E^2 \cdot V^2 \cdot F_d \}^* \\
A_d & \\
B \cdot P_d^* & \end{bmatrix} \cdot \theta_d
\]

The initializers, array and function suffixes can be dismissed using the semantic constraints on a constructor, leaving

\[
\delta_c = F_d(\pi_c) \cdot \theta_d
\]
The ambiguity arises for \( \delta_{\text{c}} \cap \delta_{\text{v}} \), which is non-trivial when the name preceding the parentheses is \( \sigma_{\text{d}} = \theta_{\text{d}} \cap (D \cdot \varepsilon) \) (see Figure 5.3). The inside of the parenthesis is ambiguous when the parameter

\[
\pi = D^{+} \cdot I^{2} \cdot P_{d}^{*} \cdot \left\{ \begin{array}{c}
E^{2} \cdot V^{2} \cdot F_{d}^{*} \cdot A_{d} \\
B \cdot P_{d}^{*}
\end{array} \right\} \cdot \theta_{\text{d}}
\]

is ambiguous with respect to

\[
P_{d}^{*} \cdot \left\{ \begin{array}{c}
E^{2} \cdot V^{2} \cdot F_{d}^{*} \cdot A_{d} \\
B \cdot P_{d}^{*}
\end{array} \right\} \cdot \theta_{\text{d}}
\]

Trimming impossible terms gives the two constraints on the parenthesised ambiguity in

\[
\text{TypeName}(\pi)
\]

\[
\pi_{\text{c}} = \left\{ \begin{array}{c}
D^{+} \cdot \left\{ \begin{array}{c}
E^{2} \cdot V^{2} \cdot F_{d}^{*} \cdot A_{d} \\
B \cdot P_{d}^{*}
\end{array} \right\} \cdot \theta_{\text{d}} \\
E
\end{array} \right\}
\]

\[
\pi_{\text{v}} = \left\{ \begin{array}{c}
E^{2} \cdot V^{2} \cdot F_{d}^{*} \cdot A_{d} \\
B \cdot P_{d}^{*}
\end{array} \right\} \cdot \theta_{\text{d}}
\]

\[
\sigma_{\text{d}} = \theta_{\text{d}} \cap (D^{+} \cdot \varepsilon)
\]

is a simple solution to \( \pi_{\text{c}} = \pi_{\text{c}} \cap \pi_{\text{v}} \).

A more complicated solution arises through the recursive ambiguity between precedence and function argument parentheses.

\[
\left\{ \begin{array}{c}
E^{2} \cdot V^{2} \cdot F_{d}^{*} \\
A_{d}
\end{array} \right\} \cdot F_{d} \left\{ \begin{array}{c}
E^{2} \cdot V^{2} \cdot F_{d}^{*} \\
A_{d}
\end{array} \right\} \cdot \sigma_{\text{d}} \cdot \sigma_{\text{d}}
\]

leading to the complete form of the constructor / parenthesised variable ambiguity.
Parenthesis overloading would also appear to lead to an ambiguity between construction of an object and a redundantly parenthesised variable. However, construction of an object requires an explicit type and so the object construction must at least be of the form $D^*J(L^*\eta)^{\theta_d}$. The ambiguity of this with respect to a function declaration is considered next.

$D^*J(L^*\eta)^{\theta_d}$ or $D^*F_d(L^*\pi)^{\theta_d}$

There is a lexical ambiguity between

- $D^*J(L^*\eta)^{\theta_d}$ - a constructed object declaration
- $D^*F_d(L^*\pi)^{\theta_d}$ - a function declaration

for:

```cpp
T a(b), c(d), e(f);
```

Semantic constraints permit and require ambiguous `decl-specifier-seq` prefixes for the first element of an `init-declarator-list`, but require no `decl-specifier-seq` for subsequent elements. The D and L terms are therefore eliminated to give the following forms for each alternative `simple-declaration`.

\[
\delta_o = J(L^*\eta)^{\theta_d} \cdot P_d^* \cdot \left[ E^* \cdot V^* \cdot F_d^* \right] \cdot \left[ \begin{array}{c} A_d^* \\ B \end{array} \right] \cdot \theta_d
\]

\[
\delta_f = P_d^* \cdot F_d(L^*\pi)^{\theta_d} \cdot \left[ E^* \cdot V^* \cdot F_d^* \right] \cdot \left[ \begin{array}{c} A_d^* \\ B \end{array} \right] \cdot \theta_d
\]

These two terms are a direct match syntactically, subject to the recursive ambiguity to satisfy the parameter lists. This is $F_{de}^*(L^*\rho)$ and is analysed in the next section. The ambiguity is therefore:

\[
\delta_{of} = P_d^* \cdot F_{de}(L^*\pi)^{\theta_d} \cdot \left[ E^* \cdot V^* \cdot F_d^* \right] \cdot \left[ \begin{array}{c} A_d^* \\ B \end{array} \right] \cdot \theta_d
\]

5.5.3.2 Declaration / Expression ambiguity

The declaration / expression ambiguity arises when a

\[
simple-declaration = \delta = D^* \cdot L \cdot \left[ \begin{array}{c} I^* \\ J \end{array} \right] \cdot P_d^* \cdot \left[ E^* \cdot V^* \cdot F_d^* \right] \cdot \left[ \begin{array}{c} A_d^* \\ B \end{array} \right] \cdot \theta_d
\]

followed by a semi-colon, forming part of a statement, is ambiguous with respect to
which when followed by a semi-colon is an expression-statement.

The expression / declaration ambiguity may be identified by comparing terms to give

\[
\delta \cap ^+L \bullet \eta = D^0 \bullet ^+L \bullet \left\{ I \right\} \cap \left\{ \begin{array}{c}
F_{de} \\
A_{de} \\
B \bullet ^+L \bullet P_e^*
\end{array} \right\} \cap \left\{ \begin{array}{c}
\theta_\theta \\
\lambda \\
\sigma_\theta
\end{array} \right\}
\]

\( D^0 \) matches the absence of \( D \) in an expression.

\( ^+L \) enforces the expression requirement for at least one element.

Commutativity of prefix and suffix operators permits the intervening pointer to be traversed by the initializers so that: \( I \) resolves the ambiguity between \( I \) and \( O_l \) since \( I \subset O_l \). \( J \) is covered by an \( F_e \) in an expression.

\( P_e \) unifies \( P_e \) and \( P_d \) acquiring the \( P_e \) from the expression multiplier term.

\( F_{de} \) is the recursive ambiguity between \( F_d \) and \( F_e \) analysed below.

\( A_{de} \) resolves the ambiguity between \( A_d \) and \( A_e \) restricting the array argument to exactly one constant-expression.

\( ^+L \) is the identity operator necessary to cover \( B \bullet P_d^* \) by successive \( B \bullet ^+L \) and \( P_e \) terms from the expression multiplier term.

\( \theta_\theta \) resolves the ambiguity between \( \theta_d \) and \( \theta_e \) since \( \theta_e \subset \theta_d \).

The \( F_{de} \bullet \sigma_\theta \) terms arises because \( \sigma_\theta \subset \theta_d \).

A quick test of this formula makes the prediction that

\[ i = 0; \]

should be ambiguous, although such a simple ambiguity is not mentioned elsewhere, and resolution as a declaration would reject most C++ programs. The reason is that the analysis above is purely syntactic. In the production

\[ \text{simple-declaration: declspecifier-seqopt init-declarator-listopt;} \]

the strict syntactic interpretation permits the declspecifier-seq (the type) to be omitted. \( i = 0 \) is a valid form of init-declarator-list. An untyped name alone is therefore syntactically ambiguous. Since §6.8-3 prohibits the use of more than semantic type information to disambiguate, there appears to be a problem. The problem disappears if the constraint in §7-7 is interpreted as a syntactic rather than semantic constraint. The constraint specifies that a declspecifier-seq may only be omitted for function-like declarations.
Incorporating this constraint, together with the constraint on no implicit int functions, eliminates the possibility of the \( P_e \) prefixes.

The constraint that functions are not constructed eliminates the \( J \).

\[
\delta \cap ^{+L} \eta = ^{+L} I^2 \cdot \left\{ F_{de} \right\}^* \cdot A_{de} \cdot \left\{ \theta_e \right\}
\]

The \( I^2 \) covers only the = 0 of a pure-specifier and can be eliminated since there is no D prefix to supply a virtual keyword.

\( \theta_e \) covers only destructor names and conversion-function-ids, which cannot be declared where expression statements are valid.

Functions returning arrays or functions are invalid.

Application of semantic constraints therefore reduces the ambiguity to the more familiar:

\[
\delta \cap ^{+L} \eta = ^{+L} F_{de} \cdot \left\{ \theta_e \right\}
\]

Continuing the analysis to determine the recursive ambiguity. The \( F_{de} \) ambiguity arises when a parameter-declaration-clause is ambiguous with respect to a parenthesised expression-list. This occurs when each parameter-declaration

\[
\pi = D^+ \cdot I^2 \cdot P_d^* \cdot \left\{ \left[ E^? \cdot V^? \cdot F_d \right]^* A_d \cdot \left\{ E^? \cdot V^? \cdot F_d \right\}^* A_d \cdot B \cdot P_d^* \right\} \cdot \left\{ \theta_d \right\}
\]

is ambiguous with respect to the corresponding

\[
\text{assignment-expression} = \eta = \left[ \begin{array}{c} A_v \\ F_e \\ O_s \\ O_p \\ O_i \\ P_e \\ C \\ B \end{array} \right] \cdot \left\{ \theta_e \right\}
\]

The presence of \( D^+ \) in the parameter-declaration would appear to preclude an ambiguity, however \( \sigma_e \subset D^e \epsilon \). Trimming completely unsuitable terms and arranging to exploit the lexical commutation \( D \cdot F \cdot \epsilon = F \cdot D \cdot \epsilon \) gives the solution as
\[
\rho = \pi \cap \eta = D \cdot I^2 \cdot \left\{ F_d \right\}^* \cdot F_d \cdot \varepsilon \cap \left\{ A_e \right\}^* \cdot F_e \cdot \sigma_e
\]

This simplifies to give
\[
\rho = \pi \cap \eta = I^2 \cdot \left\{ F_{de} \right\}^* \cdot F_{de} \cdot \sigma_e
\]
as the form of an argument of \( F_{de} \) that is ambiguously either a parameter-declaration or an expression. This ambiguity is not only recursive, but also exhibits multiplicity in its recursion.

Performing a sanity check: the simplest form is

\( \text{Name}() \)

which is recognisable as an expression involving construction of a \( \text{Name} \). That it is also a parameter-declaration requires understanding of a very dark corner of C++ (§8.3.5-3). The declaration interpretation is of an abstract (unnamed) function taking no arguments and returning \( \text{Name} \). This has no meaning in C++, since functions are not first class entities. Function names are interpreted as pointers to functions and so the example is equivalent to:

\( \text{Name}(*)() \)

5.5.3.3 Expression / Expression ambiguities

The foregoing analysis has taken little account of ambiguity between operators. Operators are excessively overloaded: parentheses variously denote a cast, a function-call or arithmetic grouping. We therefore analyse an assignment expression in terms of its lexical layout eliminating the application operator and using only lexical adjacency. For this purpose we introduce \( \_ \) so that \( a \) denotes a comma separated list of zero or more elements of \( a \). Punctuation such as (, ), [ and ] represent the lexical character.

The lexical production rule for an assignment expression is:

\[
\eta \rightarrow \begin{cases} 
\eta[\eta] & \text{array} \\
\eta(\tilde{\eta}) & \text{call} \\
\eta O_s & \text{suffix} \\
O_p \eta & \text{prefix} \\
(\tau) \eta & \text{explicit-cast} \\
\eta O_i \eta & \text{infix} \\
P \eta & \text{pointer} \\
(\tilde{\eta}) & \text{parenthesis} \\
\theta_e & \text{name} \\
\lambda & \text{value} \\
\sigma_e(\tilde{\eta}) & \text{functional-cast} 
\end{cases}
\]

The use of \([\eta]\) for an array is almost unique to indexing an array and so creates no expression ambiguities. The sole other use occurs in operator \text{new} [ ] and operator \text{delete} [ ] where the absence of \( \eta \) disambiguates.
The () of calls, casts, parentheses and function-casts may create
- parenthesised-call / cast-parenthesis
- parenthesised-binary / cast-unary
- call / functional-cast
ambiguities.
The parenthesised-call / cast-parenthesis ambiguity is
\[
\left( \tilde{\eta}_1 \right) \left( \tilde{\eta}_2 \right) \cap (\tau) (\eta_3)
\]
and the parenthesised-binary / cast-unary is
\[
\left( \tilde{\eta}_1 \right) O_1 \eta_2 \cap (\tau) \left[ \frac{O_p}{P_e} \right] \eta_3
\]
There are ambiguities when \( \eta_2 = \eta_3 \) and \( \eta_1 \cap = \tau \) and \( O_1 \) is +, -, \*, or &. Further
ambiguities exist when \( O_1 \) is ++ or -- and \( \eta_2 \) and \( \eta_3 \) are parenthesised.
The ambiguities arising when \( \eta = \tau \) are analysed in Section 5.5.3.4 as the type-id / assignment-expression ambiguity.
The call / functional-cast ambiguity analysed in Section 5.5.3.5 does not arise
when type-name information is available.
Some suffix operators such as ++ are also prefix operators. No ambiguity can
arise because suffix operators cannot precede a prefix operator.
Some infix operators such as + are also prefix operators, while others such as *
are also pointer operators. No ambiguity arises since, in an expression such as
\[ a \ast \ast \ast b \]
the absence of a further suffix ambiguity ensures that the first * must be infix and
subsequent *'s prefix. However, a conversion-function-id is covered by \( \theta_e \) and may
end in a * creating this further suffix ambiguity.
\[ \&Class::operator int* * *pointer \]
This is resolved by language definition (§12.3.2-4) to maximise the length of the
conversion-function-id: a resolution that can never avoid a subsequent semantic error.
An equivalent ambiguity within a new-expression is similarly resolved (§5.3.4-2).
\( \theta_e \) and \( \lambda \) are independent and cause no ambiguity, beyond those already
discussed.

5.5.3.4 type-id / expression-list ambiguity
A parenthesised-call / cast-parenthesis or parenthesised-binary / cast-unary
ambiguity exists when the
\[
\text{type-id } = \tau = T^+ \bullet P_d^* \bullet \left\{ \left\{ E^2 \# V^2 \# F_d \right\} A_d \right\} \bullet \left\{ \left\{ E^2 \# V^2 \# F_d \right\} A_d \right\} \bullet \varepsilon
\]
of the cast may be confused with the
of the parenthesised call as (a) in
(a) (b)
(a) -b
(a) ++ (b)

The T* term can only match the σ_ε, since σ_ε ⊂ T*ε, leading to the solution

\[ \tau \cap (\mathcal{L} \cdot \eta) = \left\{ A_{de} \right\}^* \cdot F_{de} \cdot \sigma_\epsilon \]

which again makes use of the strange equivalence of a function and pointer to function to determine that type information alone is insufficient to disambiguate

(\text{Class[]}())(x) \quad // \text{Cast to function}

which could be

\text{Class()}.\operatorname{operator}()(x) \quad // \text{Call of operator()}

or

(\text{Class} (*)())(x) \quad // \text{Cast to pointer to function}

A cast to function is not one of the recognised forms of cast enumerated in §5.4, and so the ambiguity has a well-defined semantic resolution, which may be used to avoid the syntactic ambiguity.

### 5.5.3.5 call / functional-cast ambiguity

The call / functional-cast ambiguity arises when the type-name σ_ε in a functional-call can be mistaken for an \textit{assignment-expression}. A type-name is not valid as an \textit{assignment-expression} so there is no call / functional-cast ambiguity when type-name information is available.

### 5.6 Parsing the ambiguities

We have used an extended regular expression notation to derive the declaration and expression ambiguities in C++ syntax, in a way that is not possible with the standard regular expression or C++ grammar notations. We now examine how ambiguities may be resolved in a practical parser.

If the BNF provided as part of the C++ standard is converted directly into a yacc grammar, yacc reports many ambiguities, mostly relating to declarations and expressions. The expression / declaration ambiguities are by far the most serious and much the hardest to resolve.
We now consider the traditional approach to resolving an ambiguity, a multi-pass approach formerly adopted by FOG, and a more efficient superset grammar approach.

5.6.1 Parsing against an ambiguity (the traditional approach)

An ambiguity may be resolved within the grammar, by rewriting the grammar to remove the ambiguity, or by providing assistance in the form of disambiguation tokens from a lookahead parser. The lookahead approach is straightforward, but just redefines the problem as one to be solved elsewhere, potentially using ad hoc code that may be prone to incorrect programming assumptions resulting from the complexity of a recursive ambiguity.

Removing the ambiguities within the grammar is very hard. Given two mutually ambiguous subgrammars \( A \) and \( B \), it is easy to see that the ambiguity is removed by identifying the ambiguity \( A \) and \( B \) comprising all sentences that could form part of \( A \) and could form part of \( B \). The ambiguous case may then be removed from each of the original subgrammars to leave \( \text{Only} A \) and \( \text{Only} B \). The revised grammars comprising \( \text{Only} A, A \) and \( B \) and \( \text{Only} B \) are free from the \( A/B \) ambiguity.

\[
\begin{align*}
\{A\} &= \left\{ \begin{array}{l}
\text{Only} A \\
A \text{and} B \\
A \text{and} B \\
\text{Only} B \\
\end{array} \right. \\
\{B\} &= \left\{ \begin{array}{l}
\text{Only} A \\
A \text{and} B \\
\text{Only} B \\
\end{array} \right. \\
\end{align*}
\]

The analyses in the previous section show how complex grammars could be analysed, enabling \( A \) and \( B \) to be identified from \( A \) and \( B \), and show that in the case of the expression / declaration ambiguity, the ambiguity has to be expressed recursively. Converting \( A \) and \( B \) into BNF rules is relatively straightforward. However \( \text{Only} A \) and \( \text{Only} B \) require a subtraction of formulae and consequently result in very complicated expressions for \( \text{Only} A \) and \( \text{Only} B \). Determining these in the non-recursive context is somewhat daunting. An accurate recursive resolution of expression-that-is-not-a-declaration is a major undertaking.

For the C++ declaration / expression case, the other subtraction leading to declaration-that-is-not-an-expression does not need to be evaluated, because the disambiguation rule (§6.8) mandates that the ambiguity be resolved as a declaration. It is sufficient to parse the declaration unchanged as \( A \) and the expression-that-is-not-a-declaration as \( \text{Only} B \).

Once unambiguous formulae have been identified, they then need to be converted to BNF in a way that does not require more than one token of lookahead. It is not sufficient to convert the formulae independently. They must be converted together so that no shift-reduce or reduce-reduce conflicts are introduced when the parser has seen a partial input that could prefix more than one alternative.

In conventional C++ parsers, the use of type information resolves nearly all ambiguities, so that the shared prefix constraint is the hard problem.

The relatively arbitrary nature of the gcc, CPPP and Roskind grammars suggest that an approximate solution to the above problems was discovered empirically. The problem is too complex to be amenable to an empirical approach. Both the gcc and CPPP grammars are reported to fail to correctly resolve more complicated declaration / expression ambiguities.

In summary, parsing against an ambiguity requires an accurate implementation of the grammar. This is hard to achieve for a deep and recursive ambiguity. [Roskind91] describes his solution as "A LOT of work" and notes that some
ambiguities are resolved prematurely. The gcc implementation was also not easily reached.

5.6.2 Parsing without an ambiguity (the multi-pass approach)

Expressions and declarations are disambiguated by preferring a declaration to an expression whenever there is an ambiguity (§6.8). A two-pass parser can therefore be designed that first parses for a declaration, and if that parse fails, then parses for an expression.

\[
\begin{align*}
A, B
\end{align*}
\]

This makes for a much simpler grammar implementation since no grammar revision is required to remove the ambiguity. We just need to support the ability to perform multiple passes.

Back-tracking in the context of a parser involves examining the input token stream to see whether the stream satisfies a candidate syntax, and if not backing up again to try another candidate.

This practice is common in hand-written parsers, which normally use a top-down left-most reduction at the left policy (LL). Examining the left of a production tends to make premature decisions that then need to be undone.

Use of derivation at the right in an LR parser avoids premature decisions and can make back-tracking unnecessary. Generally, back-tracking is undesirable, since work performed upon each backed-up path is wasted. Well-structured grammar code does not need to back-track.

The standard parser tools do not support back-tracking, and so the implementation of back-tracking presented in Section 5.8 may be novel.

Cost

Accurate determination of the cost of back-tracking requires instrumentation of a parser that adopts both approaches. Such a parser has not been developed, so we can only estimate the likely costs.

Back-tracking incurs three costs:
- marking and unmarking a restart position (always)
- restarting at the mark (only when a back-track necessary)
- wasted analysis effort (only when a back-track necessary)

Maintenance of the marked position need not be particularly costly, if each token is already represented by a polymorphic object, but may be more noticeable if the tokens would otherwise have been acquired directly as a binary stream from a preprocessor.

FOG maintains a garbage collector context at each mark, so re-establishing the mark not only back-tracks on the input context, but also destroys any unwanted objects created to support the failed analysis. This cost will be low, since a failed analysis will normally fail before creating many objects.

The wasted analysis effort will also be small, since an illegal syntax will usually fail after only a few tokens.

Back-tracking incurs no costs for syntax such as selection-statements that can be identified from their first token. Costs remain small for syntax that corresponds to the first analysis alternative. The costs only become significant for expression-statements that closely resemble declaration-statements and so cause significant wasted effort.
The parsing cost is estimated to increase by 20%, which is undesirable for a production compiler, but justifiable in terms of the improved modularity for a research tool.

The approach recommended in the next section reduces this 20% estimate to a negligible level, and so no attempt at accurate measurement has been made.

5.6.3 Parsing with the ambiguity (the superset approach)

A design philosophy of C is the principle that declarations imitate the style of their usage. The declaration syntax is therefore deliberately rather than accidentally and inconveniently similar to the expression syntax. This property can be exploited to develop a superset syntax that encompasses both declarations and expressions.

\[
\begin{align*}
A & \in AorB \\
B & \in AorB
\end{align*}
\]

AorB covers all sentences that satisfy A and all sentences that satisfy B. Deriving AorB involves adding the relevant formulae and pruning any duplication. In order to simplify the grammar we may choose to add further terms to AorB provided we do not introduce any new ambiguities, or at least provided we can resolve any new ambiguities that we do introduce. We require the grammar for AorB to provide cover for at least A and at least B. We do not require precise equivalence.

The consequence of choosing to make AorB larger is that some sentences that were formerly syntax errors, are now accepted by the superset grammar. These sentences should be diagnosed in a later semantic analysis. This is actually beneficial, since the extra sentences that are accepted have a close similarity to legal sentences and so cover likely programming errors. Accepting such errors syntactically improves the likelihood that a diagnostic will report an error that is relevant to the programmers intent. For instance given

```c
typedef type a virtual;
```

many compilers may succeed in diagnosing an illegal typedef. The greater syntactic coverage may allow the compiler to accept the typedef initially but report that virtual is not a legal qualifier for the name of a typedef.

With such a superset grammar there is then no ambiguity, merely a loss of resolution. The loss of resolution can be recovered by semantic processing following the superset syntactic parse.

5.7 The Superset Grammar Approach

The superset grammar approach described in this section comprises two innovations, each of which could in principle be used independently.

Unification of declaration and expression syntaxes provides a solution to the major C++ parsing problem: the declaration / expression ambiguity. The problem ceases to be syntactic. It is deferred to the semantic level where it belongs and is relatively easily resolved.

Elimination of the use of type information avoids the need for potentially infinite lookahead to perform type disambiguation (of p in Section 5.5.3.2), since incorporation of type requires substantial grammar elaboration to handle the undecided lookahead. Removal of type simplifies the grammar, allowing type related ambiguities to be removed from the grammar and deferred for semantic resolution, where they too belong.

We first revisit the ambiguity analysis of Section 5.5.3 to see what problems a lack of type information causes. We then present relevant parts of the superset grammar to show how the superset is implemented. The full superset C++
grammar may be found in Appendix B, and the extended FOG superset in Appendix C.

5.7.1 C++ Ambiguities (without type information)

With type information, $\sigma_e$ is a type-name and is distinct from $\theta_e$ which is a non-type-name. Their C++ usage is distinct:

Without type information there is no distinction, and we just use the superset name $\theta$. This was shown graphically in Figure 5.3 on page 158.

$\theta \supset D \cdot e \cup \theta_e$

The $F_e \cdot \sigma_e$ term may be subsumed by the replicator to give the superset expression.

5.7.1.1 Declaration / Expression ambiguity

The strict ambiguity for a declaration / expression ambiguity (from Section 5.5.3.2) simplifies initially to

and after application of semantic constraints to:

$\delta \cap \{ A_e \} = \{ D^0 \} \cdot \{ P_e \}^+ \cdot \{ F_{de} \}^+ \cdot \{ \theta \}$
with the corresponding recursive parameter ambiguity ultimately simplifying to

$$\pi \cap \eta = I^\gamma \bullet \left\{ F_{de} \right\}^* \bullet A_{de} \bullet F_{de} \bullet \theta$$

These are greater ambiguities but they have the same structure as before. The former ambiguities involving just \( \sigma_e \) now involve \( \theta \). This loss of precision can be recovered as soon as type information is available. Type information is not necessary to identify the syntactical structure in which \( \theta \) is used. The ambiguity may therefore be safely deferred for semantic rather than syntactic resolution.

### 5.7.1.2 Expression / Expression ambiguity

Removal of the distinction between \( \sigma_e \) and \( \theta_e \) requires reassessment of ambiguities related to \( \sigma_e \) and \( \tau \).

The former declaration / declaration ambiguity involving \texttt{operator new[]} now becomes an expression / expression ambiguity as well.

### 5.7.1.3 type-id / expression-list ambiguity

A parenthesised-call / cast-parenthesis or parenthesised-binary / cast-unary ambiguity exists when the

$$
type-id = \tau = T^+ \bullet P_d^* \bullet \left\{ \left\{ E^7 \bullet V^7 \bullet F_d \right\}^* \bullet \left\{ A_d \right\} \bullet \left\{ E^7 \bullet V^7 \bullet F_d \right\} \right\} \bullet \varepsilon
$$

of the cast may be confused with

$$
assignment-expression = \eta = \left[ \begin{array}{c}
A_e \\
F_e \\
O_s \\
O_p \\
O_l \\
P_e \\
C \\
B \bullet \varepsilon^L
\end{array} \right]^* \bullet \left\{ \theta \right\}
$$

The \( T^+ \) term can now match part of \( \theta \), since without type information \( T \bullet \varepsilon \cap \theta \neq \varepsilon \), leading to the solution (for the potential type names)

$$
\tau \cap (T^+ \bullet \eta) = \left[ A_{de} \right]^* \bullet (\theta \cap T \bullet \varepsilon)
$$

indicating that

\( (T^+) + 5 \)

is now ambiguous: is it a cast of \( +5 \) to type \( T \), or the sum of \( T \) and \( 5 \)?

This decision cannot be made without type information. However it can be deferred until type information is available, since an AST node that misleadingly
describes the addition of a type to a value can be detected and corrected to describe the corresponding cast. This error is highly localised unlike the template corrections of Section 5.8.1.

5.7.1.4 Call / functional-cast ambiguity

The call / functional-cast ambiguity arises when the type-name $\sigma_e$ in a functional-call can be mistaken for an assignment-expression. A type-name is not distinct from $\theta$ which is a valid assignment-expression. All forms of functional-cast become ambiguous. The functional-cast is excised from the grammar, with detection of functional-casts deferred until type information is available to determine whether the function-name associated with a call is a type-name or not.

5.7.2 A naive Assignment-Expression / Parameter-Declaration superset

assignment-expression and parameter-declaration occur as part of the recursive ambiguity between an expression-list as a function call argument list and the parameter-declaration-clause of a function-definition. We must find a superset that covers both

$$\pi = D^+ \cdot \{\} * P_d * \cdot \left\{ \begin{array}{c} E^2 \cdot V^2 \cdot F_d \cdot A_d \\ B \cdot P_d^+ \end{array} \right\}$$

and

$$\eta = \left\{ \begin{array}{c} A_e \\ F_e \\ O_s \\ O_p \\ O_l \\ P_e \\ C \\ B \cdot \{\} \end{array} \right\}$$

A very naive common superset for $\omega$ is $\{\pi, \eta\}$.
where
- \( A \) is the superset of \( A_d \) and \( A_e \)
- \( F(\cdot L \cdot \omega) \) is the superset of \( F_d(\cdot L \cdot \eta) \) and \( F_e(\cdot L \cdot \eta) \)
- \( P \) is the superset of \( P_d \) and \( P_e \)

This superset covers \( \varepsilon \) and so introduces numerous ambiguities. For instance a name could be either \( \theta \) or \( D \cdot \), and an infix operator with \( \varepsilon \) as its left operand is indistinguishable from the equivalent prefix operator. A much tighter superset is required.

5.7.3 The Superset

Definition of a superset that covers a parameter-declaration and assignment-expression requires that their component terms also represent corresponding supersets. We therefore present the component supersets in increasing order of complexity before finally reaching the generalised parameter-declaration and showing that it provides superset grammar coverage.

5.7.3.1 Generalised Name

The ambiguity in

\[
\text{prefix ( b )}
\]

causes extreme parsing difficulties because the C++ grammar prepends the prefix to the parenthesis for two different reasons, only one of which is guaranteed to have parentheses present:
- a prefix name is associated with parentheses for a function call
- a prefix name is associated with an optionally parenthesised declarator for a simple-declaration

One of these alternatives must be eliminated to resolve the parsing difficulty. Eliminating a function call is undesirable since parsing of function arguments occurs in the midst of productions that enforce arithmetic precedences. Eliminating the name from the decl-specifier-seq prefix of declarations is possible.

The solution is to maximise the parsed length of any word-like sequence before associating punctuation. Therefore

\[
\text{extern int f()}
\]

is parsed so that \text{extern int f} is parsed in its entirety before the parentheses are applied. This avoids the problem of whether a prefixed name may be followed by a parenthesis or not, since a name is only prepended in one place. The cascade of names is parsed first. As a result of this, the parse trees for
int (var)
Class(arg)
Class(int)

are all the same. Semantic processing must use type information to separate the possibilities and recognise the equivalence of

int (var);
int var;

The template keyword affects the way in which a name is used. It must therefore bind to that name. template is therefore always parsed close to the name. This potentially causes a conflict with an explicit-instantiation which provides an external binding of the template keyword.

explicit-instantiation: template declaration

The C++ grammar therefore suggests that

template int X<int>::f();

be parsed as

f
X<int>::f
X<int>::f()
int X<int>::f();
template int X<int>::f();

in which each subsequent line represents the increased knowledge resulting from reducing a parsing rule.

In FOG, names and the template keyword are resolved with high priority to avoid ambiguities, the reduction order is

f
X<int>::f
int X<int>::f
template int X<int>::f
template int X<int>::f()
template int X<int>::f();

The conflict is resolved by ensuring that every valid declaration that can participate in an explicit-instantiation incorporates the template prefix. In practice this means that every prefixing rule that could form part of a declaration in the generalised expression must apply a template prefix if it applied any other prefix. Parsing of an explicit-instantiation is therefore subsumed by parsing a declaration.

In the grammar analysis, a maximised name is denoted by D+, which is convenient for analysis but does not perform very much of a syntactic breakdown. Classifying the components of D as

u a user-defined name (e.g. ::name or operator const char *)
or class X (...) already including an optional template prefix
b a built-in type name (e.g. short)
t the template keyword
q anything else (e.g. const or extern or virtual)

D+ is parsed (using regular expression syntax) as

(t*q)* ((t*b(b|q)*?) (uq*(b(b|q)*?))? | t*b(b|q)*)

rather than

(u|b|q|t)+

All possibilities are parsed and the requirement to gather template prefixes is observed. The b(b|q)* grouping captures a complete (multi-word) built-in type name together with all interspersed and trailing qualifiers. The uq* similarly
captures a single word user-defined name together with all trailing qualifiers. An arbitrary mix of names is permitted recognising that two built-in type-names cannot be adjacent. This grouping ensures that only one AST node need be created for each name and that trailing qualifiers attach to their preceding name. The \((t\cdot q)^*\) prefix associates any prefix qualifiers with the first name.

The leading \((t\cdot q)^*\) is separated and not implemented as part of a D+ parse. This exploits the semantic constraint that a name must contain at least a user-defined or built-in type name, and avoids ambiguities when a generalised name:

- follows a cast generalised with a trailing \textit{cv-qualifier-seqopt} to cover an abstract function declarator
  \begin{verbatim}
  (cast) const p
  \end{verbatim}
- follows a cv-qualified pointer
  \begin{verbatim}
  int * const p
  \end{verbatim}
- is used with the FOG negated qualifier extension
  \begin{verbatim}
  static int p;
  \end{verbatim}

The omitted prefix \((t\cdot q)^*\) term is only valid as part of a \textit{decl-specifier-seq} which occurs at the start of certain declarations. The missing specifiers are therefore parsed as a prefix once all other ambiguities have been removed:

\begin{verbatim}
simple_declaration: ....
    | init_declaration_list ;
    | decl_specifier_prefix simple_declaration
\end{verbatim}

A related complexity arises with pointers to members

\begin{verbatim}
Class :: *
\end{verbatim}

since the following is syntactically valid

\begin{verbatim}
Type Class::* p = 0
\end{verbatim}

In order to pursue the same policy of maximising the name to avoid shift-reduce conflicts, the parse for the pointer scope absorbs all the preceding name components. This must of course eventually be sorted out by the semantic processing, but differs little from the problem of resolving

\begin{verbatim}
int ** p = 0
\end{verbatim}

which the superset parse identifies as an assignment of zero to the product of the name \texttt{int} and the dereference of \texttt{p}. This might appear to be a severe misparse, but actually corresponds to an economy of AST nodes. The pointer-to declarator node does not need to exist. Its functionality is folded into the superset multiplier node, which now has one behaviour for types and another behaviour for values.

### 5.7.3.2 Generalised Array

The two forms of array suffix: \(A_d(\kappa^\pi)\) and \(A_d(\cdot\pi\cdot\eta)\) are generalised to \(A(\cdot\pi\cdot\eta)\).

### 5.7.3.3 Generalised Parentheses

The two forms of function suffix: \(E^\pi\cdot V^\pi\cdot F_d(\cdot\pi\cdot\eta)\) and \(F_0(\cdot\pi\cdot\eta)\) are generalised to \(E^\pi\cdot V^\pi\cdot F(\cdot\omega)\) where \(\omega \supseteq \pi \cup \eta\).

The precedence enforcing parentheses are generalised from \(B^\pi\cdot L\) to \(E^\pi\cdot V^\pi\cdot B^\pi\cdot L\). Although not necessary to unify declarations and expressions, generalisation of a cast is necessary to avoid shift-reduce conflicts. \texttt{C} is therefore replaced by \(E^\pi\cdot V^\pi\cdot C\).

In order to parse \texttt{delete[]} followed by an unparenthesised expression, \(C(\tau)\cdot \alpha\) is further generalised to cover \([\eta^\pi] \alpha\) as well as \((\tau) \alpha\).
5.7.3.4 Generalised pointers

The `ptr-operator` is not generalised, however the usage of `*` as a binary operator is replaced by `star-ptr-operator` in order to accept a `cv-qualifier-seq-op` following a `*` in a `multiplicative-expression`. Additionally any `decl-specifier` preceding the scope of a pointer to member is associated with the scope, thereby avoiding conflicts and the need to introduce another infix expression operator.

```c++
unary_expression:
  postfix_expression
  | "++" cast_expression
  | "--" cast_expression
  | ptr_operator cast_expression
  | suffix_decl_specified_scope star_ptr_operator cast_expression

multiplicative_expression:
  pm_expression
  | multiplicative_expression star_ptr_operator pm_expression
  | multiplicative_expression '/' pm_expression
  | multiplicative_expression '%' pm_expression

star_ptr_operator:
  '
  | star_ptr_operator cv_qualifier
```

5.7.3.5 Generalised `primary-expression`

In order to cover some forms of `abstract-declarator` as expressions and thereby avoid shift-reduce conflicts, a `primary-expression` is extended to cover an abstract array `A*e` in addition to an abstract function covered by the parenthesis generalisation to `E?*V?*B*"L*w`. The same

```
\begin{array}{c}
A \\
E? \cdot V? \cdot B \cdot "L"
\end{array}
```

term is therefore used as an expression (to support `abstract-declarator`), as an expression prefix (to support a cast) and as an expression suffix (to support function parameters). Use of precisely the same syntax avoids shift-reduce conflicts.

```c++
primary_expression:
  literal
  | "this"
  | ...
  | abstract_expression

abstract_expression:
  parenthesis_clause
  | [ expression_opt ']' 
  | "template" abstract_expression

parenthesis_clause:
  parameters_clause cv_qualifier_seq_opt exception_specification_opt

parameters_clause:
  '(' parameter_declaration_clause ')' 

5.7.3.6 Generalised `assignment-expression`

In order to provide complete coverage of a declarator initializer, the right-hand expression of an `assignment-expression` using = is extended to accept `{ }` forms.

```c++
assignment_expression:
  conditional_expression
  | logical_or_expression assignment_operator assignment_expression
  | logical_or_expression '=" braced_initialiser 
  | throw_expression

braced_initializer:
  '{' initializer_list '}'
  | '{' initializer_list ',' '}'
  | '{' '}'
```
The overall form of the generalised assignment-expression is

\[
\chi = \begin{pmatrix}
A \\
E^\gamma \cdot V^\gamma \cdot F \\
O_s \\
O_p \\
O_i \\
P \\
E^\gamma \cdot V^\gamma \cdot C \\
E^\gamma \cdot V^\gamma \cdot B \\
\end{pmatrix}^* \\
\begin{pmatrix}
D^+ \cdot \varepsilon \\
\lambda \\
A \\
E^\gamma \cdot V^\gamma \cdot B \\
L \cdot D^+ \cdot I^\gamma \cdot P_d^+ \cdot \varepsilon \\
\end{pmatrix}
\]

\(\chi \triangleright \eta\) since \(A \triangleright A_e, F \triangleright F_e, P \triangleright P_e, D^+\varepsilon \triangleright \emptyset\).

5.7.3.7 Generalised parameter-declaration

Adding one extra term to \(\chi\) provides coverage for \(\pi\), and ensures that \(\omega \triangleright \chi \triangleright \eta\). The additional term defines \(\omega\) as a generalised parameter-declaration

\[
\omega = \left\{ \chi \right\}
\]

The extra \(D^+P_d^+I^\gamma\varepsilon\) exhibits significant prefix ambiguity with a binary expression. It is only the \(\varepsilon\) at the right-hand side that disambiguates

\[a \text{ **** b},\]

Implementation of this term therefore re-uses intermediate expression productions to avoid shift-reduce conflicts, and consequently covers many meaningless sentences.

**abstract_pointer_declaration:**

- ptr_operator_seq
  - multiplicative_expression star_ptr_operator ptr_operator_seq.opt

**abstract_parameter_declaration:**

- abstract_pointer_declaration
  - and_expression \(\varepsilon\)
    - and_expression \(\varepsilon\) abstract_pointer_declaration

**special_parameter_declaration:**

- abstract_parameter_declaration
  - abstract_parameter_declaration \(==\) assignment_expression

**parameter_declaration:**

- assignment_expression
  - special_parameter_declaration
    - declSpecifier_prefix parameter_declaration

5.7.3.8 Coverage of the generalised parameter-declaration

It must be shown that \(\pi \subseteq \omega\) where
Meta-compilation for C++

Parsing

\[
\omega = \left\{ \begin{array}{l}
A \\
E^2 \cdot V^2 \cdot F \\
O_s \\
O_p \\
O_i \\
P \\
E^2 \cdot V^2 \cdot C \\
E^2 \cdot V^2 \cdot B \ast L \end{array} \right\}^* \\
\left\{ \begin{array}{l}
D^+ \cdot \varepsilon \\
\lambda \\
A_d \cdot \varepsilon \\
E^2 \cdot V^2 \cdot B \ast L \ast D^+ \ast I^2 \ast P_d^+ \ast \varepsilon \\
D^+ \ast I^2 \ast P_d^+ \ast \varepsilon \\
\end{array} \right\}
\]

\[
\pi = D^+ \ast I^2 \ast P_d^+ \ast \left\{ \begin{array}{l}
E^2 \cdot V^2 \cdot F_d^* \\
A_d \\
B \ast P_d^* \end{array} \right\} \cdot \left\{ \begin{array}{l}
E^2 \cdot V^2 \cdot F_d \\
A_d \\
B \ast P_d^+ \end{array} \right\} \cdot \varepsilon
\]

To keep the equations more manageable:

\[
\xi = \left\{ \begin{array}{l}
E^2 \cdot V^2 \cdot F_d^* \\
A_d \\
B \ast P_d^* \end{array} \right\} \cdot \left\{ \begin{array}{l}
E^2 \cdot V^2 \cdot F_d \\
A_d \\
B \ast P_d^+ \end{array} \right\} \cdot \varepsilon
\]

\pi may be split to create 4 sub-problems:

\[
\pi = \left\{ \begin{array}{l}
D^+ \ast P_d^+ \ast I^2 \ast \xi \\
D^+ \ast I^2 \ast \xi \\
D^+ \ast I^2 \ast \xi \\
D^+ \ast P_d^+ \ast I^2 \ast \xi \\
\end{array} \right\}
\]

each of which will be shown after first showing that pointers to \(\xi\) are covered.

\(P_d^+ \ast I^2 \ast \xi \subset \omega\)

Of the four alternatives offered by the right hand term of \(\xi\), the top one and bottom two are clearly covered, since \(I \subset O_i, P_d \subset P, \theta_d \subset D^+ \varepsilon\). The second may be written more fully as
and then rewritten

\[
P_d \ast l^2 \ast \left[ \begin{array}{c} E \ast V \ast F_d \\ A_d \\ B \ast P_d \ast \end{array} \right] \ast \cdot E \ast V \ast F_d \ast (L \ast \omega) \ast e
\]

and then expanding \( \omega \) gives

\[
P_d \ast l^2 \ast \left[ \begin{array}{c} E \ast V \ast F_d \\ A_d \\ B \ast P_d \ast \end{array} \right] \ast \cdot \left\{ E \ast V \ast B \ast L \ast \chi \right\}
\]

Examining top and bottom right-hand terms shows them to be covered by \( \omega \).

1) \( D^+ \ast l^2 \ast e \subset \omega \)

This term is covered by \( O_1 \ast D^+ \ast e \) since prefix and suffix operators commute and we generalised the right-hand side of an assignment to support the \{ \} form of declaration initializer.

2) \( D^+ \ast P_d \ast l^2 \ast e \subset \omega \)

This is covered by the extra term added for use of a generalised parameter-declaration rather than a generalised assignment-expression.

3) \( D^+ \ast P_d \ast l^2 \ast e \subset \omega \)

Taking the lexical perspective for ordinary pointers and references: the first \( P_d \) in a term which could be a multiplier in a declaration of the form

\[
\text{static long int } * * a = 0;
\]

The problem

\[
D^+ \ast P_d \ast P_d \ast l^2 \ast e
\]

can be written, after introducing lexical separation around the first \( P_d \), as

\[
D^+ \ast e \quad P_d \ast e \quad P_d \ast l^2 \ast e
\]

which is covered by

\[
D^+ \ast e \quad O_l \ast e \quad P_d \ast l^2 \ast e
\]

since \( P_d \) is covered by \( O_l \), and the possibility of a trailing cv-qualifier-seq on \( P_d \) is covered by the generalisations to unary-expression and multiplicative-expression. Introduction of the dyadic \( O_l \) partitions the problem into the two smaller problems:

\[
D^+ \ast e \subset \omega \quad \text{and} \quad P_d \ast l^2 \ast e \subset \omega.
\]

It is clear that \( D^+ \ast e \subset \omega \), and we have already shown that \( P_d \ast l^2 \ast e \subset \omega \).

For pointers to members such as

\[
\text{extern int Class:: * p = 5}
\]
the generalised form of pointer P subsumes the $D^+$ prefix leaving the problem
$P_d^+ \in \omega$ which has already been shown.

4) $D^+ \in \omega$

This may be shown by considering the three alternative locations for the left-most occurrence of a B in:

$$D^+ \cdot \mathcal{I} \cdot \left[ \begin{array}{c} E^2 \cdot V^2 \cdot F_d \\ A_d \\ B \cdot P_d^+ \end{array} \right] \cdot \left[ \begin{array}{c} \theta_d \\ \left[ \begin{array}{c} E^2 \cdot V^2 \cdot F_d \\ A_d \\ B \cdot P_d^+ \end{array} \right] \cdot \varepsilon \right] \subseteq \omega$$

There may be no occurrence of a B:

$$D^+ \cdot \mathcal{I} \cdot \left[ \begin{array}{c} E^2 \cdot V^2 \cdot F_d \\ A_d \\ B \cdot P_d^+ \end{array} \right] \cdot \left[ \begin{array}{c} \theta_d \\ \left[ \begin{array}{c} E^2 \cdot V^2 \cdot F_d \\ A_d \\ B \cdot P_d^+ \end{array} \right] \cdot \varepsilon \right] \subseteq \omega$$

This is covered since the absence of prefix operators allows the $D$ prefix to be combined with the $\theta_d$ or $\varepsilon$.

There may only be a B in the final replicator.

$$D^+ \cdot \mathcal{I} \cdot \left[ \begin{array}{c} E^2 \cdot V^2 \cdot F_d \\ A_d \\ B \cdot P_d^+ \end{array} \right] \cdot \varepsilon \subseteq \omega$$

which may be rewritten as

$$\mathcal{I} \cdot \left[ \begin{array}{c} E^2 \cdot V^2 \cdot F_d \\ A_d \\ B \cdot P_d^+ \end{array} \right] \cdot F(P_d^+ \cdot \varepsilon) \cdot D^+ \cdot \varepsilon \subseteq \omega$$

which is satisfied since the function argument $P_d^+ \varepsilon \subseteq \omega \subseteq L^*\omega$.

There may be a B in the first multiplier

$$D^+ \cdot \mathcal{I} \cdot \left[ \begin{array}{c} E^2 \cdot V^2 \cdot F_d \\ A_d \\ B \cdot P_d^+ \end{array} \right] \cdot \xi \subseteq \omega$$

which may be rewritten as

$$\mathcal{I} \cdot \left[ \begin{array}{c} E^2 \cdot V^2 \cdot F_d \\ A_d \\ B \cdot P_d^+ \end{array} \right] \cdot F(P_d^+ \cdot \xi) \cdot D^+ \cdot \varepsilon \subseteq \omega$$

which is covered since the recursive problem $P_d^+ \cdot \xi \subseteq \omega \subseteq L^*\omega$ has already been shown.

5.7.4 Ambiguities in the superset grammar

The complete C++ grammar implemented using the superset approach is provided in Appendix B. Processing that grammar through bison and yacc provides a rigorous check of the ambiguities, which are described in Appendix F.2.
The overall accuracy of the grammar depends on transcription errors in the conversion of the textual exposition of Annex A of the standard [C++98] to an executable yacc grammar, and the number of changes required to resolve ambiguities. The grammar is similar to the standard and so relatively easily checked in conjunction with the justification of the superset.

The more substantial changes to resolve ambiguities are based on establishing a superset of the declaration and expression syntax. The ability of the superset to parse the composite declaration and expression syntax was shown in the previous section. Much simpler proofs are used to show that the same or a slightly tailored superset solves other ambiguity problems. Restructuring of a few productions is necessary to remove or at least resolve shift-reduce conflicts.

Provided that the syntactic parse discards no information, ambiguities that are not resolved syntactically can be resolved semantically. This is easily achieved by ensuring that each reduction fully parameterises each AST node that is created.

A discussion of the resolution of the practical difficulties associated with the detailed C++ syntax may be found in Appendix F.2 including details of the new ambiguities resulting from the superset approach in Appendix F.2.6. The semantic processing required to recover from the reduced syntactic resolution is described in Appendix F.3.

5.8 Back-tracking

Introduction of back-tracking into an LALR parser grammar in order to support a reparse against an alternate syntax proves to be remarkably easy. The author is not aware of any other implementation that back-tracks, however tools such as yacc are in such widespread use by 'imaginative' programmers that it is unlikely that the approach is totally new.

We first show how a linear search of alternative syntaxes may be made, as was required to resolve the declaration/expression ambiguity in the earlier multi-pass grammar.

We then show how a binary tree search of alternative syntaxes can be orchestrated in order to resolve the template/arithmetic ambiguity for the superset grammar.

5.8.1 Linear search in yacc

The multi-pass FOG productions to perform a linear search through the declaration and expression ambiguities of a statement resemble

```
statement: mark declaration_statement unmark
        | mark remark expression_statement unmark
        | mark remark error ';' unmark
mark:    /* empty */  { push_input_context_marker();  }
remark:  error        { rewind_input_context_to_marker();  }
unmark:  /* empty */  { pop_input_context_marker();  }
```

in which

- /* empty */ is a comment to highlight use of the ε terminal
- {} surrounds the code invoked when a rule is reduced.
- error is a special error recovery token generated by the parser when an inconsistent token is encountered.

The mark rule places a marker on the input token sequence, remark back-tracks and starts another attempt to parse the same sequence, unmark removes the marker but does not affect the token stream. In practice, statements nest, and this approach is used to resolve other parsing problems too, so mark pushes onto and unmark pops from a stack of marked positions.
The operation of the parser requires that all three alternatives start by shifting and then reducing the mark to ensure that the parser generator has a common prefix, and that the context stack is consistently maintained. The first alternative then attempts to parse the input token sequence as a declaration-statement and unmarks if that parse succeeds. The other two alternatives are inactivate until an error occurs, as will be the case if the declaration-statement parse fails. Reduction of the remark rule that handles the error rewinds the input context back to the mark so that an attempt at an expression-statement parse occurs. If this also fails, the further error of the third alternative is satisfied and error recovery proceeds by discarding tokens until a semicolon is encountered.

Successful parsing of the first alternative therefore occurs without the use of any error tokens. Successful parsing of the second alternative occurs after a single error. Only after two errors does a proper error recovery get activated as the third alternative.

This approach directly implements the disambiguation rule (§6.8-1): if it could be a declaration, it is (first alternative), otherwise it's an expression (second alternative). However the parse is a syntactic one. If it looks like a declaration, it is parsed as a declaration, otherwise it is parsed as an expression. This is not a problem when accurate type information is available, however we are advocating parsing without type information and so those ambiguities that require type information are resolved in favour of declarations (because declarations are parsed in the first pass). Expressions that are misparsed because they look like, but cannot be, declarations must be corrected once type information is introduced in subsequent semantic processing. The nature of these expressions is discussed in Appendix F.2.5 and their resolution in Appendix F.3.1.

Code to detect the misparse has the full construct available as AST nodes, avoiding the difficult partial context and lookahead problems that occur while trying to disambiguate in the grammar. Code to correct the misparse performs a localised tree rearrangement with many of the misparsed declaration AST nodes re-usable as nodes in the corrected expression tree.

The approach is relatively easy to implement in the parser, but requires the input token source to support marking, and back-tracking to specific positions. In practice there is a potential for errors in the grammar. Failure to pair mark and unmark leads to stack drift that materialises as an unhelpful parsing failure with a generally confusing diagnostic. Parsing control actions are transparent to the parse table generator, and so unnecessary marks may be omitted:

```markdown
statement: selection_statement
 | mark declaration_statement unmark
 | mark remark expression_statement unmark
 | mark remark error ';' unmark

selection_statement:
  if ( condition ) statement
  if ( condition ) statement else statement
  switch ( condition ) statement
```

A statement starting with an if keyword can only be a selection-statement, since if occurs nowhere else in the grammar. A selection-statement can therefore be parsed without marking the context for backtracking. This works because the parser generator uses one token of lookahead. It may therefore examine the token logically following the mark before deciding which parsing alternatives to retain. This avoids incurring the cost of back-tracking. The corollary is that the marked position in the input token sequence must be adjusted to ensure that the lookahead token is made available following a back-track. A more unpleasant corollary occurs if the omission of a mark is a programming error. In the above example, if a declaration-statement could start with an if, the ambiguity would be diagnosed by the parse generator as a shift-reduce conflict. However if the expression-statement could start with an if, no diagnostic results, the generated
The back-tracking approach is able to resolve the type-name or identifier ambiguity adequately because there is very little syntactic difference: both are names. Most ambiguities are correctly resolved in favour of the declaration.

**Back-tracking for templates**

Back-tracking is less well-suited to resolving the template-name ambiguity, since in one case the syntax involves what should be paired brackets

```
template_name < a , b > - 5 // template_name<a,b> - 5
```

and in the other case an infix operator

```
non_template_name < a , b > - 5 // non_template_name<a , b>-5
```

The ASTs for the two alternatives are rather different, the error in the tree structure is poorly localised, and grows as the number of ambiguous <'s increase. However, it is not resolvable by back-tracking to try an alternative restarting at the <, because the template name may be parsed successfully

```
template_name < a > b [ 5 ] // Ok
```

```
template_name < a > b [ 5 ] + 7 // Syntax error
```

```
on_template_name < a > b [ 5 ] + 7 // Ok
```

and consequently reduced and removed from the parser stack before a syntactic contradiction is detected.

For back-tracking to succeed, the entire expression and more generally the entire declaration or statement, must be retried to test each template / non-template hypothesis. This requires a search of the binary tree of alternate hypotheses.

Back-tracking for template-names is perhaps not appropriate for a C++ parser that performs semantic interpretation. Relatively loose coupling between syntactic and semantic processing will suffice to provide the information needed for syntactic analysis, since template-names cannot be introduced in mid-statement.

Back-tracking for template-names may be appropriate for C++ parsers that do not need full program comprehension. For instance, the relatively rare errors from a template misparse need not render a pretty printing program unusable.

However back-tracking for templates is unavoidable in FOG, since semantic information cannot be provided.

### 5.8.2 Binary tree search in yacc

The lack of context during meta-programming described in Section 4.2.3 mandates parsing without template information, and so a search of the binary tree of possible parses is required.

The problem to be solved is this: given a declaration or statement containing an arbitrary number of occurrences of an *identifier* followed by a <, find a permutation of the template/arithmetic interpretations for each occurrence such that a syntactic parse can succeed.

The solution may be expressed as a boolean vector whose length is the number of *identifier* < occurrences, with a false value for each template verdict and a true value for each arithmetic verdict.

The parser interacts with the solution in two contexts. An outer context sequences the search, advancing in depth first fashion through the solutions. Interaction occurs through

- `start_search` to mark the current context and create the first solution
- `advance_search` to restore the marked context and select the next solution

29-June-2001
end_search to accept the current (or no) solution and clean up
(Solutions are maintained on a stack, so that nested problems can be solved.)
The outer context of course has no knowledge of the problem depth, which is
maintained by the inner context: where the identifier < occurs.
For the first solution, and for deeper exploration of later solutions, each encounter
of identifier < establishes a deeper problem and so the encounter lengthens the
solution vector and enters a false verdict making a presumption of template usage.
If a solution attempt fails, advance_search is invoked and the solution vector is
advanced to the next candidate solution by binary addition at the deep end of the
vector with ripple carry to the shallow end. The current depth is then reset to zero
ready for the next pass, during which the depth is incremented as each identifier < is re-encountered. Each re-examined identifier < therefore responds according
to the template/arithmetic behaviour appropriate to the candidate solution.

Outer context

The search is orchestrated by marking the input token context at the start of each
declaration or statement, and associating a binary tree search context with that
mark. The grammar

\[
\text{compound-statement:} \quad (\text{statement-seq}_{\text{opt}})
\]

\[
\text{statement:} \quad \text{control-statement}
\]

is implemented as

\[
\begin{align*}
\text{compound-statement:} & \quad \{ \text{statement-seq}_{\text{opt}} \} \text{'} \{ \text{statement-seq}_{\text{opt}} \text{ looping-statement '}' bang error '}' ; \}
\text{statement-seq}_{\text{opt}}: & \quad / \text{ empty } / \text{ statement-seq}_{\text{opt}} \text{ looping-statement }
\text{statement-seq}_{\text{opt}} \text{ looping-statement '}' bang error '}' ; \}
\text{looping-statement:} & \quad \text{start_search looped statement } \{ \text{end_search}() ; \}
\text{looped-statement:} & \quad \text{statement}
\text{looped-statement:} & \quad \text{advance_search '+' looped-statement}
\text{looped-statement:} & \quad \text{advance_search '-'}
\text{statement:} & \quad \text{control-statement}
\text{advanced_search:} & \quad \text{error} \{ \text{yyerrok; yyclearin; advanced_search();} \}
\text{bang:} & \quad / \text{ empty } / \{ \text{BANG();} \}
\text{start_search:} & \quad / \text{ empty } / \{ \text{start_search();} \}
\]

Usages of statement are served by looping_statement which organises the binary
tree search around statement.
The search iteration is managed by the three functions start_search(),
advance_search() and end_search() invoked from the action routines.
start_search() and end_search() occur as the pre and post actions that
looping_statement imposes upon looped_statement.
Retries are triggered by the error token, which invokes advance_search to
rewind the context for the next attempt.
In order to avoid an infinite loop, advance_search needs a way to return a value
to the parser to signal whether to continue looping, or to finish the loop. It might
seem that an error token could terminate the loop, but the error token is already
being used to sequence the loop, so another approach is required.
advance_search() injects either a '+' token or a '-' '}' token sequence into
the source from which yylex() reads tokens, and ensures that one is fetched by
clearing out any lookahead with yyclearin.
The third solution is therefore accepted after the parser has apparently parsed the rule:

```c++
start_search advance_search '+' advance_search '+' statement { end_search(); }
```

An unsuccessful attempt may have the apparent rule:

```c++
start_search advance_search '+' advance_search '-' { end_search(); }
```

Re-use of the error token enables the loop to be realised, however the unsuccessful parse has to be successful in order to avoid making double use of the error token. The extra '//' token is therefore injected to cause a guaranteed regeneration of the error, since '//' is reserved for use by the C preprocessor and so never occurs after preprocessing. In order to give a useful diagnostic, the injected '//' is caught in the sequence

```c++
... looping_statement '//' bang error ')'; UNBANG("Bad statement-seq."); }
```

The bang production is empty but has an action routine that interacts with an error message handler to ensure that users don’t see a "parser error" diagnostic from each failed parsing attempt. There are no productions able to successfully handle any tokens following the injected '//' so the parser is forced to start error recovery, for which it finds continuation of the above possible. Error recovery continues by discarding tokens until resynchronisation occurs at the next '}', at which point the UNBANG macro interacts once again with the error message handler to restore the previous behaviour and emit an appropriate message, together with helpful context information.

Generation of a good quality diagnostic requires the diagnostics associated with the most-nearly-successful loop iteration to be cached during the loop for emission at the end of the loop. The most-nearly-successful metric is conveniently determined as the largest number of tokens parsed prior to the error.

**Inner context**

The domain of the search cannot be directly known by the loop management productions. It is maintained as a side effect of identifier parsing.

```c++
<table>
<thead>
<tr>
<th>id</th>
<th>identifier %prec SHIFT THERE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>identifier template_test '+' template_argument_list '&gt;'</td>
</tr>
<tr>
<td></td>
<td>identifier template_test '-'</td>
</tr>
<tr>
<td></td>
<td>template_id</td>
</tr>
</tbody>
</table>
```

```c++
<table>
<thead>
<tr>
<th>template_test</th>
<th>'&lt;' { template_test(); }</th>
</tr>
</thead>
<tbody>
<tr>
<td>template_id</td>
<td>&quot;template&quot; identifier '&lt;' template_argument_list '&gt;'</td>
</tr>
</tbody>
</table>
```

A template ambiguity arises, whenever an `identifier` is followed by a `<` without a template prefix. The ambiguity is resolved, in the absence of template information, by the prevailing state of the binary tree search. This state is determined by the `template_test()` routine which injects '+' token to select a template interpretation or a '-' '<' token sequence for an arithmetic interpretation.

The `template_test()` rule introduces a shift-reduce conflict between the usage in the expression syntax and the usage presented above. This conflict is resolved by the `%prec` to force all occurrences of `identifier` < to take the test. When the test injects a '+', parsing continues by analysing template arguments. When the test injects a '-' and '<', the '-' satisfies the parse for an `identifier` and the extra '<' restores the token required to proceed with the arithmetic interpretation.

On the first traversal of source tokens, `template_test()` increases the binary tree search depth and selects the template hypothesis. On subsequent traversals, `template_test()` signals the prevailing binary tree search hypothesis.

---

3. Any pair of tokens other than `error` could be used since the injected token is used solely to distinguish two forms of `advance_search` in the grammar.
It is rarely necessary to perform the full binary tree search, since a branch and bound can exploit early failure and avoid searching other hypotheses that share the same failing prefix.

There is unfortunately no guarantee that the first accepted hypothesis is correct, and so the subsequent semantic analysis must be prepared to reorganise the AST to accommodate errors in either direction.

It should be noted that there is no error handling for template arguments, so that a template argument parse failure propagates onwards enabling the statement level binary tree search to poll syntax alternatives.

Cost

This algorithm clearly has exponential complexity with respect to the number of ambiguous <'s in a statement, although this complexity does not arise in practice as is demonstrated by applying the grammar of Appendix C to three large bodies of code to determine its cost. The results are summarised in Table 5.5.

<table>
<thead>
<tr>
<th>Code body</th>
<th>Preprocessed Lines</th>
<th>Statements</th>
<th>identifier &lt; statements</th>
<th>% ambiguous</th>
<th>Back-tracks</th>
</tr>
</thead>
<tbody>
<tr>
<td>fog</td>
<td>88133</td>
<td>38470</td>
<td>290</td>
<td>0.75</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>product</td>
<td>119383</td>
<td>44353</td>
<td>1652</td>
<td>3.7</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>gcc</td>
<td>426797</td>
<td>127366</td>
<td>2099</td>
<td>1.7</td>
<td>29 2047 7 21 1 2 2 1</td>
</tr>
</tbody>
</table>

Table 5.5 Back-tracking costs

The fog code is the entire C++ source code for FOG. Fewer than 1% (290 out of 38470) of statements contained an identifier followed by a <, and a consistent syntax for these was usually (166 out of 290) found without back-tracking by assuming the template form. A consistent syntax was always found after one back-track. The comparatively low proportion of ambiguous statements is due to a coding habit that hides templated lists and pointers behind typedefs.

The product code is the entire C++ source code for a proprietary product. A higher proportion but still fewer than 4% of statements contained an identifier followed by an <, and a consistent syntax for these was almost always found after one back-track. A consistent syntax was always found after three back-tracks.

The gcc code comprises almost the entire source code of the gcc compiler. This is C code and so gives worst case performance under the back-tracking presumption to try template syntax first. Nearly 2% of statements were ambiguous. 29 statements were incorrectly resolved as templates without back-tracking. 2047 statements were correctly resolved after the back-track necessary to change the template presumption. The higher number of back-tracks do not indicate the correctness of their parsing conclusion. However since the maximum depth of the binary tree search was 6, it is clear that the use of branch and bound in the search ensured that the 64th solution was found in at most 8 tries.

The C++ examples use templates but predate the Standard Template Library [C++98]. Although STL code may well use more templates, the parsing is likely to be more accurate since the current implementation assumes a template resolution and only retries when that is inconsistent. The gcc code without any templates is therefore a representative worst case.

These results confirm the intuitively expected behaviour. Arithmetic expressions tend to have at most one relational operator, and so favouring template usage
when syntactically consistent results in few expressions that need semantic correction.

The exponential search complexity does not arise in practice. From the C++ results, we may estimate that back-tracking may occur for 50% of ambiguous statements, which appear to be about 2% of all statements. From the gcc results we may estimate that a wrong parse is found 1% of the time. The syntactic cost is therefore an extra 1% of syntactic statement analysis. Estimating the cost of an aborted partial analysis at half the cost of a successful complete analysis, we may estimate that the net semantic cost involves the correction of a template misparse for 0.01% of statements.

5.9 FOG grammar

The full FOG grammar in Appendix C uses similar techniques to those for the simpler C++ grammar in Appendix B.

In addition to the use of back-tracking in the superset grammar to solve the C++ parsing problems
- template name (Appendix F.2.1.3 and Appendix F.2.1.4)
- bit field or inheritance (Appendix F.2.4.1)

the back-tracking approach is also used to solve the following parser problems
- "old-style" C (type I) function declarations(Appendix F.2.4.2)

Multi-pass implementation

The earlier multi-pass implementation of FOG parsed first for declarations and then back-tracked for expressions. Further nested back-tracking was used to resolve
- cast ambiguities (Appendix F.2.5.2)
- new placement / initializer ambiguity (Appendix F.2.5.5)
- sizeof ambiguity (Appendix F.2.5.6)
- typeid ambiguity (Appendix F.2.5.7)
- type / value template arguments (Appendix F.2.5.8)

The preliminary scan of the body of a meta-function marked the position of the start of any tree-literal and attempted to parse a formal parameter. If a formal parameter was referenced, its value was used. Otherwise the parser back-tracked and copied the tree-literal unchanged into the meta-function body.

The multi-pass implementation did not parse statements within function bodies further than the replacement of $-expressions. It therefore incurred very few of the severe ambiguities that arise when expressions and declarations coexist.

5.10 Code Structure

The file input processing acquires tokens from source files and performs ANSI C preprocessing. The implementation closely follows the translation phases described in §2.1. It is shown in pictorial form in Figure 5.4.

Phase 1 (character mapping, trigraph replacement and universal character replacement) and phase 2 (backslash line continuation) are performed by nested subroutines in the input routine to the lexer.

Phase 2 operates in a demand-driven fashion, returning one line at a time to the lexer.

The lexer, automatically generated by flex++, performs phase 3. The lexer is very simple (150 lines of rules), just identifying each of the different lexemes, and creating an appropriate derived instance of the polymorphic FogToken.
Construction costs are reduced by making substantial use of the Flyweight pattern [Gamma95], with all tokens stored in a tokenization symbol table which is initially populated by special objects for all the reserved words and punctuation sequences. Whenever a text string is tokenized, the appropriate Flyweight is chosen, so that a string such as `and_eq` is encoded as an object that retains its spelling for the purposes of stringizing, but which otherwise behaves the same as the object created for the text string `&=`. Reserved words such as `bool` or `else` are similarly associated with objects that have relevant behaviour.

Minor complexities are resolved by using four alternate lexer states to handle

- normal tokenization
- waiting for `*/` in a multi-line C-style comment
- recognising preprocessor keywords between the `#` and subsequent directive
- non-standard tokenization after a `#include`.

Phase 3 operates in a demand driven fashion, returning one lexeme at a time. Each lexeme has a dual identity, as a preprocessor token and as a token.

The phase 4 preprocessing directives and macro substitution are not performed by the lexer. Phase 4 processing is performed by two further layers of processing, each operating in a demand-driven fashion to return one lexeme at a time to its caller. The inner layer performs macro substitution. The outer layer detects and implements preprocessor directives. Macro substitution is performed by three very simple parser grammars that

- locate any macro arguments following a function-like macro
- substitute any identifiers in the macro replacement
- retokenize around `##` and stringize `#` identifier

Substitution (scan 1) and replacement (scan 2) of identifiers may involve a recursive substitution, but not where scan 1 detects an identifier matching the current macro (§16.3.4-2). The entire replacement for the detected macro and
arguments is determined as soon as the macro is detected, with the resultant
token sequence cached and then emptied by successive demands for more
lexemes.

Preprocessor # directive lines are parsed using a further grammar, with identifier
substitution bypassed except where permitted by the syntax. Conditional
directives maintain a state stack, the top of which determines whether lexemes
from non-preprocessor lines should be discarded. #define and #undef maintain
the dictionary of macro definitions referenced by the inner preprocessor layer.
#include pushes a new source context on to the lexer.

The lexemes returned from the outer preprocessor layer comprise the token
stream after all ANSI C preprocessing has been completed, and before any FOG
extensions have been realised. The token stream includes whitespace and new
lines. To improve error diagnostics, each token is returned with its source file,
source line and source column context.

There is no phase 5 in FOG, the execution and source character sets of a source
to source translator are necessarily the same. There is however a distinct internal
character set in which escape sequences, universal characters and other
characters are uniformly represented by binary values within characters and
strings. The uniform internal representation forms the basis for concatenation,
and is created (lazily) when the string or character lexeme is created during phase
3. Each number, string or character has three spellings: the original ASCII source
spelling to be used by preprocessor stringizing, the uniform internal binary
representation, and a regenerated ASCII spelling for output purposes. The output
may need to differ from the input to preserve the meaning of concatenated escape
sequences (§2.13.4-3):

"\xA" "B" concatenates to "\012B" not "\xAB"

The concatenation phase 6 and the analysis phase 7 are described in
Section 3.1.1.

Early FOG implementations performed $-substitution, Cpp substitution and
#directive resolution within the lexer, permitted $-substitution within characters
and strings and identified reserved words explicitly. The implementation was
large, complicated, vulnerable to obscure programs and lacked clear semantics.
The current implementation is simple, with a clear responsibility for each
processing layer. The leakage between layers is very constrained:

- macro definitions are accessible via the std::get_cpp() meta-function
- the using form of include currently shares the same low-level
  implementation mechanism as #include
- the source file identification describing the name and origin of each source
  file contains a mixture of pre-processor and non-preprocessor attributes

The very mundane organisation of analysis phases demonstrates how the
superset approach successfully avoids the phase dependencies prevalent in other
C++ compilers.

5.11 Grammar Metrics

The complexity of the FOG grammar and the success of the multi-pass and
superset approaches can be judged by comparing the parser generator statistics
for three alternative C++ grammars. Table 5.6 provides a summary of the statistics
variously extracted from the source grammar, from the generated output or from the report file of the modified version of bison++ used to build FOG.

<table>
<thead>
<tr>
<th>Unresolved Shift-Reduce conflicts</th>
<th>Roskind</th>
<th>gcc</th>
<th>CPPP</th>
<th>C++ superset</th>
<th>C++ multi-pass</th>
<th>FOG multi-pass</th>
<th>FOG superset</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>704</td>
<td>410</td>
<td>23</td>
<td>14</td>
<td>16(+1)</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>%prec usage</td>
<td>0</td>
<td>109</td>
<td>22</td>
<td>8</td>
<td>6</td>
<td>6(+1)</td>
<td>15</td>
</tr>
<tr>
<td>Tokens with precedence</td>
<td>0</td>
<td>61</td>
<td>40</td>
<td>14</td>
<td>4</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>Rules</td>
<td>664</td>
<td>779</td>
<td>431</td>
<td>561</td>
<td>661</td>
<td>1456</td>
<td>958</td>
</tr>
<tr>
<td>States</td>
<td>1257</td>
<td>1399</td>
<td>702</td>
<td>897</td>
<td>1119</td>
<td>2217</td>
<td>1585</td>
</tr>
<tr>
<td>Non-ASCII terminals</td>
<td>77</td>
<td>82</td>
<td>104</td>
<td>95</td>
<td>82</td>
<td>241</td>
<td>110</td>
</tr>
<tr>
<td>YYLAST</td>
<td>13954</td>
<td>9534</td>
<td>3724</td>
<td>5060</td>
<td>6501</td>
<td>30119</td>
<td>14545</td>
</tr>
</tbody>
</table>

Table 5.6 Grammar Statistics

Imperfections in the grammar result in conflicts, which may be left unresolved, in which case they are tabulated in the top two lines. Alternatively %prec may be exploited to resolve them, in which case the conflicts appear on the third line. Whether conflicts should be resolved is something of a matter of taste: any unresolved conflicts give cause to concern that grammar problems are present, and that the default resolution is erroneous, but equally a large number of resolved conflicts may hide some erroneous resolutions.

The number of %prec's used to resolve conflicts are shown on the fourth line. %prec's operate by resolving the competition between a shift and a reduction. A large number %prec's is bad since a %prec introduced to solve one problem may accidentally resolve another wrongly.

%prec's operate by establishing a precedence between rules and input tokens. The number of input tokens that can be compared is shown on the fifth line. Again large numbers are bad because there is greater opportunity for accidental wrong resolution of problems.

The remaining lines are explained in subsequent commentary.

The figures should be treated with some caution, because the goals and completeness of each grammar are not comparable. (A smaller number is better for all entries.)

The Roskind grammar is accurate, but lacks templates and other modern facilities.

The gcc grammar is an almost complete C++ grammar, and contains a few extensions. The grammar code also contains error recovery and performance optimisations.

The CPPP grammar is preliminary, and lacks some modern facilities.

The multi-pass and superset grammars aim for syntactic consistency, whereas the other grammars aim for semantic accuracy.

The C++ superset grammar (Appendix B) is a complete working C++ grammar with error recovery. The grammar unifies the declaration and expression syntax to resolve ambiguities.

The FOG multi-pass grammar is that of the earlier FOG implementation. It is fairly complete, adds syntax for meta-level processing, has some error recovery, but does not provide parsing within function bodies.
The C++ multi-pass grammar is a version of the FOG multi-pass grammar trimmed to handle just C++ but adapted to parse all of C++. The grammar contains some error recovery, and a structure that retains some unnecessary generality from the full grammar. The grammar has never been used. The results serve to compare the parsing approach adopted in FOG with other C++ approaches, and to determine the extra complexity introduced by FOG extensions.

The FOG superset grammar (Appendix C) is the working FOG grammar, using unified parsing and incorporating error recovery.

Conflicts

Each conflict provides an opportunity for a programming error to yield an incorrect grammar implementation. The low number of conflicts in the Roskind and even smaller number in the new grammars ensure that the residual conflicts can be analysed by hand.

Roskind makes a point of avoiding %prec, relying on line ordering in the grammar instead. This is an understandable policy with yacc, since yacc does not report conflicts resolved by using %prec. However, the use of %prec in FOG, and validation by bison, allows lines to be reordered and explicitly identifies each conflict where that conflict is intended to be resolved. The extensive use of precedence by gcc and CPPP and the very large number of tokens which can participate in precedence resolution offer ample scope for a conflict to be silently and erroneously resolved.

The FOG grammars use only two active precedence levels, reducing hazards of fortuitous resolution between levels. 16 tokens share the same level, and so only one token has a distinct precedence level. Two bordering precedence levels have the dummy tokens REDUCE_HERE and SHIFT_THERE so that precedence rules are coded as in Appendix F.2.3.1.

The superset and multi-pass approaches clearly have fewer conflicts. The multi-pass approach has fewest, since grammar conflicts are resolved in independent passes. The (+1) on the FOG grammar accounts for the dangling else resolved by the token acquisition routine. Substantially larger numbers should be added to gcc and CPPP metrics to account for their lookahead code.

Size

The number of rules in a grammar corresponds to the line count in a more conventional program, and so measures the programming effort. The number of states in the generated state machine is a measure of the complexity of the problem that the program solves. YYLAST is the dimension of the two largest tables generated for the table-driven parser. YYLAST therefore provides a good indication of the executable size after all compactions have been performed. The size of the FOG multi-pass grammar is close to the 32767 implementation limit for the standard bison parser. During development FOG exceeded that limit, which is why a modified version of bison++ was used to support 99999 table entries.

The number of rules and states in the Roskind and gcc grammars are surprisingly similar, more so after allowing for the greater completeness and error recovery in the gcc grammar. The CPPP grammar is noticeably smaller, which might be an indication of a better grammar, or confirmation of its inaccurate status.

The C++ superset grammar is noticeably smaller, the benefit of merging declarator rules into expressions rather than flattening out to try to create more lookahead.

The trimmed C++ multi-pass grammar is similar in size to the gcc and Roskind grammars. Each of these grammars is over-size through the use of grammar flattening to extract or resolve extra semantic context at the syntactic level.
The full multi-pass FOG grammar is at least twice as large, the result of some extra grammar for the meta-level syntax and the extra rules to support syntax-driven interaction with meta-functions and meta-variables. The extra rules are mostly simple. It would be wrong to infer that FOG doubles the grammar complexity.

A fairer comparison of the parsing cost of FOG can be drawn from the more effective superset grammar which is rather less than twice the complexity of its C++ counterpart and pretty similar to the gcc and Roskind grammars.

Parsing Assistance

The number of non-ASCII terminals used by the grammar provides a little insight into the assistance provided to the parser. Some non-ASCII terminals are unavoidable to represent the 24 special character sequences (+=), 63 primary reserved words (class), 11 alternate reserved words (xor_eq) and 4 parametric tokens (identifier). The exact number of such terminals depends upon whether alternate reserved words are resolved by the lexer, and whether groups of similar reserved words such as private, protected and public are replaced by a parametric token such as access-specifier. A practical implementation uses more than the minimum number of terminals in order to communicate extra disambiguation information to the parser.

The Roskind figure does not account for some more recent reserved words and has one extra disambiguating token.

The gcc figure is more representative, adding six disambiguating tokens.

The larger value for CPPP is the result of the extra information provided by its lookahead parser.

The values for the superset C++ grammar correspond to exactly one token per reserved word, punctuation sequence, and 5 parametric tokens.

The much larger figure for the FOG multi-pass grammar is mostly caused by the need for two tokens per meta-type, one to prime the parser to perform the correct syntax-driven parse, and one to identify the meta-type of a meta-variable instantiation.

The additional tokens for the FOG superset grammar are required for the non-reserved words.

Performance

The superset approach incurs minimal costs from back-tracking, since a back-track occurs only for very unusual code and the very occasional template retry. The parser is probably slightly slower because more reductions are performed for the many levels of expression precedence than for the normal nesting of declarators. This cost could be alleviated by flattening the expression syntax and using precedence as in the CPPP grammar. The extra semantic disambiguation will incur a small cost, but since it occurs within the disciplined context of a tree rather than the less appropriate parser stack, there is probably a small net saving.

The need for semantic correction to the AST is rare and so should incur very little cost.

The overall relative speed cost of the superset approach may be slightly adverse. Additional semantic analysis work is introduced by the need to correct the syntax analysis, and some time is wasted on pursuing inconsistent candidate template/arithmetic syntaxes. It is difficult to assess to what extent these moderate costs are offset by the significant simplifications of each of the lexical, syntactic and semantic analyses. However this loss is well justified given the

4. Estimated at 0.01% in Section 5.8.
elimination of intra-statement coupling between syntactic and semantic analysis stages and the possibility of demonstrating that the grammar implementation is accurate.

The parsing tables for the superset grammar approach are clearly smaller. The AST provides an easier environment in which to resolve ambiguities. It is therefore likely that deferring ambiguity resolution to the semantic level may contribute a further small code saving.

5.12 Summary

We have shown how the need for type information to parse C++ can be eliminated by identifying a larger superset language of which C++ is a part.
We have introduced an extended form of regular expressions that enables ambiguities in the C++ grammar to be analysed.
We have described a potentially novel approach to back-tracking in yacc that enables C++ to be parsed without template information.
We have therefore shown how a C++ parser can be constructed without leakage between its lexical, syntactic and semantic stages leading to considerable simplifications in each stage.
A minor disadvantage arises from the need to correct a syntax analysis error during semantic analysis. This is estimated to occur for approximately 0.01% of statements.
6 Files

This chapter is concerned with the code generation stages of the FOG to C++ translator; that is the preparation of the AST for output and then emission of the AST as C++ code.

We first describe some practical problems that arise from the use of a standalone meta-compiler and introduce the concept of utility level to coordinate multiple sessions and multiple generated files.

A default mapping of declarations to output files is described. The straightforward syntax extensions to support user control of placement and dependencies is relegated to Appendix F.4.

Then we describe the declaration dependency analyses necessary to establish a legal ordering and the subsequent emission of output files using that ordering.

Finally we consider the need for integrity between meta-compilation sessions and suggest how this may be achieved after further research.

6.1 Practical problems

The compilation model for FOG in Figure 1.1 showed how any number of conventional C++ interface and implementation files are generated by one or more meta-compilations.

Three practical problems arise from the use of a meta-compiler as an independent translator prior to a conventional compiler, rather than an integrated part of that compiler.

The new problems concern

- naming and location of output files
- mapping of declarations to output files
- unique and consistent generation of each output file

The problems are resolved by providing a default naming and placement policy, which may be at least partially suitable for many applications.

Syntax extensions are necessary to support full control of output files for more interesting applications. These extensions are an unfortunate but unavoidable corollary of the need to generate multiple outputs.

Consistency between sessions is facilitated by classification of source declarations as

- unique to a meta-compilation session
- shared contributions to multiple sessions
- shared immutable reference declarations

6.2 File disposition

FOG translates extended C++ to plain C++, which must comply with the C++ standard. In particular the resulting C++ declarations must satisfy the One Definition Rule (§3.2); each declared entity must be defined just once. FOG must therefore partition declarations between interface and implementation files.

A conventional C or C++ compilation processes many input files and generates a single output file. The input files and output file are readily specified on the compilation command line in conjunction with additional information to identify search paths for include files.

This policy does not extend directly to a meta-compiler that may generate many output files, whose existence may be unknown to the author of the command line.
The default behaviour of FOG is to emit an implementation and an interface file for each non-nested class and each namespace encountered in its source files. Template specializations are emitted with the primary template. Template instantiations may be placed using file-spaces (Appendix F.4.4). Command line options allow specification of:

- default paths for output directories
- output file prefixes (such as sys/)
- output file suffixes (such as .hxx)
- a file name for the global namespace

Files are named from their constituent class or namespace, with interface files acquiring a .hxx suffix, and implementation files a .cxx suffix. Template files are given distinct .H and .C suffixes. The following code is therefore partitioned according to the comments:

```cpp
class MyClass
{
    int i;
    !inline void f() { /*...*/ } // MyClass.hxx
    class Nested();
}

template <class T>
class MyTemplate
{
    static T t;
}

template <>
static int MyTemplate<int>::t = 0; // MyTemplate.c (and H)
```

Include file guards are incorporated into all generated files. The guard name is determined by converting all alphanumeric characters of the file name to upper case and non-alphanumerics to underscore. The file MyClass.hxx is:

```cpp
#include <string>

class MyClass // MyClass.hxx
{
    private:
    !line 3 "Thesis_6_2.fog"
        int i;
    
    private:
    !line 4
        void f();
    
    private:
```

```cpp
#include <string>

class MyClass // MyClass.hxx
{
    private:
    !line 3 "Thesis_6_2.fog"
        int i;
    
    private:
    !line 4
        void f();
    
    private:
```

```cpp
#include <string>

class MyClass // MyClass.hxx
{
    private:
    !line 3 "Thesis_6_2.fog"
        int i;
    
    private:
    !line 4
        void f();
    
    private:
```

```cpp
#include <string>

class MyClass // MyClass.hxx
{
    private:
    !line 3 "Thesis_6_2.fog"
        int i;
    
    private:
    !line 4
        void f();
    
    private:
```

```cpp
#include <string>

class MyClass // MyClass.hxx
{
    private:
    !line 3 "Thesis_6_2.fog"
        int i;
    
    private:
    !line 4
        void f();
    
    private:
```

```cpp
#include <string>

class MyClass // MyClass.hxx
{
    private:
    !line 3 "Thesis_6_2.fog"
        int i;
    
    private:
    !line 4
        void f();
    
    private:
```
class Nested
{
};
#endif

The file naming and disposition policy outlined above provides a convenient and often adequate default. Programmers may use environment variables, command line tokens or extended declaration syntax to choose their own modularization where appropriate.

6.3 Utility

It is rarely appropriate to emit interfaces and implementations for every class, since this generates new interfaces for standard classes such as iostream. Although it could be convenient to generate a new interface with an extra virtual function, this requires availability of the source and an ability to recompile all code that makes use of iostream.

In practice, certain external classes must be immutable. In order to support multiple meta-compilations, it is also desirable to group application classes into sub-systems that are mutually immutable.

When FOG is used to support an Aspect Oriented Programming style, weaving the contributions from a number of algorithm-centric source modules into the data-centric perspective of C++ classes, a single source file may contribute to many class files, and a class file may have contributions from many source files. This many-to-many transformation must be tamed if a number of partial rather than one massive meta-compilation session is to be supported.

These problems are resolved by extrapolating from the concept of a class utility [Booch94], where a class utility is free-standing code independent of the current application. A utility level metric determines the extent of each contribution to the generated code.

FOG associates a utility level with each source file and source declaration. From these FOG determines a utility level for each output file and output declaration. On output, the utility level is used to suppress output for declarations independent of the current meta-compilation, and to diagnose inconsistent packaging of declarations.

Utility levels

The utility levels in increasing order of stringency are:

pool

The pooled utility level supports source files contributing to more than one meta-compilation session. The source file contains a pool of declarations, some of which may be output in one session, others in another.

Files and declarations with pool utility are not emitted, however once composed with other declarations or into scopes with emit utility, the declarations contribute to emitted files.

In a typical usage, a pool utility file defines some virtual functions for classes A, B and C. These virtual functions support some algorithm-centric programming concern that cuts across the conventional data-centric organisation of the classes. When the conventional class file (with emit utility) is meta-compiled in conjunction with the pool file, the pool declarations for that class are promoted to emit and so complete interface and implementation files for the class are
Files

generated. Declarations for other classes remain as pool declarations and do not provoke partial or conflicting emission.

emit

This is the default utility level. It imposes no constraint on the composition or emission of declarations. All declarations should be emitted to some output file.

utility

This is the most stringent utility level. Declarations from a utility file are not to be changed or overwritten. This utility level is the same as frozen, differing only in that the name indicates that the file was included via using/utility rather than #include.

Output files with a utility or frozen utility level are not emitted and any contained declarations of emit level are diagnosed.

frozen

The frozen utility level is applied automatically whenever a file is read as a result of a #include directive. Declarations from such a file can only be used, they cannot be changed, and of course the file cannot be overwritten. This provides the requisite compatibility to use C++ libraries.

Semantics

The more stringent of the prevailing utility level and the enclosing scope utility level is applied to each potential declaration. Composition of potential declarations to produce the final actual declaration again selects the most stringent utility level.

No utility level is maintained for forward declarations, and the utility level is not inherited.

As each source file is included, any specified non-emit utility becomes the prevailing utility to be applied to all declarations in that source file. The previous utility level is restored on completion of the file. The source file utility is specified by a switch as part of the replacement syntax for #include.

A more stringent utility level may be explicitly specified for an individual declaration. And when that declaration is a scope, the utility level applies recursively to all declarations in that scope.

A more stringent utility level may also be specified for a potential output file to ensure that its usage is restricted to references from generated #include directives. Emission of declarations to that output file is therefore inhibited.

Compilation

The utility of each declaration is recursively propagated to enclosing scopes and containing files, so that any scope or file has the most stringent utility of any declaration in that scope or file. (No propagation occurs for namespaces). The results of this propagation are checked in a verification pass over the AST. Any declarations with a less stringent utility than their enclosing scope are diagnosed as illegal attempts to compose additional functionality where a change is not permitted. Similarly any declarations with a less stringent utility than their containing file are diagnosed as attempts to modify immutable files.

6.4 Dependency Analysis

FOG declarations permit considerable freedom in their ordering and placement. C++ declarations are much more stringent. FOG must therefore find an
appropriate form in which to emit the declarations to avoid C++ compilation errors. This requires establishing a legal order for declarations, a partitioning of those declarations into files and incorporation of appropriate forward references and \#include directives. While observing the constraints of a legal order, FOG groups similar declarations to improve readability, and normalises the output using an alphabetic ordering to maximise the likelihood of an unchanged regeneration.

The legal order is established by a number of passes over the AST. These first establish file names and then build graphs to define the ordering constraints.

An implementation and interface file is established for each declaration using an explicit specification or an algorithmic default. The default is determined by the enclosing scope, which at the top level associates distinct files with each class or namespace using the policy summarised in Table F.1.

Usage nodes are defined for each form of usage of each declaration, and a usage dependency graph is built between these usage nodes by traversing each declaration and identifying the declarations upon which it depends. This is most easily understood from an example.

```cpp
class A {}

class B {}

class F;

class C
{
    static A a;
    B b;
    inline F *f() { return 0; }
}

// class A : public C {};
```

The corresponding usage dependency graph is shown in Figure 6.1, in which solid arcs denote a directed dependency of the node at the start upon the node at the finish. Dashed arcs denote redundant dependencies. The dotted arc is an extra dependency for the commented line at the end of the example.

Six of the eight different forms of usage node are used in the diagram:

- \textit{Cname}, a forward referenced name of \textit{C} such as \texttt{class C};
- \textit{Chead}, the start of the interface for \textit{C}, such as \texttt{class C {}};
- \textit{Ctail}, the end of the interface for \textit{C}, such as \texttt{}};
- \textit{Cinline}, the start of the inline implementation of \textit{C}: just a place-holder.
- \textit{C::fint}, the interface of \textit{C::f}, such as \texttt{F *f();}
- \textit{C::fimp}, the implementation of \textit{C::f}, such as \texttt{F *C::f() ( return 0; )}

A further two forms are required to represent:

- friend relations
- explicit file dependencies

The interfaces of class members are bounded by the head and tail of their class. Although not shown, a nested class fits in naturally with its head and tail bounding its contents. The original ARM-style nested classes may be enforced by declaring the nested name and the nested head as dependent on its enclosing head, and correspondingly that the enclosing tail depends on the nested name and nested tail. Alternatively, when the \texttt{--unnest} command line option is used, the ISO C++ option of nested class definition appearing independently is enforced. The nested name is then defined as dependent on its enclosing head, and correspondingly the enclosing tail as dependent on just the nested name. In addition the nested head is then dependent on the enclosing tail.
The external dependencies of the class member interfaces are promoted to the class, since extra-class declarations may not occur intra-class. Thus the dependency of `C::f_int` upon `F_name` shown dashed is promoted to a dependency from `C_head`.

The differing dependency patterns of `C::a` and `C::b` upon `A` and `B` reflect the distinct dependency of static and non-static member variables; a static member is not part of the class and so its type may be incomplete (§9.4.2), whereas a non-static member is. As a result, removing the comment from the final line in the example adds the extra dependency shown dotted, and does not form a loop, although a casual examination of the code might indicate that it does.

The usage dependency graph describes the entire sub-program visible to the meta-compilation session. This is generally smaller than the entire application, but much larger than the distinct subset of declarations used by each output file.

In the example, declarations of `F` remain unknown. When `C.hxx` is emitted, it must `#include B.hxx`, but need only forward reference `A` and `F`. If `A.hxx` is included, subsequent compilations of `C` will be burdened with unnecessary dependencies. If `F.hxx` is included, a failure may occur since the forward declaration of `F` is insufficient to establish `F.hxx` as the appropriate interface file name.

The declarations emitted in each file must be carefully organised, so that each emitted file is useable and doesn't refer to unnecessary files. Unnecessary declarations should be avoided. Necessary declarations must be referenced.

The usage graph contains all the necessary information. If appropriately organised, emission of any file just involves selecting all the corresponding usage
nodes for emission, and ensuring that all their ancestors are resolved by
#include directives or forward declarations. The passes over the AST and usage
graphs to achieve this appropriate organisation are now described.

6.4.1 Create Usages

The create usages pass traverses the complete AST to create the usage nodes
and dependency arcs.

Six different forms of usage were used on Figure 6.1, and show how even a simple
program generates a complicated graph. The complexities arise from two sources
- the context-sensitive form of each usage in C++ (§3.2)
- the need to observe modularity constraints in the generated files

The C++ complexity requires careful coding of the create_usages() virtual
function at the relevant AST nodes. Particular care and complexity arises for
templates for which instantiations delegate dependencies to the primary template,
but for which (partial) specializations compound dependencies. Namespaces,
friends, comma-separated declarations, typedefs, inlined functions, default
argument lists and anonymous unions provide their own minor complexities.

The modularity constraints arise because declarations cannot be freely ordered.
Forward referencing constraints must be observed. Class declarations must occur
together without any interleaving. Grouping of declarations in files must be
possible, although this may sometimes necessitate some declarations preceding,
and others following a #include. Files may therefore appear to overlap.

Most of these modularity constraints can be enforced by adding additional
dependencies to the usage graph. Thus in the earlier example, member
dependencies were promoted to the class head to ensure that they were satisfied
before class emission started, and so ensure that class emission could proceed
without interleaving. The remaining modularity constraints are observed, if
possible, by careful ordering of usages and files. It is possible for a programmer
to over-specify the placement of declarations and so define an insoluble problem.
In this case the problem is diagnosed and then approximated.

6.4.2 Usage ordering

Each node of the usage graph is assigned to one of the following worklists
- 25 lists of nodes with no remaining dependencies upon other nodes
- an input list of nodes with dependencies

25 lists are maintained so that each different style of declaration is in a separate
list. They are, in prioritised order:

"interface-finish",       // Tail/end of a scope
"name",
"friend",
"public typedef",
"protected typedef",
"private typedef",
"public enum",
"protected enum",
"private enum",
"public variable",
"protected variable",
"private variable",
"public function",
"protected function",
"private function",
"static public variable",
"static protected variable", 

29-June-2001
"static private variable",
"static public function",
"static protected function",
"static private function",
"interface-start",       // Head/start of a scope
"file",                  // #include
"inline",                // inline implementation
"implementation",       // not-inline implementation

Thus public/protected/private, static/non-static and function/variable/typedef
distinctions are each associated with different lists. The lists are prioritised and,
as a result, the ordering algorithm automatically groups similar declarations. This
improves readability and repeatability. Most of the priorities are chosen for
aesthetic properties of the output files. However, in order to avoid overlapped and
consequently interleaved emission of classes, the tail of a scope is given very high
priority and the head of a scope given very low priority. If the dependency graph
has been built correctly, this ensures that the class members are emitted and the
class brace closed before another class brace is opened.

The worklist ordering algorithm repeatedly chooses the highest priority non-empty
dependency-free list, ensures it is in alphabetical order and moves the first entry
to a further list, the ordered output list, in which maintains the output in a
dependency observing order. The dependencies from all nodes dependent upon
the moved node have now been observed. These are removed with the result that
some nodes may be moved from the final list to the dependency-free lists.

The algorithm has quadratic asymptotic complexity, but the behaviour is close to
linear in practice.

The algorithm stalls if there is a dependency loop. In this case the loop is printed
out in an error message and a dependency in the loop is arbitrarily discarded
before the algorithm resumes.

The asymptotic complexity under error conditions is quartic. This is highly
pathological requiring dependency loops between all nodes.

6.4.3 Usage closure

Once the legal order has been established, the transitive closure of the
dependencies of each usage node is determined by propagating the set of
ancestors to each child. Each usage node now has a set of precursors:
declarations that must be visible before the usage node is emitted. This trades off
the one-off execution time of this propagation and the memory costs for each set
against the cost of repeated traversals of ancestors in later algorithms.

6.4.4 File usage ordering

A similar worklist algorithm is used to establish the ordering of potential #include
directives, after projecting the usage node dependency on to the much smaller
number of file nodes to create a file node dependency graph. Only one list of
dependency-free nodes is used.

6.4.5 File pre-ordering

In order to emit dependencies efficiently, it is necessary to know what usage
nodes are visible after a file has been included. This information is gathered in two
passes.

The first pass is a simple initialisation of all files to identify the usages visible
within the file but excluding those visible from nested includes.
6.4.6 File post-ordering

The second pass is performed using a traversal of all files ordered according to their earliest last (most dependent) usage node. This order guarantees that nested include files are processed before less nested includes. In this pass all usage nodes visible in included files are added to those visible in the including files.

6.4.7 Emission

The final emission pass creates the output file from the ordered usage nodes for that file. Code generation is relatively straightforward since each usage node closely corresponds to a partial declaration. Linear traversal of the usage nodes generates the required output.

In order to ensure compact and correct resolution of references, the set of currently visible declarations is maintained during emission. As each usage node is considered for output, its necessary precursors are checked. If any of these is not visible, appropriate forward references or #include directives are emitted, with more highly dependent include files favoured to maximise the rate at which necessary declarations become visible. Eventually the usage node can be emitted, but only after the appropriate scope has been established. Namespace scopes are established lazily to avoid repetition of the same scoping construct, since eager establishment of scope would fail to suppress the commented lines in:

```cpp
extern "C"
{
    declaration-1;
    //
    // extern "C"
    // {
    declaration-2;
}
```

Generation of C++ text from the AST nodes inserts appropriate braces or parentheses to respect the exposure/encapsulation of lists and to use comma or whitespace separation between elements. This ensures that the results of meta-programming are correctly formatted in accordance with their usage context. The meta-programmer does not need to worry about the punctuation, since there is no punctuation in the AST, merely metaobjects with appropriate meta-types.

The potential output file is generated in memory so that it can be compared with any pre-existing file. An unchanged file is not overwitten to avoid corrupting its modification time.

6.5 Target File Generation Policies

6.5.1 Global namespace

Class and namespace declarations are normally written to files based upon the declaration name. Unfortunately, the global namespace has no name and creation of name-less files such as .hxx and .cxx would cause confusion.

The global namespace contains many utility declarations from C libraries which do not need repetition in another file. The global namespace contains few declarations from application code, since C++ encourages the use of encapsulation to avoid polluting the global name-space.

The default behaviour of FOG is therefore to ignore global namespace declarations. A command line option permits a name to be specified for the global name-space. If this option is used, the global namespace behaves (for output file naming purposes) as if it were a namespace with the specified name.
6.5.2 Friend functions

A significant exception to the above discussion occurs for friend functions. Such functions although logically part of some class are often given namespace scope to avoid biasing overload resolution to the left argument

\[
\text{friend MyClass operator\+(const MyClass&, const MyClass&);} \\
\text{or to use some other class as the first argument}
\]

\[
\text{friend ostream& operator<<(ostream&, const MyClass&);} \\
\]

Since these are (global) namespace scoped, the implementation should be emitted to the namespace implementation, which is unlikely to be what is required. The implementation of a namespace-scoped function is therefore placed with the implementation of the class that declares it a friend, provided there is exactly one such class.

6.5.3 Source File Protection

The output from FOG comprises C++ source files that may not differ significantly in name or path from files input to FOG. There is ample scope for accidental overwriting of input files by output files, either through ill-considered user commands, or through coding errors in FOG.

FOG protects against such errors by starting every generated file with the character sequence \/*$OGS! and only allowing overwrites of files with this signature. If the user really wants to give FOG complete freedom to overwrite anything, then the -f command line option must be used.

6.5.4 Suppressed Non-changes

FOG may generate many output files from a single invocation, normalising each file to minimise changes. It may often be the case that some generated files are unchanged and so subsequent dependent recompilations are unnecessary. FOG therefore compares each potential output file with any pre-existing file, and if the potential output file differs only in whitespace (or comments), the existing file is retained, avoiding any change of creation date. A make script can then skip unnecessary compilations.

The operation of this algorithm may be observed using command line options: -nc to notify file changes and -ne to notify preservation of existing file. The policy may be overridden by using -f to force all files to be created.

6.5.5 Net dependencies

In order for a makefile to determine whether FOG needs to be executed, the makefile needs to know the dependencies of output files and input files. The -o command line option may be used to emit a make include file that defines the dependencies as the dependants of that include file. All other generated files can then be reliably created by side effects.

6.5.6 Pretty Printing

The files generated by FOG are readable, unlike files generated by code generators such as cfront. The generated text differs little from what could have been typed manually.

In the multi-pass implementation, function bodies were emitted directly from the almost unparsed phase 6 token stream. This token stream retained an indication of the original source whitespace, and so the generated output could reproduce much of the user layout. In order to achieve consistent indentation, source text indentation was normalised to the first non-whitespace character in a function-
body, and then denormalised after pretty-printing had determined the position of the first character.

Source text normalisation required tab characters to be converted to spaces. The default FOG behaviour assumed the traditional 8 column tab spacing. The increasingly common configuration of 4 column tabs could be specified by adding `-t4` to the command line.

The superset implementation parses function bodies into syntax trees and so loses original whitespace. The entire output is therefore pretty printed without reference to the input.

### 6.5.7 #line

Translators to C or C++ may incorporate `#line` directives in their generated files, so that compiler diagnostics and debugger single stepping refer to the original source context, rather than the intermediate generated files. This is generally very helpful to the programmer, but when the translation process is unreliable or when obscure problems arise, access to the hidden intermediate is more useful.

The default behaviour of FOG is to incorporate `#line` directives in its output. The `#line` directives may be replaced by comments by using the `-comment_line_numbers` command line option, or suppressed altogether by `-no_line_numbers`.

Use of `#line` directives has the unfortunate consequence that almost any change to a source file causes regeneration of most output files, if only to update line number information.

Total suppression of line numbers produces the most readable intermediate, but gives no clue as to the origin of each source code segment.

Preservation of the line numbers as comments avoids the regeneration problem, while retaining traceability. Operation at this level may often be appropriate, since single stepping a misbehaving function composed from many contributions is probably best performed when the function body appears in its composed form. Strange compilation errors are also more easily diagnosed when the full context is visible. However, the line numbers are in comments and so the intermediate files are not regenerated to update them, and the commented line numbers may therefore prove to be seriously adrift.

Ideally a debugger would support more than one line number domain. In practice users may have to choose which domain best suits the prevailing needs.

### 6.5.8 Integrity

There is a danger that the composite declarations in different meta-compilation sessions may be incompatible. This danger may be compounded by command lines requesting more than one meta-compilation session to generate the same output files, thereby causing subsequent compilations to be inconsistent. If the user is fortunate, the inconsistencies will lead to helpful compilation errors. More likely there will be confusing linker diagnostics or worse still, run-time failures. The latter problems are violations of the C++ One Definition Rule (§3.2) which the standard allows an implementation to leave undetected. It is a user programming error. In C++, use of shared include files makes this form of mistake relatively rare, although over-enthusiastic incremental compilers can easily trigger such errors.

The increased level of abstraction and greater opportunity for configuration problems in FOG justifies extra effort to assist the programmer. This is an area for future research. Two possible solutions are outlined:
Checksum

The integrity problem may be solved by incorporation of a cyclic redundancy check (CRC) hash code as part of the name of a static member variable in every FOG generated class. The member variable is initialised by a meaningless calculation that causes the linker to reference the equivalent names in all dependent classes. Thus if class A uses classes B and C, the consistency checking code for class A could be:

```cpp
class A
{
public:
    static char _c_r_c_A43F1507; // A43F1507 is CRC of rest of class
};
```

```cpp
char A::_c_r_c_A43F1507 = char(&B::_c_r_c_5670BD33 + &C::_c_r_c_EE8241C5);
```

The definition originates from the defining session. The references are independently calculated by referencing sessions. If the two sessions are using inconsistent definitions, the problem will eventually show up as a requirement for the linker to resolve two different symbols for the one class. This will fail and prevent run-time anomalies, subject to the CRC algorithm producing highly uncorrelated hash codes.

When class A uses class B is a matter for further research and careful definition. A safe definition could follow the normal C++ usage definition (§3.2-4). This is unnecessarily strong, since meta-compilation does not establish class layout, and so the usage of types during meta-compilation is generally by name rather than by value.

A simpler but not quite safe definition defines the usage as all classes generated by the same meta-compilation session. This is unsafe because derivation rules for meta-programs may have used declarations from base classes.

It would seem that each meta-compilation session should keep track dynamically of each type used in more than name, and that the references in the linker expression should involve all such types.

This approach incurs a one byte per class penalty, which could be eliminated if it were possible to access the linker symbol calculations directly. A zero cost implementation could use the generated symbol name as the name of the virtual function table.

Database

The problem of detecting conflicting or redundant generation of an output file by more than one meta-compilation session is less easy to detect early. The checksum approach will detect a conflict late in the build process, and is unlikely to produce a helpful diagnostic. A rebuild using -nc should not result in notification of any file creations, so this at least provides a way to investigate the conflicts.

Direct diagnosis of such conflicts, or any diagnosis of redundant but compatible generation from multiple meta-compilations requires a database file that maps output files to meta-compilation sessions. Duplicate mappings are errors.

The coordinating database could easily be specified as a command line parameter, and new or repeated meta-compilations would update the database. However meta-compilation sessions do not have a reliable identity, so it is not possible to detect whether a slightly changed meta-compilation command line represents a new or replacement session. It is certainly not possible to detect obsolete sessions. This problem is most easily resolved by manual or automatic deletion of the database whenever make scripts are changed.
This problem and solution is closely related to the template instantiation database necessary to satisfy the C++ requirement for instantiation of precisely the used functionality. The template database is necessary for language compliance and an aid to compiler efficiency. A similar or combined FOG database is just a diagnostic aid.

6.6 Summary

We have introduced the concept of utility level to assist in placing and diagnosing misplacement of declarations across multiple files.

We have shown how a valid C++ output file ordering can be established for the FOG declaration AST by building usage dependency graphs first at the declaration level and then at the file level.

Finally we have described some minor details concerning file emission and identified the need for further work to establish reliable diagnosis of inconsistent meta-compilation sessions.
This chapter provides example uses of FOG, starting with very simple idioms and progressing to more serious applications.

7.1 Idioms

An idiom is a simple coding construct that arguably could be part of the C++ language.

Although some of the simple idioms are little more than one-liners, they should not be dismissed as trivial programs. They demonstrate how real programming problems can have very simple solutions in FOG. The examples support direct expression and realisation of programming intent and consequently improve reliability and maintainability, trading off some declarative complexity against ease of instantiation.

7.1.1 InheritedTypedef

Using inheritance, a programmer may derive a specialisation from a base class. This may involve a refined implementation of a base class operation. Whereas the derived implementation is constrained by the inner invocation from the base class, in languages such as BETA [Madsen93], C++ imposes few limitations. The programmer is free to provide a replacement implementation, or to incorporate the derived functionality where appropriate. The derived functionality is incorporated by invoking the base-class method explicitly.

```cpp
void DerivedClass::method()
{
    // some code
    BaseClass::method();
    // more code
}
```

When DerivedClass is derived directly from BaseClass

```cpp
class DerivedClass : public BaseClass { /* ... */ };
```

the code fragment operates as might be expected.

However, if DerivedClass is indirectly derived

```cpp
class IntermediateClass : public BaseClass { /* ... */ };  
class DerivedClass : public IntermediateClass { /* ... */ };
```

it is unclear why IntermediateClass::method() was bypassed.

It may be that when the fragment was first written, DerivedClass was directly derived, and there was no IntermediateClass method to worry about. When subsequent maintenance or evolution introduced IntermediateClass and IntermediateClass::method(), a bug was also introduced.

If IntermediateClass::method() is accidentally bypassed, then the use of the BaseClass name may lead to incorrect behaviour.

If IntermediateClass::method() is deliberately bypassed, then the code is obscure, since in some respect DerivedClass is not a specialization of IntermediateClass, or IntermediateClass is not a specialization of BaseClass. Such obscurity probably hides a design flaw, and certainly merits a comment to explain why the IntermediateClass::method() has been bypassed.

Summarising: The use of explicit type names to refer to inherited members leads to fragile code.

Stroustrup on page 292 of The Design & Evolution of C++ [Stroustrup94] describes discussions about a potential language extension to resolve this
problem, and credits Michael Tiemann for providing the following very simple solution:

```cpp
class foreman : public employee {
    typedef employee inherited;
    // ...
    void print();
};

class manager : public foreman {
    typedef foreman inherited;
    // ...
    void print();
};

void manager::print()
{
    inherited::print();
    // ...
}
```

The base class is referred to as `inherited` throughout the class, limiting the knowledge of the inheritance hierarchy to lines //1a and //1b. The reader is assured that the intent is to invoke the base class functionality, and may then question any reference to class names other than `inherited`. The maintainer does not risk breaking code when the inheritance hierarchy is reviewed.

FOG offers three alternative solutions that avoid the redundancy of line //1b, and consequently avoid the risk of an inconsistency between line //1a and //1b.

### Super meta-variable

FOG provides a built-in meta-variable `Super` that refers to the first base class. There is therefore no need for a typedef. The built-in meta-variable can be used instead.

```cpp
void manager::print()
{
    $(Super)::print();
    // ...
}
```

This is not a very elegant solution. It requires a $-expression to appear in normal application code.

### Super typedef

The typedef can be reintroduced to hide the $-expression from application code.

```cpp
class manager : public foreman {
    typedef $(Super) inherited;
}
```

This clearly expresses the intent and is guaranteed to be locally correct. There is however a possibility that the typedef could be omitted for a derived class, and if the typedef is accessible no compilation error would arise.

### InheritedTypedef meta-function

The problem of an inaccurate definition in a derived class, and the inconvenience of providing the definition in every class can be resolved by a derivation rule, and since the final solution is of general utility it is presented as a meta-function.

```cpp
auto declaration InheritedTypedef()
{
    private typedef $(Super) inherited :
    { derived(!is_root()); };
}
```
Expressing the solution as a meta-function introduces three changes to the typedef declaration:

A private keyword is added to ensure that the typedef is not visible outside the class. Omission of the keyword would leave the declaration vulnerable to assuming a prevailing accessibility from the invocation context.

An object-statement-scope {...} follows the typedef so that additional declarations can qualify the typedef. The derived(!is_root()) derivation rule, causes the typedef declaration to be generated in all classes that derive from the class for which the meta-function is invoked. Thus when invoked as

```cpp
class employee
{
    $InheritedTypedef();
};
```

no typedef is generated at the root (employee) for which Super may be undefined, but typedefs are generated for all classes derived from employee.

The use of @Super rather than $Super defers resolution of the meta-expression until each actual declaration generated as a result of the derivation rule is installed in the derived class. @Super therefore resolves to the super-class of the actual derived class. In contrast, $Super would be resolved when the potential declaration is defined for the root class, and would therefore result in the typedef always resolving to the super-class of the root class.

**Benefit**

Use of the meta-function requires a single line of source code for each base class, and ensures consistency throughout an inheritance hierarchy.

The conventional implementation requires one line of source code per derived class in the inheritance hierarchy, and is vulnerable to typographic errors.

**Alternatives**

This problem cannot be solved reliably without access to the name of the base-class.

The Tiemann approach localises the redundant declarations to a single place, potentially adjacent to the necessary declaration.

Applications that already make extensive use of preprocessor macros to define class scaffolding can implement the typedef with no additional application code:

```cpp
<Interface-file>
class DerivedClass : public BaseClass
{
    SCAFFOLDING_INTERFACE(DerivedClass, BaseClass)
    // ...
};
<Implementation-file>
SCAFFOLDING_IMPLEMENTATION(DerivedClass)
```

The typedef can be incorporated in the SCAFFOLDING_INTERFACE preprocessor macro. The examples in Sections 7.3.3 and 7.5 show how other standard declarations that may form part of SCAFFOLDING_INTERFACE can also be resolved automatically, rendering both of the SCAFFOLDING_IMPLEMENTATION and SCAFFOLDING_INTERFACE macros obsolete.

**7.1.2 NoAssign, NoCopy**

C++ provides powerful facilities to support the definition of encapsulated data types. C++ also eases the definition of user defined types by providing default
implementation for constructors, destructors and assignment operators. For classes involving pointers or allocated resources, the default implementations are often inappropriate. Replacement implementations may need to be provided.

For some classes, there is no possible replacement. Consider a class that defines properties of colours, with a single unique object for each used colour combination. Creating a copy of such an object is meaningless because that would break the uniqueness property. Assigning to such an object is illegal since assignment would involve creation of a new combination, and so could not be represented by the old object.

For other classes, the replacement implementation might never be used and so the development cost and code size of the replacement cannot be justified.

These problems are solved in C++ by declaring the relevant methods to be private and not providing an implementation for them [Coplien92] p45.

class UniqueColor
{
private:
   UniqueColor(const UniqueColor&); // not implemented
   UniqueColor& operator=(const UniqueColor&); // not implemented
};

Any attempt to copy or assign to a UniqueColor object outside of the scope of UniqueColor encounters a compilation error through the private access restriction. Within the scope of UniqueColor, compilation succeeds, but a linker error results from the missing implementations.

This is a well-known idiom, but is obscure and so not always recognised by the novice programmer. An accidental implementation of the not implemented functions may break the informal coding convention.

Meta-functions may be provided in FOG to express the intent more clearly and to enforce the non-implementation constraint.

auto declaration NoAssign()
{
   private $Scope& operator=(const $Scope&)
      :{ export/noimplementation; }
};

auto declaration NoCopy()
{
   private $(Scope)(const $Scope&) :{ export/noimplementation; }
};

The meta-functions may be invoked as

class UniqueColor
{
   $NoCopy();
   $NoAssign();
};

clearly expressing the programming intent.

The implementation of each meta-function makes extensive use of $Scope to define declarations appropriate to the invoking scope. The :{ } declares an object-statements-clause, within which function annotations can occur. In this case the function scope contains the single declaration export/noimplementation, which ensures that FOG generates a compilation error if any attempt is made to compose an implementation.

Use of $Scope and &Super are interchangeable in this example, since there is no re-evaluation in derived contexts. The extra () on $(Scope) is necessary to avoid the interpretation $(Scope(const $Scope&)).
Alternatives

NOCOPY and NOASSIGN preprocessor macros could be defined, however the name of the surrounding scope would have to be passed as a parameter.

class UniqueColor
{
  NOCOPY(UniqueColor);
  NOASSIGN(UniqueColor);
};

Enforcement of no-implementation requires the additional language support provided by FOG.

7.1.3 Mutate

ANSI C introduced the const qualifier to types to define unchanging values. C++ extends const to apply to objects and consequently member functions. Use of const ensures that any attempt to change an object is detected at compile time.

However, it may be appropriate for the implementation of an object to perform lazy evaluation of some of its properties, caching the results to avoid a re-evaluation. For example, the conventional complex class has a Cartesian representation, and must therefore calculate a polar representation, each time the polar representation is requested. A more sophisticated complex number class could cache the polar representation lazily, so that no calculation cost was incurred when the polar representation was unused, and avoid additional calculation cost for uses after the first. From an external perspective, the complex number object is unchanged by the use of a polar representation and so the usage method should be const. Internally the cached context changes and so the object is not const.

This distinction is referred to as physically-const and logically-const on p26 of [Coplien95b]. On p76, [Meyers92] uses the term conceptual constness.

Implementation of logically-const code requires that the const qualifier to be cast away.

```c
((ComplexNumber *)this)->_polar_value = ...
```

Usage of casts in application code is considered poor style. They are prone to error: any kind of change can be performed, although only a very subtle change was intended. When reviewing code it is difficult to locate casts with searching tools, and it is not always obvious what the intent of the cast is. Wrapping the necessary cast up in a private overloaded inline function makes the meaning clear, and simplifies searching for the usage, and allows a stronger no-casts programming practice to be used elsewhere.

class ComplexNumber
{
  /* ... */
  private:
    inline ComplexNumber& mutate() const
    { return *(ComplexNumber *)this; }
  
  supports use within ComplexNumber member functions as:
  mutate()._polar_value = ...

  This can be implemented by a simple meta-function:
  auto declaration Mutate()
  {
    private $Scope& mutate() const { return *(Scope *)this; }
  }
```

and installed in a class by
class ComplexNumber
{
    $Mutate();
};

Alternatives

The introduction of the mutable storage-class-specifier resolves many of the problems of logically-const const. In the above example, declaring _polar_value as mutable would be a complete solution. However mutable is a storage-class-specifier and not a cv-qualifier and so non-const methods cannot be invoked without some form of cast.

mutate().set_polar_value(...);

The introduction of const_cast<T> resolves the danger of inadvertently casting to a different class, since the type of T can be statically checked. However

const_cast<ComplexNumber>(*this)._polar_value = ...

is a little harder to read, contains a redundant typename, and does not work on old compilers. The improved functionality of const_cast can be exploited to give a more robust meta-function:

auto declaration Mutate()
{
    private $Scope& mutate() const
    { return const_cast<$Scope>(*this); }
}

7.1.4 Clone, Prototype

In languages such as Smalltalk, creating a copy of an object presents no problem, since there is direct language support. In C++, creating a copy of an object whose type is known at compile-time makes use of the copy constructor. However when the type is not statically known, the programmer must provide support code.

Stroustrup on p424 of [Stroustrup97] refers to this support code as a virtual constructor. The technique is also referred to a cloning and forms part of the Prototype pattern [Gamma95].

The support code requires that every concrete\textsuperscript{1} class implements a virtual function to create a clone of itself.

class RootClass
{
    //...
    public:
    virtual RootClass *clone() const = 0;
};
class IntermediateClass : public RootClass
{
    //...
};
class ConcreteClass : public IntermediateClass
{
    //...
    public:
    virtual RootClass *clone() const;
};
RootClass *ConcreteClass::clone() const
{
    return new ConcreteClass(*this);
}

\textsuperscript{1}. a class with no pure virtual methods
Invocation of the virtual function upon an object of unknown type therefore invokes
the appropriate class-specific method to create the clone.

   const RootClass& someObject = ...;
   RootClass *clonedObject = someObject.clone();

Implementation of this idiom requires contributions to the interface and to the
implementation of each concrete class. These will often be in different files,
because of the need to avoid excess include file dependencies. Observance of the
protocol is largely enforced by the use of a pure virtual function. However an
inaccurate implementation can arise through failing to create an instance of the
correct class, or through failing to implement clone when one concrete class
inherits from another concrete class.

A FOG meta-function using a derivation rule can generate all derived class code
automatically

auto declaration Prototype()
{
    public virtual $Scope *clone() const = 0
    {
        derived(!Scope.is_pure()) { return new $Scope(*this); }
    }
}

The clone functionality is woven by a single invocation from the root class.

class RootClass
{
    $Prototype();
}

The pure virtual function is defined in the invoking class. The function body has
an associated derivation rule requiring implementation in all concrete classes.
The function body, and consequently its declaration, is therefore generated in
each concrete class. The use of $Scope within the function body ensures that
resolution of the name is deferred until the function body is installed in its actual
class, and so ensures that a new instance of the concrete class is created.

It is important to use derived(!is_pure()) rather than a less restrictive rule
such as derived(true) to avoid generation of code that creates instances of
abstract classes, since compilers are required to generate error messages if an
abstract class is constructed.

Alternatives

This problem is insoluble without an automatic code generator. Traditional
approaches require extensive use of scaffolding macros.

7.2 Patterns

The general problem of providing implementations of patterns is not soluble,
because patterns are too vague and require tailoring to suit the application
context. However, implementations of patterns suitable for more restrictive
contexts are possible. A few such implementations are presented in the following
sections.

7.2.1 StaticFlyweight

Components are easier to use when components that exhibit similar functionality
provide it in a predictable way. This can be achieved by providing an external

2. $Scope may be changed to $Scope to use the derived type as the return type, which
 is permitted in standard C++ but not allowed in earlier implementations of C++.
interface for a standardised internal behaviour, often realised by the use of virtual functions to provide polymorphic behaviour at run-time. Classes (and more generally declarations) with this form of external interface compatibility are called isomorphic.

The concept of an isomorphic interface at compile-time is not normally used in C++, but lies at the heart of the Standard Template Library, where many of the templates operate on any type that complies with the defined isomorphic interface. The concept of isomorphism is independent of polymorphism and inheritance: the templates work for a variety of independent inheritance trees; there is no need for a common base class.

Provision of meta-functions and meta-variables makes the advantages of isomorphism more visible to the programmer. Families of isomorphic meta-functions can be declared, one per class, to provide the same functionality for the programmer, but using a distinct implementation appropriate to each class.

The Flyweight pattern [Gamma95] describes how shared objects can be used to reduce allocation costs. The pattern comprises a Flyweight manager responsible for managing a pool of Flyweight objects, one of which is returned in response to a request from the client. The manager creates a new Flyweight when no existing one is available.

In the general implementation, the pool may be large and so the factory manager needs some form of map to locate the Flyweights.

A useful variation occurs when the Flyweight objects can be enumerated at compile-time and may have the same life-time as the program. The manager then degenerates to a set of static member functions one per flyweight.

```cpp
class DayOfWeek // ...
{
    public:
        static const Strings& monday()
        { static const String theDay("Mon"); return theDay; }
        static const Strings& tuesday()
        { static const String theDay("Tue"); return theDay; }
    // ...
};
```

The DayOfWeek class makes the textual representation of each day available as flyweight objects, using function scope to define the object lifetime and thereby avoid the race conditions during object construction that can arise from the use of static member variables.

Using a family of isomorphic meta-functions to capture this variant of the Flyweight pattern avoids the need for an application programmer to understand the construction protocols of the relevant flyweight.

```cpp
class String
{ /* ... */
    auto declaration StaticFlyweight{identifier name,
                                                string init = ""}
    { 
        public static const String& ${name}() 
        { 
            static const String staticInstance${init};
            return staticInstance;
        }
    }
};
```

The DayOfWeek class may then be simplified to
Examples

Meta-compilation for C++

class DayOfWeek // ...
{
    $String::StaticFlyweight(monday, "Mon");
    $String::StaticFlyweight(tuesday, "Tue");
    // ...
};

The programming complexity is now partitioned appropriately. Provision of a flyweight requires just a single line. The two to five lines to implement flyweight construction appear just once as part of the flyweight class, rather than repeated throughout each static flyweight manager. Since the construction protocol for the flyweight is encapsulated by the meta-function, different protocols may be used, without affecting the callers, except in so far as additional initialisation arguments might be required. For instance, if an implementation of String makes use of the Flyweight pattern to share all identical strings across an application, the above code might not work. The String class might define private constructors and/or destructors to prohibit static instances. The following may be necessary:

    auto declaration String::StaticFlyweight(identifier name, string init = "")
    {
        public static const String& ${name}()
        {
            static const StringHandle staticInstance($init);
            return *staticInstance;
        }
    }

where StringHandle is a smart pointer to a String.

Alternatives

It is possible to use the preprocessor to solve this problem

    #define STRING_STATIC_FLYWEIGHT(name, init) 
        static const String& name() 
        {
            static const String staticInstance(init);
            return staticInstance;
        }

However the preprocessor approach is unnatural and so programmers tend to avoid its use. Meta-functions fit within the context of the language and so provide a useful addition to a programmer's tool-box.

This kind of problem can sometimes be solved using templates. However in this case the initializer is a string value which is not a legal template parameter, so in the following example the initializer is changed to a character to produce legal code.

    class String
    {
        /* ... */
        public:
        template <const char initString>
        class StaticFlyweight
        {
            public:
            const String& operator()() const
            {
                static const String iT(initString); return iT; }
        };
    };

29-June-2001
class DayOfWeek
{
    /*...*/
public:
    static const String::StaticFlyweight<'M'> monday;
    static const String::StaticFlyweight<'T'> tuesday;
};

The initializer is cached as the template parameter so that access of the flyweight as
DayOfWeek::monday()
invokes
String::StaticFlyweight<'M>::operator()()
which maintains the appropriate flyweight instance. The above code compiles, but
tails to link because no implementation objects were created. It is unfortunately
necessary to duplicate the declarations of the flyweights in the interface by
defining implementations of the functional objects, even though they are never
used and have no content.

const String::StaticFlyweight<'M'> DayOfWeek::monday;
const String::StaticFlyweight<'T'> DayOfWeek::tuesday;

This example shows many of the limitations of templates:

- limited range of parameter types
- more than one line to express the invocation
- relatively obscure implementation

Note that use of operator()() is the only way in which instantiation of a template
can result in a user-defined function name. Since it is necessary to provide a
_corresponding definition, this approach requires two declarations for a user-
defined name, just the same as for renaming with an inline function or a typedef.

When more than a single declaration is required, the template approach fails,
although the preprocessor approach may remain viable.

7.2.2 Member

It is customary to declare member variables and associated support declarations
individually, typically involving

- member declaration
- initialization
- get methods

and sometimes

- set methods

in addition to any actual application functionality. Any intended mode of behaviour
of the member is left to be inferred from the miscellaneous declarations.

For variables with primitive types, coding errors can easily arise from missing
_initializers.

For variables with more complicated types, the associated declarations may be
less obvious.

The standard declarations are easily provided by meta-functions, whose name
demonstrates the programming intent, and whose use offers opportunities for
uniform code evolution along the conventional development path, or for meta-
programming to compose additional functionality.

The three declarations required to define a member variable (name), a protected
set accessor (set_name) and a public get accessor (get_name) are provided by:
auto declaration ScalarMember(identifier type, identifier name, expression init = 0)
{
    private $type $name = $init;
    protected void set_${name}(const $type& aValue)
    { $name = aValue; }
    public const $type& get_${name}() const { return $name; }
};

A similar isomorphic meta-function can be provided to use a smart pointer as the member type:

    template <class T>
    classSmartPointer
    {
    public:
       SmartPointer(T&);
       SmartPointer(constSmartPointer&);
       SmartPointer& operator=(T&);
       SmartPointer& operator=(constSmartPointer&);
       T& operator*();
       const T& operator*() const;
    };

auto declarationSmartPointerMember(identifier type, identifier name, expression init = 0)
{
    privateSmartPointer<$type> $name = $init;
    protected void set_${name}(const $types aValue)
    { $name = aValue; }
    public const $type& get_${name}() const { return *$name; }
};

Member variables and their standard accessors may then be defined as
class Application
{
    /* ... */
    $ScalarMember(bool, is_valid, false);
    $SmartPointerMember(Client, client);
};

The example uses multiple parameters to pass the type, name and initial values. A complete declaration may be passed as

$ScalarMember(bool is_valid = false);
at the expense of a little extra declaration effort:

auto declaration ScalarMember(variable_specifier var)
{
    private $var.type() $var.name() = $var.value();
    protected void set_${$var.name()}(const $var.type& & aValue)
    { $var.name() = aValue; }
    public const $var.type& & get_${$var.name()}() const
    { return $var.name(); }
};

Alternatives

Much of this functionality can be provided by preprocessor macros, however FOG extensions are required to support member or static member initialisation from a single invocation. ISO C++ has relaxed the initialisation rules for static but not for non-static members.

Default initialisation can be enforced by a templated member such as

    Initialized<bool> _is_valid(false);
and `operator()` can be used to support a sensibly named get method. However, further flexibility cannot be provided by templates.

### 7.2.3 ReferenceCount

C++ has no garbage collector. It is therefore the C++ application programmer's responsibility to ensure that allocated memory is appropriately released. For simple forms of object use, allocated memory can be freed later in the same function. For slightly more complicated situations, objects may be organised in trees, with the parent objects assuming responsibility for releasing the resources of children; the application programmer's responsibility is then only with the roots of the trees. In the general case, it is too difficult, inconvenient or even impossible for the program structure to ensure that resources are released.

In the absence of a garbage collector, the problem is resolved by making each object responsible for releasing its own resources. Users of the object register their usage with the object. Then, when no users are registered, the object detects that it is no longer required and self-destructs, releasing the redundant resources. It is not necessary for the object to know the identities of its users, merely their number. This form of resource management is therefore implemented by reference counting: maintaining a count of the number of registered users [Coplien92].

Implementation of reference counting involves two collaborators: the reference counted object and its user. The reference counted object maintains a count of its users, that register their usage, often using a smart pointer. Construction and destruction of the pointer increment and decrement the reference count.

The reference count may be intrusive or non-intrusive.

An intrusive reference count adds an extra member variable to the reference counted object, it

- increases object size
- must be part of the class declaration
- is efficient

A non-intrusive reference count, maintains the count in a disjoint area of memory, possibly accessed by an associative look-up from the object address, possibly at a location known to the user. It

- need not be part of the class declaration
- is inefficient on space and/or time

The example here supports injection of the intrusive functionality. An isomorphic meta-function could be defined for the non-intrusive behaviour.

**External Interface**

In order to support reference counting we need an application class to have an external interface such as

---

3. A version of this example that was tested using the multi-pass implementation of FOG was presented at TOOLS Eastern Europe [Willink99a]. The version presented here uses the improved syntax for derivation rules and clarifies the heap/non-heap allocation policies.
class ApplicationClass
{
protected:
  virtual ~ApplicationClass();
public:
  void annul() const;
  void share() const;
  friend inline void annul(const ApplicationClass *anObject)
  { if (anObject) anObject->annul(); }
};

share() and annul() increment and decrement the reference count, with
annul() provoking self-destruction once all references have been removed. Since
the object manages its own destruction, it is important to prevent deletion by any
other mechanism. In particular

ApplicationClass *countedObject = ...;
delete countedObject;

could cause premature deletion. Use of delete should therefore be trapped as a
compile-time error whenever possible, with a run-time trap as well to double
check. Making the destructor protected ensures a compile-time error

annul() acts as a replacement for delete and so the friend function is provided
to create a closer analogue for delete as

  annul(countedObject);

(An exact analogue could be provided using a syntax macro, but at the expense
of defining annul as a reserved word.)

Internal implementation

The internal representation involves a counter, which is constructed with value 1,
so that decrementing to 0 triggers self-destruction. Care is required to ensure that
copy construction and assignment preserve registrations; the copy constructor
defines the count to one, and the assignment leaves the counts unchanged, since
the number of referenced objects should increase by one on construction and be
unchanged by assignment.

It is convenient to encapsulate this behaviour in a class, so that the
unconventional implementation of copy construction and assignment can be
automatically incorporated into otherwise conventional implementations of the
same functions in client classes.

class ReferenceCount
{
private:
  mutable unsigned int _shares;
public:
  ReferenceCount() : _shares(1) {} 
  ReferenceCount(const ReferenceCount&) : _shares(1) {} 
  ReferenceCount& operator=(const ReferenceCount&) 
  { return *this; }
  ~ReferenceCount() { /* ASSERT(_shares == 1); */ }
  bool annul() const
  { return (_shares == 1) ? false : (_shares--, true); }\footnote{This rather contrived form of conditional ensures that there is only one return state­
  ment, which is a prerequisite for inlining by some compilers.}
  bool heap_only_annul() const { return --_shares != 0; }
  void share() const { _shares++;
  }
  unsigned int shares() const { return _shares;
  }
};

29-June-2001
The destructor may beneficially validate that the share count is 1. This may detect a premature deletion through the use delete, or some more obscure problem such as double destruction through some unpleasant pointer recursion. The implementation of share() performs the increment for registration of an additional user. The implementation of annul() collaborates with the counted object to decrement when a usage is removed. The return status indicates whether the counted object should continue in existence. The count is not decremented if destruction is due, in order to ensure consistent count behaviour for the three following usage patterns:

```cpp
void test()
{
    static ApplicationClass staticallyAllocatedObject;
    ApplicationClass stackAllocatedObject;
    ApplicationClass *heapAllocatedObject = new ApplicationClass();
    // ...
    annul(heapAllocatedObject);
}
```

For the statically and stack allocated objects, construction and destruction bound the lifetime, any additional sharing through annul and share is optional. Only the statically allocated object may live on beyond the function return. For the heap allocated object the life-time of the object terminates with respect to the local function with the annul, but any additional registration in the commented section may prolong the lifetime. If only heap usage is required, the simpler implementation of annul() as heap_only_annul() is adequate.

**Glue code**

Installing the ReferenceCount into an ApplicationClass class requires the following code to convert the partial functionality of the ReferenceCount class into the required external interface of the ApplicationClass class:

```cpp
class ApplicationClass
{
private:
    ReferenceCount _shares;
protected:
    virtual ~ApplicationClass();
public:
    void annul() const
        { if (!_shares.annul()) delete (ApplicationClass *)this; }
    void share() const { _shares.share(); }
friend inline void annul(const ApplicationClass *anObject)
        { if (anObject) anObject->annul(); }
    // ...
};
```

This is rather too much to be entered by the programmer. In the past, the author has used a preprocessor macro for all except the virtual destructor which has been dealt with manually. A meta-function provides a more powerful solution:

```cpp
auto declaration ReferenceCount::install(bool heapOnly = false)
{
    auto if (!defined(has_reference_count)) //1.1
    {
        auto bool has_reference_count = true; //1.3
        private ReferenceCount _shares; //2.1
        public void share() const { _shares.share(); } //2.2
        auto if (!heapOnly) //2.3a
        {
            public void annul() const
                { if (!_shares.heap_only_annul()) //2.4a
                    //2.5a
```
The meta-function may be invoked as

```cpp
class ApplicationClass /* ... */
{ /* ... */
  $ReferenceCount::install();
};
```

leaving little opportunity for error or misunderstanding.

Lines 2.1 to 2.8 of the meta-function closely follow the glue code above, save for the use of per-declaration rather than prefix access-specifiers (Section 3.1.3.2) and for the use of $Scope to access the invoking class name.

The meta-function contains an outer conditionalisation on lines 1.1 to 1.4 to guard against double installation. When first invoked, the has_reference_count meta-variable is not defined, and so the conditional succeeds. The meta-variable is then defined on line 1.3, and any reinvocation of the meta-function within the ApplicationClass class or its derived classes is suppressed by the conditional.

The guard code is followed by the definition of the member variable (2.1), two member functions (2.2, 2.4) and a friend function (2.7). The _shares member variable provides the run-time storage needed by the share count, and interacts via its constructors and destructors with the equivalent functions for the reference counted class. The share() member function simply delegates the member variable. The annul() member function completes the share counting protocol by deleting the reference counted object when the final share is removed. The annul() friend function just provides a more convenient destruction option avoiding the need to worry about null pointers.

The inner conditionalisation (2.3) upon the formal parameter heapOnly selects between declarations that inhibit non-heap object construction, and the more flexible default behaviour allowing static as well as heap objects.

Line 2.6a defines the destructor as protected in the invoking class and as a result of the derivation rule, in all derived classes. This provides a compile-time check to catch most attempts to create counted objects statically or on the stack, which might other wise have a lifetime extended either through a start-up/shutdown race condition or beyond the return of a function. The virtual keyword is omitted to avoid prejudicing the class designer's decision on whether the destructor should be virtual. Although the destructor should probably be virtual, it is not appropriate for a meta-function to enforce 'good' style [Meyers92] when there may be legitimate reasons for an alternate style.

Accidental double installation within the same class may seem unlikely for conventionally structured code, however when more than one meta-function is invoked as in

```cpp
class ApplicationClass /* ... */
{ /* ... */
  $NonIntrusiveList::install();
  $NonIntrusiveMap::install();
};
```

Meta-compilation for C++
a multiple installation could occur indirectly. Accidental double installation within an inheritance tree can easily occur, and is partially trapped by the inheritance of has_reference_count. The guard is not proof against a later installation of a reference count into a base class, or against multiply inherited reference counts. Solutions to these problems require installation to be split into two phases, first to determine the overall requirements, and then to generate the corresponding declarations. The Monitor example in Section 7.4.2 shows how the more general problem can be resolved, without affecting application or glue code.

Alternatives

There is an efficient solution to this problem, for the case of unconstrained usage from a well-defined class, using the Curiously Recurring Template Pattern [Coplien95a] in which a derived class parameterises its base class.

```cpp
class ApplicationClass : /* ... */,
    public ReferenceCounted<ApplicationClass>
{ /* ... */ }
```

In this application, the template parameter is required to support a static_cast from a pointer to the base ReferenceCounted<ApplicationClass> to a pointer to the derived ApplicationClass for use by the delete in a complete rather than partial implementation of annul().

```cpp
template <class T>
class ReferenceCounted
{ /* ... */
    void annul() const
    {
        if (!--_shares)
            delete static_cast<T*>(this);
    }
};
```

Alternatively, if the compulsory expense of the virtual function table is acceptable, a non-templated version of ReferenceCounted can be written using a virtual destructor and no cast.

If protected destructors are used to enforce heap-only usage, an extra friend declaration is required to allow the base class access to the derived destructor, violating the goal of a single mention in the instantiation glue (unless a meta-function is used to define base class and friend).

A non-templated approach is more flexible, and necessary when multiple inheritance is involved:

```cpp
class ApplicationClass : /* ... */,
    public virtual ReferenceCounted
{ /* ... */ }
```

The latter approach can always be used, but at the expense of an extra indirection for the much commoner single inheritance cases.

The meta-programming approach supports an arbitrary request for reference counting functionality, leaving the meta-program to choose an efficient implementation strategy. The conventional approach requires the programmer to choose and implement the strategy directly.

7.2.4 WholePart

Soukup [Soukup94] makes a persuasive case for implementing patterns using pattern classes. A pattern class is just a grouping of static functions templated by the types of each collaborator as shown in Figure 7.1.
The pattern class contains no member variables and is a friend of each collaborator. A pattern is used by invoking the static member function of the pattern class with the appropriate collaborator instances as parameters.

```cpp
class C1
{ /*...*/
  friend class P1<C1, C2>;
};
template cclass T l, class T2>
void P i::do_something(T1& c l, T2& c2)
{
  c1.poke = c2.peek;
}
```

Since the pattern class is a friend of each collaborator, the function is free to peek and poke the working variables in the collaborator objects to perform the required actions. This approach achieves a very regular style of implementation, and has very beneficial effects in reducing include file dependencies. However the extensive use of friend declarations runs counter to normal programming practice.

Instantiation of a pattern using a pattern class requires instantiation of the pattern class, and insertion of friendship declarations and working variables into the collaboration classes. Instantiation of the pattern class is readily resolved by conventional C++ template instantiation. Insertion of declarations into the collaborators could be performed by manual editing. [Soukup94] describes a custom preprocessor for the CodeFarms library that performs this insertion automatically, provided the programmer has left a hook in each class of the form.

```cpp
class State
{
  MEMBER_State // Hook to enable Cpp to insert declarations
  /*...*/
};
class Town
{
  MEMBER_Town // Hook to enable Cpp to insert declarations
  /*...*/
};
```

The custom preprocessor scans pattern class instantiations such as

```
WHOLE_PART(State,Town) // Declaration that State has many Towns
```

to produce conventional Cpp definitions such as

```
#define MEMBER_Town
friend class WholePart<State,Town>; 
State * _whole;
```
Multiple patterns are readily accommodated; the generated Cpp macro just grows. This approach demonstrates that implementation of a particular pattern solution requires that declarations be injected into the code for collaborator classes. Only for the degenerate case of a pattern involving a single class can injection be avoided. In the terminology of Aspect Orientation, the declarations associated with each aspect (or pattern) must be woven together to create composite declarations acceptable to the C++ compiler. FOG supports this weaving and eliminates the need for a custom preprocessor and for the preprocessor hooks that support the CodeFarms library.

```cpp
template <class Whole, class Part>
auto declaration WholePart::install()
{
    class $Whole
    {
        friend class $Dynamic;
        private list<$Part> _parts;
        /* optional construction, delegations and destruction */
    };
    class $Part
    {
        friend class $Dynamic;
        private $Whole *_whole;
        /* optional construction, delegations and destruction */
    };
}
```

Invocation of the installation function as

```
$WholePart<State,Town>::install();
```

just adds the required friend declaration and member variable to each collaborator class identified by the template parameters.

The semantics of meta-function execution involve replacement only of formal parameters in the scope of the meta-function, before returning the declarations in the meta-function body for interpretation within the invocation scope. The meta-function apparently has no formal parameters. It actually has four (see Section 4.3.11). All meta-functions have two built-in formal parameters Static and Dynamic corresponding to the declared scope of the meta-function and the actual scope, which differs if invoked for a derived class. In addition, each template parameter is also a formal parameter. The usage of $whole rather than whole therefore ensures that a replacement occurs before the body is returned to the calling scope where whole may be undefined or differently defined.

The example shows only the minimum to activate the pattern solution. Additional declarations could enforce appropriate construction and destruction protocols, and provide delegation so that users are unaware that a pattern class is in use.

It is not necessary to use Soukup's pattern class approach, although it has a pleasant symmetry. Installation can be organised with respect to a dominant collaborator, probably the Whole class in this case. It is then only necessary to perform code injection into the other collaborators. However, whatever approach is adopted, a pattern solution with more than one collaborator class requires either manual editing to spread the pattern solution or automatic code injection.
7.2.5 Visitor

An implementation of the Visitor pattern [Gamma95] provides a more complete example of some of the facilities of FOG.

The pattern involves a hierarchy of data classes (DerivedData1, DerivedData2, etc.) and a number of algorithms that may be performed on the data. The algorithms are realised by algorithm classes (DerivedAlgorithm1, DerivedAlgorithm2, etc.) and inherit from the abstract Visitor class AbstractAlgorithm. The data classes similarly inherit from an abstract class AbstractData.

Usage of the pattern requires the appropriate data and algorithm dependent action to be performed. This is achieved by an invocation of the virtual method AbstractData::accept(AbstractAlgorithm&), whose derived implementation invokes AbstractAlgorithm::accept_DerivedData1(DerivedData1&), which can in turn have a derived implementation to perform the required action as DerivedAlgorithm1::accept_DerivedData1(DerivedData1&).

In a system with A algorithms and D data classes, there may be as many as A*D functions to be declared and implemented. The implementations in the algorithm classes performs the required actions in response to the two dimensional dispatch.

The scaffolding code required in the data element classes can be generated automatically by FOG.

The pattern has two degrees of freedom, which contributes to the inconvenience of a conventional manual approach. Addition of an extra algorithm class is relatively benign, requiring just that the new algorithm class implements as many of the data functions as required. Addition of an extra data class requires that the data class complies with the inherited protocol and that an additional method be defined for the abstract algorithm. It may also be necessary to implement this method in every derived algorithm class.

The example implementation uses two meta-functions, one to be invoked in the root data class (AbstractData), and another to be invoked in each derived data class (DerivedData1).

A derivation rule cannot satisfy the derivation requirements of this pattern, since a derivation rule can only regenerate potential declarations in derived contexts. In this case, derivation of a data element class (DerivedData1) needs to export a declaration to the abstract visitor class (AbstractAlgorithm). A generalisation of the rule could be considered, but appears to be necessary only for this pattern. The cost of a one-line invocation in each derived data element class is small compared to the associated response code in the algorithm classes, and provides flexibility for deliberate omission if abstract levels of the data element hierarchy do not need visitor support.

---

29-June-2001
auto declaration VisitorBaseElement(identifier V) {
    public typedef $V Visitor;
    public virtual void accept($V & aVisitor) = 0;
};

auto declaration VisitorDerivedElement() {
    public virtual void accept(Visitor & aVisitor) {
        aVisitor.accept_${Scope}(*this);
    }
    public virtual void Visitor::accept_${Scope}($Scope & aData) {
    }
};

The abstract classes may invoke the pattern as:

```
class AbstractAlgorithm /* ... */
{
    // ...
};

class AbstractData /* ... */
{
    $VisitorBaseElement(AbstractAlgorithm);
    // ...
};
```

Invocation of VisitorBaseElement defines the AbstractData::Visitor typedef with the value AbstractAlgorithm, avoiding the need to pass the identity of the visitor class to invocations in derived classes:

```
class DerivedData1 : /* ... */ public AbstractData /* ... */
{
    $VisitorDerivedElement();
    // ...
};

class DerivedData2 : /* ... */ public AbstractData /* ... */
{
    $VisitorDerivedElement();
    // ...
};

class DerivedDerivedData : /* ... */ public DerivedData2 /* ... */
{
    $VisitorDerivedElement();
    // ...
};
```

Definition of the derived algorithm classes, like the abstract algorithm classes requires no explicit code. The declarations are provided automatically by the second declaration in the VisitorDerivedElement meta-function, which also provides a default empty algorithm implementation. Since the signatures of the algorithm are defined in the base class, implementation of the derived algorithm code need only mention the name.

```
class DerivedAlgorithm1 : public AbstractAlgorithm
{
    using accept_DerivedData1
    {
        // ...
    }
};
```

The short form using-declaration requires usage within class braces. Independent usage can be provided by a long form using-declaration at the expense of a double mention:
using accept_DerivedData1 DerivedAlgorithm1::accept_DerivedData1
{
    // ...
}

or directly at the cost of requiring distributed editing to change a signature:

void DerivedAlgorithm1::accept_DerivedData1(DerivedData1& aData) {
    // ...
}

The pattern is expressed compactly, and instantiated so that its use is clear. Compliance with the pattern is ensured because the pattern provides all the relevant declarations. The manually contributed code is reduced to that necessary to provide the actual implementation. The scaffolding is almost completely removed.

7.3 Meta-Programming

7.3.1 OS Traits

Conditional compilation is essential to support a variety of configuration options, often to resolve distinctions between different operating systems. Control values are variously predefined by the compiler, supplied by command line or defined in header files.

```cpp
#ifdef UNIX
    static const char *temp_path = "/tmp/";
#else
    static const char *temp_path = "C:\\Temp\\";
#endif
```

C++ statements occur only within functions and express evaluations to be resolved at run-time. In FOG, meta-statements are declarations and so the example simplifies to:

```cpp
auto bool unix = std::get_cpp("UNIX") != ";
auto if (unix)
    static const char *temp_path = "/tmp/";
else
    static const char *temp_path = "C:\\Temp\\";
```

The invocation of `std::get_cpp` looks up `UNIX` in the C preprocessor namespace, providing controlled access to command line definitions.

Using an Object-Oriented perspective eliminates the need for conditionalisation. The characteristics of each configuration option may be packaged as meta-variables (and meta-functions) of a (meta-)class, extending the traits concepts of [Myers95].

```cpp
auto class OsTraits_Abstract {
    auto static bool NT = false; // default value
    auto static bool UNIX = false; // ...
};
```

5. These autos were omitted from the presentation in Section 1.5.1, where, as a result, classes were defined. The extra autos here declare that only the meta-classes are required, so avoiding the eventual emission of unnecessary C++ declarations.
auto\textsuperscript{5} class OsTraits_Nt : auto\textsuperscript{5} OsTraits_Abstract
{
    auto static bool NT = true; // overriding value
    auto static string temp_path = "C:\\Temp\\";
    //...
};

auto\textsuperscript{5} class OsTraits_Unix : auto\textsuperscript{5} OsTraits_Abstract
{
    auto static bool UNIX = true;
    auto static string temp_path = "/tmp/";
    //...
};

The appropriate configuration may then be chosen using \texttt{std::get\_cpp("OS")} to resolve \texttt{OS} from the command line. Thus

\begin{verbatim}
auto\textsuperscript{5} class OsTraits : auto\textsuperscript{5} OsTraits$_\texttt{std::get\_cpp("OS")}$ {}
\end{verbatim}

maps the required configuration to OsTraits. The appropriate operating system may be specified on the FOG command line by:

\begin{verbatim}
fog ... -D OS=Nt ...
\end{verbatim}

A file may then be opened on the operating system specific temporary path by:

\begin{verbatim}
std::ofstream s($OsTraits::temp_path "results.dat");
\end{verbatim}

(The pair of strings concatenate to give the required file name).

Having isolated the configuration in separate classes and an associated header file, a new operating system can be supported by providing a prefix file characterising the new system and invoking it with an appropriate command line. Existing source files need no change.

\section*{Alternatives}

This could be achieved directly using multiple layers of name substitutions with C preprocessor, but it never is. Modularization is much easier when supported by the programming environment.

This cannot be achieved by templates, which lack the ability to perform string manipulations.

\subsection*{7.3.2 SynchronisedEnum}

In Section 4.4.7 on page 123, an example was given showing how a meta-function could transform a list of enumerations into an array of text strings. An alternative approach may sometimes be preferable, defining additional enumerators and text array elements incrementally.

For instance a compiler may be structured so that the sub-algorithms for each AST traversal algorithm are placed with one complete algorithm per file, and an enumeration may be used to identify which algorithm is in active. Addition of an additional algorithm should then be possible with the minimum of disruption to other code. The declaration of the enumerator for the constant folding pass is beneficially performed by a meta-function invocation such as \texttt{\$install\_ast\_traversal(constant\_folding)} that resolves as many of the shared declarations as possible.
class ApplicationClass
{
    public enum Enum {}
    public static const char *texts[] = {};
    auto declaration define(identifier aName, expression aValue)
    {
        enum Enum { $aName = $aValue };
        static const char *texts[] = { [aValue] ""$aName };
    }
};

Application code may then define enumerators with explicit values (and text elements) by:

$ApplicationClass::define(LABEL, 40);

The example can be usefully simplified and made more reliable by using the sequentially allocated enumerator values directly:

auto declaration define(identifier aName)
{
    enum Enum { $aName };
    static const char *texts[] = { ""$aName };
}

Alternatives

The above example cannot be expressed in C++. Conventional practice requires that a maintainer update both enumeration and array of text strings consistently. The FOG pattern updates both at once, and provides freedom for each invocation of ApplicationClass::define to be located with code associated with the invocation, avoiding the need to fragment code to satisfy compiler constraints.

7.3.3 RTTI

Prior to the adoption of RTTI into C++, it was necessary for large Object Oriented programmers to implement RTTI as part of the application code. Each third party library had its own policy, which imposed significant compliance burdens upon consumers. RTTI within the language eliminates much of the add-on functionality, however it pursues the C++ philosophy of avoiding costs of unused functionality by providing only minimal functionality. Extra functionality must still be provided extra-lingually, using the unique RTTI type identifier as an index to custom information. Applications may therefore still need to provide RTTI, albeit tailored to exploit the built-in functionality.

FOG meta-programming supports conversion of declarations into a format suitable for use at run-time.

A simple example supporting just class name diagnostics and inheritance tables is provided here. A more extensive example involving formatting of member variable names is provided to support marshalling in Section 7.4.1.

Class-name information

Class-name information may be provided by:

class Rtti;

auto declaration Rtti::class_names()
{
    virtual const char *class_name() const
    :
      { derived (true) { return ""$Scope; } }
}
and invoked as
class ApplicationClass
{
    $Rtti::class_names();
}

to have class_name() return the class name for ApplicationClass and all its
derived classes.
This does not work for template classes, whose full name is not known at meta-
compile-time. A rather more elaborate approach is necessary that concatenates
the names of each argument during construction of a static instance within the
invoked method.

Inheritance information

We are trying to generate code such as:

class MyClass : public Basel, protected Base2
{
    public:
        virtual const Rtti& dynamic_type_info() const //1
            { return static_type_info(); }
    static const Rtti& static_type_info() //2
        {
            static const Rtti rtti("MyClass", _base_info);
            return rtti;
        }
    protected:
        static const Rtti::BaseInfo _base_info[] = //4
        {
            //7
            { 
                &Basel::static_type_info(),
                (char *)&(Basel&)(MyClass&)* (char *)0x1000-(char *)0xl000 },
                &Base2::static_type_info(),
                (char *)&(Base2&)(MyClass&)* (char *)0x1000-(char *)0x1000 },
            { 0,
                0 }
        };
    }

The dynamic_type_info and static_type_info methods provide the class
description in a similar way to the get_info and info methods on p448 of
[Stroustrup91]. dynamic_type_info is a virtual function and so returns the
dynamic type of a pointer, whereas static_type_info is static and so returns the
declared type.

The detailed class description uses a null-terminated list of Rtti::BaseInfo base
class descriptors to define the context of a class with respect to its bases. The
descriptor provides two fields, one that points to the base class, and the other that
identifies the offset of the particular base class object within the enclosing object.

The horrendous sequence of casts establish a phantom MyClass instance at
address 0x1000 upon which offset calculations are performed. (The phantom
object can be placed at any address other than the obvious 0x0 for which C++
mandates that all offset arithmetic returns 0.)

class Rtti
{
    public:
        struct BaseInfo // List element in list of bases
        {
            const Rtti *base_rtti; // Base class description
            int _offset_to_this; // Base position
        };

Page 238 29-June-2001
Examples

Meta-compilation for C++

Rtti(const char *className, const BaseInfo baseList[]);

auto declaration Rtti::base_names()
{
    public virtual const Rtti & dynamic_type_info() const //1
    { derived (true) { return static_type_info(); } }
    public static const Rtti & static_type_info() //2
    { derived (true)
        { static const Rtti rtti("@Scope, _base_info"); //3
            return rtti;
        }
    }
    protected static const Rtti::BaseInfo _base_info[]; //4
    { derived (true); }
    auto ~${Scope}(){ //5
    
    auto for (iterator b = $bases(); b; ++b) //6
        using _base_info = { (char *)&$(b)::static_type_info(), //7
            (char *)&($b&){$Scope}&*(char *)0x1000 - //8
            (char *)0x1000
        };
    using _base_info = { { 0, 0 } }; //9
    
};

The functionality is installed in the same way as before with a single line at the root of an inheritance hierarchy:

class ApplicationClass
{
    $Rtti::base_names();
};

as a result of which ApplicationClass has two methods static_type_info (2) and dynamic_type_info (1), a null-terminated array of base class descriptors (4) and a meta-destructor (5). The methods and array have derivation rules with a true predicate, these methods and the array are therefore regenerated in all derived classes. Construction of the local Rtti object (3) uses $Scope ensuring that the derived class name is used.

The meta-destructor executes for each derived class. It loops over all base classes (6). Each iteration adds a list element comprising a pointer to the base-class type information (7), and an offset of the base-class within the derived object (8). Finally the list is null-terminated (9). Each using exploits the extended re-using-declaration (Section 3.1.4.4) to refer to protected static const Rtti::BaseInfo _base_info[] as just _base_info before specifying an additional array element to be composed (Section 4.4.7).

This technique generates predictable functionality in derived classes automatically, and so compares very favourably with traditional approaches relying on multiple preprocessor scaffolding macros per class.

7.4 Aspects

Aspect Oriented Programming [Mens97] seeks to isolate independent programming concerns as aspects, each of which can be implemented (and re-used) independently. The next two examples show partitioning of a problem into the application, some additional concern implemented as an aspect, and a very small amount of glue code which initiates weaving the code for each aspect into the main application code. The examples also demonstrate how application code
is simplified and consequently made more reliable by the initial expenditure of extra declarative effort to achieve re-use. This is a natural extension of OO philosophy, where a class can encapsulate difficult concepts making use easy although definition difficult. FOG provides additional declarative power supporting better encapsulation.

7.4.1 Marshalling

Communication between programs requires messages to be passed between those programs. Each message is usefully represented as an object, and so the programmer is presented with the problem of transferring the contents of one object between programs. This is readily achieved using an Interface Definition Language and CORBA when such high level facilities are available, however when working at a lower level, as often occurs for embedded systems, the problem must be solved by the programmer.

A typical approach involves the conversion of each object into a sequence of bytes with a common header that describes the format and length of the subsequent bytes. The sending program must marshal the data elements of each object into the byte stream and the receiving program must perform the corresponding unmarshalling back into an object. Preparation of this marshalling code is straightforward, but not amenable to automation with conventional compilers. In order to show how this can be resolved by FOG, it is helpful to first show one possible conventional solution. The exposition matches the subsequent automated solution. Numbered comments (//3.0) may assist the reader in correlating the two solutions.

All messages inherit from the Message class, which defines the marshalling and unmarshalling interfaces and an enumeration, whose values distinguish between possible message formats.

typedef unsigned char uchar;       // Short name for shorter lines

class Message
{ /*...*/
  protected:
    enum MessageTypes
    {
      MESSAGE_StockReport /*,...*/
    };
  public:
    virtual size_t marshal(uchar data[]) const;  //2.0
    static Message *unmarshal(uchar data[]);     //3.0
  };

Invocation of the marshal function fills data[] with the byte stream and returns the message size. The unmarshal function is passed a byte stream and returns a pointer to an object if the message is valid, or 0 on failure. A single very simple message comprising just two data elements is used for this example.

---

6. A version of this example that was tested using the multi-pass implementation of FOG was presented at TOOLS Eastern Europe [Willink99b]. That version used ad hoc token pasting within function bodies. The version presented here exploits the polymorphic behaviour of token lists available with the superset implementation.
class StockReport : public Message
{
    /* ... */
private:
    unsigned long _item_number;
    short _stock_level;
private:
    inline StockReport(uchar data[]); // 5.0
public:
    static StockReport *make(uchar data[]); // 4.0
    virtual size_t marshal(uchar data[]) const; // 2.1
};

The message-specific marshalling into data[] is performed by a virtual function:

size_t StockReport::marshal(uchar data[]) const // 2.2
{
    uchar *p = data; // 2.3
    *p++ = MESSAGE_StockReport; // Message type // 2.4
    *p++ = 6; // Message body length // 2.5
    *p++ = (_item_number >> 24) & 0xFF; // 2.6
    *p++ = (_item_number >> 16) & 0xFF;
    *p++ = (_item_number >> 8) & 0xFF;
    *p++ = _item_number & 0xFF;
    *p++ = (_stock_level >> 8) & 0xFF; // 2.7
    *p++ = _stock_level & 0xFF;
    return p - data; // 2.8
}

Unmarshalling from data[] selects the message-specific routine:

Message *Message::unmarshal(uchar data[]) // 3.1
{
    switch (data[0]) // 3.2
    {
        default: // 3.3
            return 0; // 0 for bad message type error,
        case MESSAGE_StockReport: // 3.5
            return StockReport::make(data);
            /* ... */ // 3.6
    }
}

Then the message-specific object is created, but only if the length is valid:

StockReport *StockReport::make(uchar data[]) // 4.1
{
    if (data[1] != 6)
        return 0; // 0 for bad message length error.
    else
        return new StockReport(data);
}

Finally, construction performs the message-specific unmarshalling:
StockReport::StockReport(uchar data[]) {
    uchar *p = data+2);
    unsigned long temp = *p++;
    temp = (temp << 8) | *p++;
    temp = (temp << 8) | *p++;
    _item_number = temp;
    }
    unsigned long temp = *p++;
    temp = (temp « 8) | *p++;
    _stock_level = short(temp);
}

The marshalling and unmarshalling code is very predictable and in principle easy to write, however when there are many messages, it is tedious and error prone. When a data type is changed or a member variable added, there are many places where updates are required. It is preferable to generate the code automatically. This requires a meta-program that can reflect upon the message class declarations and generate code accordingly.

Application aspect

The marshalling support may be separated completely from the application code. The message classes express their own inheritance relationships, their data contents, and any other application declarations that may be necessary.

typedef unsigned char uchar; // Short name for shorter lines
class Message { /* ... */ ;

class StockReport : public Message
{ /* ... */
private:
    unsigned long _item_number;
    short _stock_level;
};

Aspect weaving

The marshalling aspect is added (woven) by invoking the installation meta function of the Marshal meta-class.

using "Marshal.fog";
class Message
{  
	$Marshal::install();
};

Marshalling aspect

The installation meta-function is:
auto class Marshal {;
auto declaration Marshal::install()
{
  auto type MessageClass = $Scope; //10.1
  auto static statement switchBody = //10.2
  {
    default: //3.4
      return 0;
  }
  auto number byte_count = 0; //10.3
  protected enum MessageTypes {}; //1.0
  public virtual size_t marshal(uchar data[]) const //2.0/1/2
  {
    derived(true) entry
    {
      uchar *p = data; //2.3
      *p++ = MESSAGE_$Scope; //2.4
      *p++ = byte_count; //2.5
    }
    derived(true) exit
    {
      return p - data; //2.8
    }
  }
  public static !inline $Scope *unmarshal(uchar data[]); //3.0/1
  {
    switch (dataBuffer[0]) //3.2
    switchBody;
  }
  public static $Scope *make(uchar data[]) //4.0/1
  {
    derived(true)
    {
      if (*p++ != byte_count)
        return 0;
      else
        return new $Scope(data);
    }
  }
  auto $($Scope)() //11
  {
    protected enum $(MessageClass)::MessageTypes
    {
      MESSAGE_$Dynamic; //1.1
    auto switchBody +=
        case MESSAGE_$Dynamic:
          return $(Dynamic)::make(data);
    private inline/implementation
    $(Dynamic)(uchar data[]) //5.0/1
    {
      uchar *p = data+2; //5.2
    }
  }
  auto ~$($Scope)() //12.0
  {
    auto for (iterator i = $all_variables(); i; ++i) //12.2
    auto if (!i->is_static()) //12.3
      $i->type().marshal($i->id()); //12.4
  }
}

The install meta-function is invoked from a class declaration for Message and executes as part of the source file reading and analysis compilation phase. All
lines declare declarations that are added to the Message class. Three meta-
variables (10.1, 10.2, 10.3), a meta-constructor (11) and a meta-destructor (12.0)
are declared in addition to more conventional declarations (1.0, 2.0, 3.0, 4.0).
The first meta-variable (10.1) caches the name of the invocation scope in the
meta-variable MessageClass for access by the meta-constructor.
The second meta-variable (10.2) defines the switchBody meta-variable, which
will accumulate the switch cases for the unmarshal routine. It is initialized with a
list comprising the default case.
The third meta-variable (10.3) defines byte_count which will be incremented with
a count of the bytes required to express the marshalled data. It is initialized to
zero.
Note that byte_count is non-static since each derived class has a distinct size,
whereas switchBody is static because all derived classes should contribute to
the single base class list.
The enumeration of message types is declared (1), ready for extension by the
meta-constructor.
The framework of each marshalling routine is defined (2), with a derivation rule to
ensure regeneration in each derived class. The framework defines three lines of
code for the function entry, and one line for the function exit. The function body
will be defined during meta-destruction.
The unmarshalling routine is defined (3) in its entirety, using a deferred @ to
reference the switchBody after meta-programming has defined its content. There
is no need for braces around the meta-variable, since FOG automatically supplies
the appropriate brace/parenthesis/comma punctuation when emitting a list as part
of a C++ declaration.
The make routine is similarly defined in its entirety (4) embedding the calculated
count, using a deferred reference to await its determination by meta-programming
and of the copy appropriate to the particular derived class, since the derivation
rules ensures that the make function appears in all derived classes.
The meta-constructor is invoked for Message and all its derived classes during the
meta-construction compilation phase. Invocations occur in least derived first
order. The meta-constructor first defines an enumerator and then appends a
switch case.
An enumerator is defined (1.1) in the MessageTypes enumeration of the Message
class. The additional enumerator extends the enumeration and so acquires a
unique value for each message class. The enumerator name is formed by
concatenation of the prefix MESSAGE_ and $Dynamic, the class name of the
derived meta-constructor (the derived message class).
The additional switch case is defined (3.5) and appended to the switchBody
meta-variable, thereby extending the unmarshal routine that uses it. The
invocation of the Dynamic built-in variable ensures that the identity of the derived
rather than base message class is used. Note the absence of braces to ensure
that a case rather than a list of one case is appended.
The meta-constructor then defines the interface and the start of the
implementation for the derived message class constructor, on whose behalf the
meta-constructor is executing.
The meta-destructor is similarly invoked on behalf of each message class, during
the meta-destruction phase, by which time all member variables have been
defined. It comprises a loop to resolve the member variable dependent code and
count the number of bytes in the message.
The byte count is maintained in a meta-variable initialised to 0 (10.3). The loop
(12.2) iterates over all member (and inherited member) variables of the derived
class, and (12.3) skips static member variables. Within the loop (12.4), invocation
of the marshal meta-function for the data-type of each member variable causes emission of member-specific marshalling code. The member variable name is passed as a parameter to a type-specific implementation such as:

```cpp
auto declaration unsigned long::marshal(expression name) //13.0
{
    byte_count += 4; //13.1
    public virtual size_t marshal(uchar data[]) const //2.2
    {
        *p++ = ($name >> 24) & 0xFF; //2.6
        *p++ = ($name >> 16) & 0xFF;
        *p++ = ($name >> 8) & 0xFF;
        *p++ = $name & 0xFF;
    }
    private $(Scope)(uchar data[]) //5.1
    {
        unsigned long temp = *p++;
        temp = (temp << 8) | *p++;
        temp = (temp << 8) | *p++;
        temp = (temp << 8) | *p++;
        $name = temp;
    }
}
```

Meta-functions can be defined for built-in types as well as user defined types. The above declaration for the unsigned long 'class' supports the marshalling of unsigned long member variables. The formal parameter name is replaced throughout the body before the body is interpreted in the invoking context, that of the derived message class. The update of the byte_count (13.1) therefore maintains the counter of the derived message class, and the two declarations (2.2, 5.1) provide additional code for the body region of the derived message class routines. The member variable iteration is in declaration order, and the ordering of function body contributions is preserved, so the final ordering of the many contributions is well-defined. The contributed code (2.6, 5.3) just performs the very simple operations appropriate to the data type.

A similar isomorphic meta-function for short is needed to complete the example (2.7, 5.4), and further routines for every other primitive data type. Nested data types can be resolved by a nested iteration, which can be specified as a general-purpose meta-function, passing the nested member name to the nested call, necessitating the use of expression rather than name or identifier for the parameter type.

```cpp
auto declaration Marshal::marshal(expression name)
{
    auto for (iterator i = $all_variables(); i; ++i)
        auto if (!i->is_static())
            $i->type().marshal(${name}.i->id());
}
```

The general purpose meta-function can be installed by meta-inheritance for use in a nested type

```cpp
struct NestedDataType : auto Marshal { /* ... */ };
```

This declares Marshal as an additional base class of the NestedDataType, but only at (meta-)compile time. The meta-names of Marshal are therefore visible to the derived class, providing the required resolution of NestedDataType::marshal().

This example shows how application code can be generated in response to the actual application declarations. The code is fully under the programmer's control. The programmer can freely choose an alternate implementation using data tables.
to describe each message rather than monolithic functions. Inheritance of messages can be exploited to trade size for speed, by changing compile-time iterations to serve only the local member variables, and changing the run-time code to invoke base class methods for inherited members. More sophisticated code can be provided to support swizzling of pointer types for database applications.

The generated code is portable, since all members are referred to by name. The example code for unsigned long::marshal has a portability problem for processors with a greater than 32 bit unsigned long, but this is a limitation of the example solution, not of the approach.

7.4.2 Monitor

The ability to use FOG to separate different programming concerns is demonstrated by application of a synchronisation monitor to a stack. One line of glue code is necessary to weave the otherwise independent functionality of monitor and stack.

Application Aspect

We first define a simple stack class.

```cpp
template <class T>

class Stack
{
    $NoCopy();
    $NoAssign();

    private:
    T *_elements ;
    size_t _capacity; // Allocated size of _elements[]
    size_t _tally;  // Used size of _elements[]

    public:
    Stack() : _elements(0), _capacity(0), _tally(0) {}  
    -Stack() {/*...*/}
    bool is_empty() const volatile { return _tally == 0; }
    T pop() { /*...*/}
    void push(const T&) { /* ... */ }
    T top() const { return _elements[_tally-1]; } }
```

The const qualifier is used conventionally to indicate that no change occurs. Concurrent readers are therefore permissible, but concurrent writing should not be permitted once a monitor aspect has been added.

The volatile qualifier is used to indicate that access may occur without the use of a lock, allowing interleaved reading and writing by other threads.

The Monitor Aspect (run-time)

The monitor functionality is provided by a Monitor class, whose detailed implementation is not relevant to this example.

---

7. A version of this example, that was tested using the multi-pass implementation of FOG, appears in a position paper for the AOP workshop at ECOOP'99 [Willink99c]. The version here resolves reverse and multiple inheritance conflicts. This version has been adapted to use the revised syntax of derivation rules as a part of an object-statement-clause.
class Monitor
{
    friend class Monitor::ReadOnlyLock;
    friend class Monitor::ReadWriteLock;
private:
    void acquire_exclusive() { /* ... */ }
    void acquire_shared() { /* ... */ }
    void release() { /* ... */ }
};

acquire_exclusive and acquire_shared block until exclusive or shared access is available to the resource(s) managed by the monitor. release terminates the resource reservation.

Reservation of the monitored resource is managed by a pair of nested lock classes, ReadOnlyLock and ReadWriteLock. They differ only in whether acquire_shared or acquire_exclusive is invoked.

class Monitor::ReadOnlyLock
{
private:
    Monitor& _monitor;
public:
    ReadOnlyLock(Monitor& aMonitor)
    : _monitor(aMonitor) { _monitor.acquire_shared(); }
    ~ReadOnlyLock() { _monitor.release(); }
};

A lock class invokes acquire_shared to acquire the resource during construction and ensures its release from the destructor, whose invocation C++ guarantees.

Aspect composition

The Stack application code above is written independently of the synchronisation code. The presence of the volatile keyword is an optional optimisation.

The monitor aspect is added to the application aspect by providing additional declarations that are woven into the application code.

    using "monitor.fog";
    template <class T>
    class Stack
    {
        $Monitor::install(); // Invoke meta-function
    };

8. The qualification of is_empty() as volatile as well as const therefore goes beyond conventional practice, but is safe because the implementation involves a single read, whereas top() involves at least two reads.

Qualification of is_empty() as volatile is of limited utility, since the return accurately reflects a state that existed but that state may no longer exist when the calling code interprets the result. It would seem that the volatile qualification is redundant since calling code must establish a lock to encompass both a !is_empty() and a subsequent pop(). However, the volatile qualification is useful when is_empty() is invoked within a polling loop that can recover on the next iteration. The presence of volatile avoids incurring locking costs for such a loop.

The usage is consistent because volatile indicates that concurrent change may occur and so inhibits any optimisation that could reorder the sequence of memory accesses.
The Monitor Aspect (compile-time)

The remainder of the code for this example forms part of the monitor.fog include file. There are two relatively independent code injections to be performed to install the monitor. Class declarations must be updated to incorporate an instance of Monitor, and function declarations must be updated to establish locks.

Direct installation of an instance of Monitor is relatively straightforward and could be achieved by just adding a member variable. However, making the meta-function work in a more general purpose fashion is harder. There are five problems to be resolved:

- Monitor::install may be invoked more than once on the same class
- Monitor::install may be invoked later for a derived class
- Monitor::install may be invoked later for a base class
- Monitor::install may be invoked later for more than one base class
- Monitor::install may be invoked for a derived monitor

There must be only one synchronisation monitor in each object, so multiples must be suppressed, retaining only the one monitor in the least derived class. If a monitor is multiply inherited, virtual inheritance must be used to share it. If multiple monitors use different implementation classes, we will generate a compiler diagnostic.

Since virtual inheritance must be used to resolve the multiple inheritance problem, it is convenient to implement the more conventional monitor by non-virtual multiple inheritance rather than as a member variable. Resolving multiple inheritance then just requires composing the virtual keyword on the simpler and much commoner inheritance.

Resolving the uniqueness problem when the invocation order of Monitor::install cannot be known requires splitting the structural problem into two passes. A third pass is required to update the functions. These three passes are performed in turn during the semantic analysis phase, meta-construction phase and meta-destruction phase.

Pass 1, Semantic analysis phase

The first pass is executed directly by the during semantic analysis of the $Monitor::install() meta-function invocation.

It sets flag variables indicating the class requirements, and arranges for the second phase to occur later.

```cpp
auto const class Monitor::needs_monitor = 0; //1
auto declaration Monitor::install() //2
{
    class $Scope : auto $Dynamic {}; //3
    auto const class needs_monitor = $Dynamic; //4
}
```

//1 declares a meta-variable in the Monitor class whose 0 (nil) initialisation flags that a Monitor class does not need Monitor functionality inserted into it.

//2 declares the compile-time meta-function invoked by the application glue code.

//3 adds the Monitor class (as a meta-base class of Stack).

```cpp
template <class T> class Stack : auto Monitor { ... };
```

$Scope is not a formal and so resolves to the prevailing scope in the invocation context.
Dynamic is a built-in formal that resolves to actual definition scope, typically Monitor, but DerivedFromMonitor if Monitor::install is invoked as DerivedFromMonitor::install.

```cpp
class DerivedFromMonitor : public Monitor
{
   // ...
};
//...
$DerivedFromMonitor::install();
```

Installation of Monitor as a meta-base-class provides an inherited meta-constructor and meta-destructor for Stack, and so ensures that the second and third passes are executed for Stack (and all its derived classes).

```cpp
//4 declares the Stack::needs_monitor meta-variable with a non-0 value to signal that Monitor functionality is required. The value of Dynamic is used as the non-0 value, so that in combination with the const, any attempt to install a different class of monitor will be caught. Re-installation of the same monitor class is allowed.
```

**Pass 2, Meta-construction phase**

The first pass sets the needs_monitor flag non-0 in all classes that are specified as requiring monitor functionality, and arranges for the meta-constructor to be invoked. Invocation in the first pass occurs in an unpredictable order. Invocation during the meta-construction phase occurs in a least derived first order, which can be exploited to install the monitor in the least derived alternative. Resolution of the multiple inheritance conflict requires a further pass, which is implemented by performing an iteration over the multiple bases.

```cpp
auto Monitor::Monitor()
{
   auto if (needs_monitor) //1
   {
      auto class baseMonitors[] = $find(has_monitor); //2
      auto if (baseMonitors.size() == 0) //3
      {
         class $Scope : public $needs_monitor {};
         auto const class has_monitor = $Scope;
      }
      else if (baseMonitors.size() == 1) //4
      {
         auto const class needs_monitor = $baseMonitors[0];
      }
      else //5
      {
         auto for (iterator m = $baseMonitors; m; ++m)
         {
            class *m : virtual $needs_monitor {};
            auto const class needs_monitor = *m;
         }
      }
   }
//1 Execution of the meta-construction code is guarded by a test for a non-0 flag, thereby inhibiting installation of monitor functionality in monitor classes.
//2 std::find returns an exposed list of all visible declarations of the has_monitor flag, which is used to identify the location(s) where monitor functionality is already installed.
//3 If there are no definitions visible, this must be a least-derived requirement and so the monitor class is specified as a public base class, composing with the existing
```
specification as a meta-base class. The has_monitor flag is defined to indicate the location of the monitor functionality.

//4 If there is exactly one definition visible, then the inherited functionality is adequate and no further functionality is required in this class. (Re-)declaration of the derived needs_monitor provokes an error message if conflicting monitor classes are in use.

//5 If there is more than one definition visible, then a multiple inheritance conflict must be resolved. The iteration loops over all definitions and redefines the base-class to use virtual inheritance, and detects conflicting monitor classing.

Pass 3, Meta-destruction phase

The final phase of monitor installation should occur after any concurrent meta-programming has defined additional member functions, so that all member functions may have locking code inserted.

```cpp
auto Monitor::~Monitor()
{
    auto if (needs_monitor) // 1
    {
        auto for (iterator f = $functions(); f; ++f) // 2
        {
            auto if (f->is_static()) // 3
            ;
            else if (f->is_volatile()) // 4
            ;
            else if (f->is_const()) // 5
            {
                $f->specifier() :
                {
                    entry { ReadOnlyLock aLock(*this); };
                }
            }
            else // 6
            {
                $f->specifier() :
                {
                    entry { ReadWriteLock aLock(*this); };
                }
            }
        }
        auto if (friends().size() != 0) // 7
        $std::error("friend of monitored " $Scope " detected.");
    }
}
```

//1 Once again functionality is guarded to prevent operation on monitor classes.

//2 The loop over all functions uses the decl-specifiers to determine whether monitor code needs inserting.

//3 static member functions are not associated with any object and so have nothing to monitor access to.

//4 volatile is recognised as a requirement to bypass locking.

//5 const member functions require a shared lock, which is provided by specifying an entry code segment for the function whose full name is returned by function::signature().

//6 Similarly non-const member functions require an exclusive lock.

//7 Finally the problem of friend functions and classes subverting the protection is resolved in a very heavy handed fashion by banning friends. (Direct access by friends must be changed to use access functions into which locks can be inserted.
automatically. It very hard and probably impossible to analyze all code associated with a friend class or function to guarantee that it does not violate access constraints.)

Function Weaving

FOG performs function weaving by concatenating the code from multiple function bodies, within the five named regions entry, pre, body, post and exit (see Section 4.4.8). The entry region precedes the default body region and so the above meta-program generates the additional contribution

```cpp
template <class T>
T Stack::top() const // from the meta-destructor
{
    entry { ReadOnlyLock aLock(_monitor); }
}
```

to be woven with the application function:

```cpp
template <class T>
T Stack::top() const // from application aspect
{
    return _elements[_tally-1];
}
```

to generate the final C++ result:

```cpp
template <class T>
T Stack::top() const
{
    #line ...
    ReadOnlyLock aLock(_monitor);
    #line ...
    return _elements[_tally-1];
}
```

References

Monitor and Stack are fundamental concepts and consequently staples for numerous articles in many Computer Science fields. [Stroud95] used OpenC++ version 1 to implement atomic data types by intercepting method calls at run-time. [Hedin97b] considered the monitor from an Aspect Oriented perspective, and introduced an attribute extension language to enable a preprocessing stage to validate that the requisite coding constraints had been observed. In this example we use introspection to synthesise the required code directly. [Bjarnason97] advocates an extensible language, so that the required monitor protocol can be incorporated into the extended language. Language extension involves manipulation of syntax trees, and it is not clear how practical this is for a language with as challenging a syntax as C++.

7.5 A Real Example - BURG

FOG grew out of work to improve productivity in a different field of compilation technology.

It is difficult to apply high level programming concepts to Digital Signal Processors because of the very poor quality of the available compilers [Willink97b]. This is in part due to lack of awareness of the need for better support and partly due to the extreme difficulty of matching the performance of hand-crafted assembler on rather challenging architectures. Research therefore started to apply modern Very Long Instruction Word (VLIW) scheduling concepts, using an intermediate representation supporting data parallelism [Munchnick93]. The intermediate
representation was extended to support user-characterised types [Willink97a], as part of a relatively general purpose compiler framework.

One of the activities of a compiler involves selection of appropriate machine instructions (such as ADD or MOVE) to implement the program, usually represented by a tree of Abstract Syntax Tree nodes [Aho86]. An effective approach to solving this problem involves a Bottom-Up Rewrite System [Proebsting95], which searches the tree from the leaves upwards identifying the lowest cost solution that has each node covered exactly once by a machine instruction. The tree may then be rewritten in terms of the selected machine instructions. In order to support multiple target architectures, alternative instruction sets must be covered. Implementation of this diversity is assisted by the use of a Bottom-Up Rewrite Generator to transform a description of each machine instruction into the form needed for an efficient tree search. An example of this form of generator is *iburg* that forms part of the *icc* C compiler [Fraser95].

*iburg* is a compact C program comprising just three files:

- *iburg.c* has 690 lines and 4652 tokens
- *iburg.h* has 66 lines and 259 tokens
- *gram.y* is a 19 rule, 37 state *yacc* parser grammar

(token counts are non-comment, non-whitespace preprocessor tokens.)

*iburg* supports single dispatch architectures (such as SPARC). An enhanced version was required in order to support less conventional processor architectures, and so a highly Object Oriented C++ rewrite was undertaken using reference counting and smart pointers to share common partial instructions. The resulting program was substantially larger, due to the extra declarations for encapsulated C++ classes, rather than the original free access to structure elements, and due to the added functionality. Preprocessor macros were used extensively to factor out common declarations.

A further revision to exploit FOG without any other change to functionality forms the basis of the comparison for this example. An implementation based on the use of preprocessor macros is compared with an implementation using meta-functions, meta-variables and derivation rules.

The benefit of using FOG for the 10 non-*yacc* modules are presented in Table 7.1.

<table>
<thead>
<tr>
<th>module</th>
<th>.cxx</th>
<th>.fog</th>
<th>% reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Burg</td>
<td>1420</td>
<td>8301</td>
<td>7681</td>
</tr>
<tr>
<td>BurgCodeScope</td>
<td>496</td>
<td>2901</td>
<td>2496</td>
</tr>
<tr>
<td>BurgEntry</td>
<td>98</td>
<td>383</td>
<td>227</td>
</tr>
<tr>
<td>BurgNonTerm</td>
<td>204</td>
<td>849</td>
<td>679</td>
</tr>
<tr>
<td>BurgParserValue</td>
<td>78</td>
<td>276</td>
<td>248</td>
</tr>
<tr>
<td>BurgRule</td>
<td>361</td>
<td>1903</td>
<td>1583</td>
</tr>
<tr>
<td>BurgSharedRules</td>
<td>86</td>
<td>327</td>
<td>171</td>
</tr>
<tr>
<td>BurgSubExpr</td>
<td>428</td>
<td>2545</td>
<td>2062</td>
</tr>
<tr>
<td>BurgTerm</td>
<td>163</td>
<td>651</td>
<td>483</td>
</tr>
<tr>
<td>BurgTree</td>
<td>366</td>
<td>2111</td>
<td>1821</td>
</tr>
<tr>
<td>total</td>
<td>3720</td>
<td>20247</td>
<td>17451</td>
</tr>
</tbody>
</table>

Table 7.1 Token size reduction through use of FOG

The pre-FOG version comprises sources that compact .cxx and .hxx into a
single .xx file. Raw (comment and blank included) line counts are presented for these. The post-FOG version comprises a single .fog file per module. (The counts exclude re-usable preprocess/meta-function definitions).

Use of (the multi-pass implementation of) FOG reduced the token count by 14%, from 20250 to 17500. The per-module reduction varied between 9% and 48%. The larger reductions occur in small classes, where the benefits of derivation rules and simplification of interface and implementation declarations are most apparent.

A reduction in token count is an easily measured reduction in programming effort. Less easily measured are the more aesthetic improvements of better modularity, improved expression of programming intent, and automatic compliance with programming protocols. A pair of before/after extracts are therefore provided for readers to make their own judgements. The code is complete save for the removal of 4 functions whose lexical structure exactly duplicates functions that remain. Code for this example is chosen because it is shortest, and so demonstrates the changes more clearly. Providing the large number of unaffected function body lines from a more typical module would not provide extra insight. The definitions of the preprocessor macros or meta-functions is not shown. The two are of comparable lexical size, the meta-function has a higher token count through the use of $ operators and meta-type names, but a lower token count through the use of more appropriate facilities. The meta-functions are modular through having fewer interdependencies than the preprocessor macros, and more readable through the use of more conventional structuring and the elimination of back-slash continuation lines.

The original preprocessor macros are almost completely eliminated. The CUSTOM_RTTI support is provided automatically by derivation. The remaining 6 macros supporting smart pointers are all subsumed by MapOfSmartPointerSpecialisations. Other meta-functions such as Mutate just implement simple idioms.

Original interface file:

```c++
#ifndef ENTRY_HXX
#define ENTRY_HXX
#include <Burg.h>
#include <Id.hxx>  // A smart string class
#include <Object.hxx>
#include <ReferenceCount.hxx>
#include <SmartPointer.H>

class Entry : public Object
{
    CUSTOM_RTTI_WITH_1_BASE_DECLARATION(Entry, Object)
    REFERENCE_COUNT_DECLARATION(Entry)
    NULL_OBJECT_DECLARATION(Entry)
private:
    const Burg& _burg;
    const IdHandle _id;  // Handle for a smart string
private:
    Entry(const Entry&);  // No copy
    Entry& operator=(const Entry&);  // No assign
protected:
    Entry();
    Entry(Burg& aBurg, const Id& anId);
```

29-June-2001
public:
    const Burg& burg() const { return _burg; }
    const Id& id() const { return *_id; }
    virtual Term *is_term();
    const Term *is_term() const
        { return ((Entry *)this)->is_term(); }
    virtual void mark_reachable();
    virtual ostream& print_this(ostream& s) const;
};
#endif

Original implementation file

#include <Entry.hxx>
#include <Burg.hxx>
#include <MapOfSmartPointer.H>

CUSTOM_RTTI_WITH_1_BASE_IMPLEMENTATION(Entry, Object)
REFERENCE_COUNT_IMPLEMENTATION(Entry)
NULL_OBJECT_IMPLEMENTATION(Entry)
SMART_POINTER_IMPLEMENTATION(Entry)
MAP_OF_SMART_POINTER_IMPLEMENTATION(Entry)

Entry::Entry()
    : _burg(Burg::null_object()) {}
Entry::Entry(Burg& aBurg, const Id& anId)
    : _burg(aBurg), _id(anId) { aBurg.add_entry(*this); }
Term *Entry::is_term() { return 0; }
void Entry::mark_reachable() {}
ostream& Entry::print_this(ostream& s) const { return s << _id; }

Revised FOG code, with use of FOG extensions italicised

using "Burg.fog"; // Improved form of #include.

class Entry : public Object
{
    using/interface "Burg.h"; // Need a #include <Burg.h>
    $NoCopy(); // Section 7.1.2
    $NoAssign(); // Section 7.1.2
    $Mutate(); // Section 7.1.3
private:
    const Burg& _burg = Burg::null_object();
    const IdHandle _id;
protected:
    !inline Entry() {} // Uses default initialiser value
public:
    const Burg& burg() const { return _burg; }
    const Id& id() const { return *_id; }
    virtual Term *is_term() { return 0; }
    const Term *is_term() const { return mutate().is_term(); }
    virtual void mark_reachable() {}
    virtual ostream& print_this(ostream& s) const
        { return s << _id; }
};
$MapOfSmartPointerSpecialisations(Entry);

protected Entry::Entry(Burg& aBurg, const Id& anId)
    : _burg(aBurg), _id(anId) { aBurg.add_entry(*this); }
7.6 Summary

We have shown a number of examples that steadily progress from apparently trivial one-liners through usage of extended declarations and on to meta-programming. We have concluded with an example that begins to show FOG in use for a real application. The examples show that FOG can capture repeated practice well and so avoid redundant source text and the consequent maintenance risks. The final example highlights the modest proportion of real code that is repetitive. FOG apparently does little to reduce the programming burden of straight application code.

Unfortunately more extensive usage remains an area for further work. Many more lines of code must be adapted to exploit FOG to determine how beneficial FOG is. More programmers must use FOG to determine how easy FOG is to use and learn. More usage is required to stress the enhanced syntax and identify areas in need of revision. Extensive usage is needed to build up appropriate standard coding styles and meta-library support. In the same way that C++ provides many programming opportunities that were not available in C, FOG provides opportunities not available in C++. Perhaps one of these opportunities may identify a way of creating more compact abstractions for the straight application code.
8 Summary

The many achievements of FOG will now be reviewed before highlighting what remains to be done, and the limitations upon what can sensibly be done. The relevance to other languages will then be discussed before finally concluding with a brief summary of how the problems with C++ described in the introduction have been resolved.

Novelty

There are few ideas in software engineering that are totally new. Most are the result of a revision or combination of prior work. The functionality of FOG combines concepts from many areas, adapting them to fit the philosophical, semantic and syntactic constraints of the C++ language. The combination is certainly novel.

8.1 Parsing

8.1.1 Context-free syntactical C++ parsing

Processing C++ declarations before their semantics have been determined necessitates context-free parsing. A clear distinction between syntax and semantics is not normally made because C++ is perceived to be inherently context-dependent requiring lexical, syntactic and semantic analyses to be tightly coupled and consequently blurring the distinctions between these concepts. The official "(Informative)" grammar provides a mixture of lexical, syntactic and a few semantic rules. The main body of the standard does not always distinguish whether described constraints are syntactic or semantic. Tradition therefore perpetuates the perception that parsing must be difficult.

FOG draws a pragmatic distinction between syntax and semantics. Syntax is what can be analysed by an LALR(1) parser such as yacc. Semantics is what has to be analysed later.

Examination of the C++ grammar shows that syntactic analysis without type information causes only a minor ambiguity for expressions using casts. The ambiguity is entirely deterministic and readily deferred for resolution at the post-yacc semantic level.

Accurate syntactic analysis without template information is impossible. However, an iteration through all alternatives of a template < / arithmetic < ambiguity can be performed to determine a consistent, but not necessarily correct, syntactic analysis. Instrumentation of practical programs shows that approximately 1% of statements contain a template ambiguity, and that for approximately 1% of those statements, the consistent parse is incorrect. A syntactically consistent parse is therefore possible without template information, subject to the requirement that the semantic processing must repair the incorrect parse for approximately 0.01% of statements.

8.1.2 Back-tracking in yacc

LALR parsers such as yacc have no overt support for back-tracking unlike their LL counterparts. Ambiguity problems that cannot be resolved within the grammar need assistance from a separate lookahead parser. Implementation of back-tracking within yacc, using the error token to rewind, proves to be fairly straightforward enabling lookahead parsing to be performed within yacc. The ambiguity between arithmetic and template interpretation of an < exploits this back-tracking technique to perform a binary search to identify a syntactically consistent parse.
8.1.3 Superset grammar

Traditional C++ grammar implementations attempt to maximise the semantic resolution of the yacc grammar, since this minimises subsequent coding. This is also motivated by the need to incorporate a large amount of semantic intelligence to resolve recursive declaration/expression ambiguities accurately.

Context-free parsing makes resolution of declaration/expression ambiguities impossible. The ambiguity must be deferred for semantic resolution rather than attempt to resolve it syntactically.

The superset FOG grammar recognises that the declaration/expression ambiguity derives from the fundamental C language design: declarations should mimic their usage in expressions. The ambiguity is therefore not a series of inconvenient barriers to be surmounted, but rather a series of partially overlapping sub-syntaxes. Generalisation of declaration and expression and a few other productions are used to perform context-free parsing in the superset grammar.

8.1.4 Semantic analysis restricted to semantics

Removal of semantic considerations from the syntactic analysis considerably simplifies the grammar, but requires additional semantic processing. However, this processing is making the same decisions as before, but in the controlled context of an AST rather than the difficult partial environment during parsing.

Appendix F.3.1.1 describes how the resolution of the declaration/expression and related conflicts at the semantic level involves a straightforward but not quite trivial dataflow algorithm propagating a bit-mask of satisfied semantic hypotheses from the leaves of the AST to the root, where any residual ambiguity can be resolved by applying the defined ambiguity resolution rules. The propagation makes use of type information, tree structure and associated semantic constraints in determining whether a hypothesis such as is-parameter-declaration is satisfied.

Deferring the ambiguity aids error diagnosis as well. An error during syntactic processing indicates that the syntactic analysis has failed to understand the token sequence and as a result an error recovery mechanism must be invoked to resynchronize. Since analysis failed, it is often difficult to make a better error diagnosis than "syntax error near line x". The more general syntax to defer the ambiguity accepts many sentences corresponding to simple typographical programming errors with the result that the syntactical analysis does not lose synchronisation, and so a more appropriate error diagnostic can be produced.

8.1.5 Extended regular expressions

Demonstrating that the superset grammar covered the existing syntax required analysis of the C++ grammar. An extended form of regular expression was introduced to describe sentences of C++ enabling the traditional ambiguities to be deduced and the superset justified.

8.2 C++ Extensions

8.2.1 Meta-programming

Meta-programming has been introduced, and as a result the C preprocessor rendered redundant, through the use of features that integrate with, rather than conflict with, the language.

Meta-variables and meta-functions supplant object-like and function-like macros, and benefit from the consistent availability of argument and return types and definition within a class hierarchy.

Meta-statements replace conditionaiisation, supporting loops as well.
An invited substitution mechanism avoids the hazards of imposed substitution with the preprocessor and provides a simple solution to the problem of lexical concatenation.

8.2.2 Composition rather than One Definition Rule

The C++ One Definition Rule requires complete declarations to occur in a single place. This prevents code being organised by algorithm rather than by data. FOG eliminates this restriction so that multiple declarations are combined to give a composite meaning. This supports weaving of declarations together for Aspect Oriented Programming, or elaboration of declarations by meta-programming.

8.2.3 Minor extensions

Some minor enhancements to C++ are introduced to provide greater consistency when declarations are composed.

8.2.4 Derivation rules

Derivation rules are perhaps just a little bit of syntactic sugar to simplify meta-programming. However many realistic problems involve a policy that has to be observed by classes within an inheritance hierarchy. This requirement is captured directly by derivation rules. Related work on automatic generation of code appears to concentrate more on resolution of composition conflicts.

8.2.5 Syntax macros

The illusion of a language extension can be created by a syntax macro, so that users may introduce new keywords such as synchronised or persistent.

8.3 Detailed Language Issues

8.3.1 Scoped preprocessing

Macros and preprocessing are a neglected, perhaps scorned, field in software engineering. Little work has been done and no work that considers macros within the hierarchical context of C++. Resolution of macro-names within a prevailing scope, with the consequent benefits that can accrue from isomorphism and inheritance is new in FOG.

8.3.2 Deferred substitution

Resolution of names within the correct name-space at the correct-time is a traditional concern of language designers and consequently programmers. The functional argument (FUNARG) problem in Lisp demonstrates the problems of avoiding name capture. The distinction between \ and & substitution operators in VAX/VMS DCL show the need to control resolution time. FOG applies related concepts to substitution within meta-programs through the $ and © operators.

8.3.3 Polymorphic syntax

Syntax macros are traditionally syntax-driven: the known syntactical requirements of the macro (a meta-function or meta-variable in FOG) are used to guide the syntactic analysis. This introduces two semantic context-dependencies to the syntactic analysis.

Exploiting the known syntactical requirements may require semantic analysis to determine what is required by the particular usage. The superset grammar unifies many C++ constructs and the approach is extended to define the tree-statement production that encompasses almost the entire C++ grammar, enabling a syntax-independent and consequently context-free parse of meta-function arguments and
Meta-compilation for C++

Summary

Meta-variable initializers. Syntactic analysis of each meta-function and meta-variable usage is therefore context-free in FOG.

Defining the macro in the first place requires semantic analysis of the definition, albeit a degenerate semantic analysis such as the extra-lingual `#define` for the C preprocessor. FOG also uses a syntax for syntax macro definitions for which a premature semantic analysis can be activated during syntactical analysis.

Syntactic analysis of the use of a syntax macro requires dynamic changes to the table of reserved words and a data-dependency upon the number but not type of parameters.

This is only novel within the context of C++. In a language with a clean syntax a generic parse should be trivial, but still worth implementing to remove context-dependency.

8.3.4 Literal source

Meta-code surrounds ordinary statements and declarations so that there is no need for special syntax or procedures to define source syntax literals.

The source syntax is its own literal (overlined).

```cpp
auto statement switchBody = { default: return 0; };
auto switchBody += case 1: { flags++; return 1; }
```

There is no need for any insight into the structure of the internal ASTs or their support functions.

The entire function body is returned:

```cpp
auto declaration declare_pointer_classes(identifier aClass) {
    class $aClass;
    typedef PointerTo<$aClass> ${aClass}Pointer;
    typedef PointerTo<const $aClass> ${aClass}ConstPointer;
}
```

8.3.5 Potential and Actual

Meta-programming is traditionally practised in Smalltalk and Lisp-like languages, where meta-programming occurs at run-time re-using functionality necessary to establish an Object Oriented execution environment. More recently meta-programming has been possible in Java at run-time and rather more interestingly and uniquely at load-time.

Compile-time (or static) meta-programming is not widely used since it is only available in research languages such as OpenC++ or MPC++. These languages support programmed manipulation of the declaration pool.

Existing approaches therefore deal with actual declarations. FOG with its syntactic support for source literals introduces the distinction between potential declarations and actual declarations, allowing meta-programs to operate consistently on declarations with determined or undetermined scopes.

8.4 Further Work

The first version of FOG currently available on the net used the multi-pass grammar approach. The more stream-lined and efficient approach supported by the superset grammar is also available, but requires considerable further development.

FOG is currently written in C++ using a trivial custom preprocessor that just splits interface and implementation from a single file and performs code synthesis only for include files and their guards. The source code for FOG should be revised to exploit FOG functionality, and thereby demonstrate and test the use of FOG more
convincingly. A measurement of the lexical source size reduction should show how beneficial FOG is for large programs with deep inheritance hierarchies.

C++ is a large language, which FOG should at a minimum parse and emit unchanged. FOG has a useful degree of functionality in many areas, however practical experience with interesting small examples tends to encounter unimplemented or misimplemented functionality with respect to the current state of the implementation.

Meta-library

Use in a diverse range of applications needs to be assessed and a meta-library of common utilities developed.

Join discipline

The original multi-pass implementation did not analyze function bodies, and so function body composition was performed by lexical concatenation without regard to even syntactical validity. The full super-set parse, combined with the use of token lists to maintain ASTs, can ensure semantic validity at the language level. However the more challenging issue of establishing or enforcing programming practices that ensure integrity of programming intent remains to be addressed.

Use of self-evident semantics at syntactical level

Section 4.2.3 identified the need for a possible variant of the $ trigger to allow the known semantic type of the argument to be exploited.

Composition of exceptions

Section 4.4.8 identified a possible policy for composition of exception specifications and function-try-blocks.

Meta-programming phases

Section 4.6 identified the inadequacy of the meta-construction, meta-main and meta-destruction compilation stages.

Syntax macros

Section 4.7 described a partially implemented proposal for syntax macros, and identified severe limitations for the case of multi-argument syntax macros.

Expression AST traversal

The built-in meta-functions described in this thesis support meta-programming of declarations. Further meta-functions could be added to support meta-programming of expressions thereby providing the ability to peek and poke in arbitrary fashion just like OpenC++. Further research is needed to determine whether it is merely necessary to support arbitrary user access, or whether a more disciplined form of support can be identified.

8.5 Limitations

FOG operates as a translator to C++ and so necessarily precedes C++ compilation. FOG cannot operate on actual compilation results, only upon predictions of those results. This has two consequences.

It is not guaranteed that FOG sees the final state of declarations. In a multi-session compilation, a class may appear to be a leaf class in one session, but further derived classes may exist in other sessions. Decisions predicated on leaf-
ness will therefore be in error. A complete fix of this problem requires global knowledge. Detection of the anomaly can be resolved as described in Section 6.5.8 through the use of a checksum to express the non-global knowledge of a meta-compilation session.

Template instantiation occurs during or after compilation. It is therefore difficult for FOG to know which parameter combinations will be used, or to detect which member functions will actually be required. In the general case where FOG is used to prepare library code, FOG cannot know what the instantiations will be. It is therefore impossible for meta-programs to manipulate template instantiations usefully. Meta-programming in FOG is limited to manipulation of template declarations.

Since templates create contexts in which FOG cannot know what type is in use, and so restricts the amount of meta-programming that can be reliably performed. This problem can only be resolved by integration of meta-compilation with normal compilation, so that meta-compilation is performed on instantiated as well as declared templates.

Syntax macros provide a limited mechanism for introducing language extensions, however it is difficult to do better within the confines of the poorly structured C++ syntax.

Most programmers have an, at least initial, dislike of the compact and idiomatic style of the C and C++ syntax. Experienced C and C++ programmers come to like it. FOG adds further extensions in the style of C++ and so provides more to confuse or dislike. It remains to be seen whether real programmers learn to find the extensions acceptable and natural.

8.6 Other Languages

Although the work described in this thesis is primarily concerned with resolving deficiencies in the use of C++, the work is of greater applicability, mainly to languages that involve significant compilation activity such as Eiffel, Ada or Java. Introduction of extra compilation stages is inappropriate for languages such as Smalltalk or CLOS where object structure is defined at run-time.

The distinction between potential and actual declarations, the concept of derivation rules and a flexible substitution based upon tree-literals combined with a lexical concatenation are not specific to C++, although some of the detailed syntactical issues are. Implementation of these concepts in other languages is likely to be a little simpler, since few other modern languages have quite such a challenging syntax as C++.

The observation that the One Definition Rule is a major hurdle to implementation of patterns and Aspect Oriented Programming is again applicable to all languages. Language designers should endeavour to support interleaved declarations.

Programming involves repetition at many levels, and programmers naturally seek to factor the repetition into some parameterisable reusable construct, which may be a loop, subroutine, class, template, macro, file or library. Omission of any of these capabilities simplifies a language, but limits the programmer's or the program's efficiency. Some form of macro to perform lexical processing and meta-programming is therefore beneficial to all languages, although the precise syntax must be carefully chosen to fit within the traditional style of each language.

8.7 Resolution of Goals

The introductory discussion highlighted problems that arise with C++. The way in which these are resolved in FOG will be summarised.

Cpp should be replaced rather than eliminated.
Object-like and function-like macros are replaced by meta-variables and meta-functions.

Substitution by imposition is replaced by a substitution invited by $ or @.

## and # are replaced by adjacent lexical element concatenation.

Conditional processing is replaced by meta-programming.

*Compile-time programming is necessary to configure declarations.*

*Introspection is useful for simple applications.*

*Reflection is almost essential for sophisticated applications.*

*Meta-programs can manipulate declarations.*

*Patterns and AOP require weaving.*

*The One Definition Rule must be circumvented.*

*Interleaved declarations should be allowed.*

The One Definition Rule has been relaxed to allow declarations to be introduced outside the confines of class braces. This supports interleaved declarations and weaving within classes. Multiple contributions to the same declaration are composed, supporting weaving of individual declarations.

*Lexical redundancy should be eliminated.*

The need for distinct interface and implementations has been removed.

Derived code can reuse inherited declarations.

*Predictable code should be provided automatically.*

Derivation rules support automatic generation of derived code.

Meta-programs can generate code for more specialised applications.

*A concept should be instantiated by a single invocation.*

Invocation of a meta-function can provide complete instantiation, exploiting composition to inject code as appropriate.
9  Glossary

9.1  Acronyms

AI            Artificial Intelligence
ANSI           American National Standards Institute
AO(P)          Aspect-Oriented (Programming)
ARM           Annotated Reference Manual [Ellis90]
AST           Abstract Syntax Tree
BNF           Backus-Naur Form
BURG           Bottom-Up Rewrite Generator
CAD           Computer Aided Design
CFG           Context-Free Grammar
Cpp            C preprocessor
CRC           Cyclic Redundancy Check
DFA           Deterministic Finite Automaton
DSP           Digital Signal Processor/Processing
FFT           Fast Fourier Transform
FOG           Flexible Object Generator (in this thesis)
              Fragmented Object Generator (in [Gourhant90])
GNU           GNU is Not Unix
GoF           Gang of Four book [Gamma95]
GP            Generative Programming
GUI           Graphical User Interface
LALR(k)       Look-Ahead parsing based on Left-to-right scanning of the input, with
              Right-most derivation in reverse, using k input symbols of lookahead.
LL(k)         Parsing based on Left-to-right scanning of the input, with Left-most
              derivation, using k input symbols of lookahead
LR(k)         Parsing based on Left-to-right scanning of the input, with Right-most
              derivation in reverse, using k input symbols of lookahead
MOP           MetaObject Protocol
NFA           Non-deterministic Finite Automaton
ODR           One Definition Rule (§3.2)
OO(P)         Object-Oriented (Programming)
RTTI          Run-Time Type Information
SO(P)         Subject-Oriented (Programming)
UML           Unified Modeling Language
VLIW           Very Long Instruction Word
yacc            yet another compiler compiler
### 9.2 Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>grammar</td>
<td>The composite syntactical definition of a language, comprising many (production) rules and one distinguished non-terminal.</td>
</tr>
<tr>
<td>isomorphic</td>
<td>Having the same shape. A set of classes that exhibit compatible interfaces, usually for the purposes of satisfying the requirements of a template parameter, are isomorphic. Isomorphic classes need not share a common base class.</td>
</tr>
<tr>
<td>lexeme</td>
<td>Synonym for terminal or token.</td>
</tr>
<tr>
<td>lexical analysis</td>
<td>Analysis determining a lexeme or token from a source character sequence.</td>
</tr>
<tr>
<td>meta-base meta-class</td>
<td>Meta-class from which another meta-class inherits.</td>
</tr>
<tr>
<td>meta-class</td>
<td>Class that describes a class.</td>
</tr>
<tr>
<td>namespace</td>
<td>The specific form of name-space established by a C++ namespace.</td>
</tr>
<tr>
<td>name-space</td>
<td>Any context in which names may be resolved.</td>
</tr>
<tr>
<td>non-terminal (token)</td>
<td>A term in a production rule defined by the left-hand side of one (or more) production rules.</td>
</tr>
<tr>
<td>polymorphic</td>
<td>Having many shapes. A class hierarchy should specialise a common base class with respect to which the classes exhibit polymorphism.</td>
</tr>
<tr>
<td>production (rule)</td>
<td>Rule describing the grammatical equivalence of a left-hand side non-terminal with a sequence of right-hand side terminals and non-terminals. Multiple rules sharing a common left-hand side are often loosely referred to as a single production.</td>
</tr>
<tr>
<td>reduction (rule)</td>
<td>Synonym for production (rule).</td>
</tr>
<tr>
<td>reflect(ion)</td>
<td>Inspection and modification of a program by itself.</td>
</tr>
<tr>
<td>root class</td>
<td>The least derived class in an inheritance hierarchy.</td>
</tr>
<tr>
<td>root scope</td>
<td>The least derived scope associated with a derivation rule.</td>
</tr>
<tr>
<td>rule</td>
<td>See production (rule).</td>
</tr>
<tr>
<td>semantic analysis</td>
<td>Analysis determining whether a (syntactically valid) sentence satisfies semantic constraints.</td>
</tr>
<tr>
<td>sentence</td>
<td>A sequence of source tokens generally satisfying some syntax.</td>
</tr>
<tr>
<td>syntactical analysis</td>
<td>Analysis determining whether and in what way a source sentence satisfies a grammar.</td>
</tr>
<tr>
<td>syntax</td>
<td>A specific subset of a grammar.</td>
</tr>
<tr>
<td>terminal (token)</td>
<td>An element in a production rule directly corresponding to a product of lexical analysis.</td>
</tr>
<tr>
<td>token</td>
<td>See terminal.</td>
</tr>
</tbody>
</table>
10 References


Meta-compilation for C++

References


References

Meta-compilation for C++


http://www.softlab.is.tsukuba.ac.jp/~chiba/opsia98/proc/opencri.pdf


http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-ecoop93.ps.gz


http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-opsia95.ps.gz


http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-isotas96.ps.gz


http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-ecoop98.ps.gz


http://www.softlab.is.tsukuba.ac.jp/~chiba/opsia98/proc/chiba.pdf


http://tree.ca.oswego.edu/pub/papers/A++/sooppa.ps


The 1.21-8 version used for developing FOG is no longer available on the web, but is part of the FOG distribution in

http://www.ee.surrey.ac.uk/Research/CSR/GFOG

[Coëtmeur93b] Alain Coëtmeur. *flex++ 2.3.8-5 based on flex 2.3.8.* 1993.

The 2.3.8-7 version used for developing FOG is no longer available on the web, but is part of the FOG distribution in `http://www.ee.surrey.ac.uk/Research/CSRG/FOG`.


References


http://bilbo.idc.hk-r.se/~bosch/lsdfora/lorenz.ps


http://bilbo.idc.hk-r.se/9080/~bosch/ltdfora/hedin.ps


<table>
<thead>
<tr>
<th>References</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>References</td>
<td>Meta-compilation for C++</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------</td>
</tr>
</tbody>
</table>
| [Matsuoka93]| Satoshi Matsuoka and Akinori Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*. In Research

ftp://ftp.is.s.u-tokyo.ac.jp/pub/member/jef/tech/coo95_coda.ps.Z

http://www.srd.nl/people/fitj/ps/pattern-tool.ps.Z (working draft)


http://www.cantrip.org/traits.html

http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/oliva.pdf


on Object-Oriented Programming (ECOOP'98), LNCS 1543, Springer-Verlag, June 1998.
http://wwwwtrese.cs.utwente.nl/aop-ecoop98/papers/00sherr.pdf


Meta-compilation for C++


http://www.cs.indiana.edu/~hyland/sobel/rop.ps.gz


[Soukup95] Jiri Soukup. *Implementing Patterns*. In [Coplien95b].

ftp://prep.ai.mit.edu/pub/anu/acc/acc-2.8.0.tar.gz


http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/mich.pdf

http://extreme.indiana.edu/~tveldhui/papers/cppworld.ps

http://extreme.indiana.edu/~tveldhui/papers/meta-art.ps


References

Meta-compilation for C++


29-June-2001
http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/hroc/wu.pdf

FOG Grammar changes

A FOG Grammar changes

This summary of the FOG syntax follows the example of Appendix A of the [C++96] standard. It is intended to be an aid to comprehension, identifying all changes in a way that is easily compared to C++. Completely unchanged grammar productions are omitted, changed rules and terms are indicated by a strike-through for removal and underline for addition.

A.1 Keywords

typedef-name: identifier

namespace-name:
  original-namespace-name
  namespace-alias

original-namespace-name: identifier

namespace-alias: identifier
class name: identifier
template-id

deer-name: identifier
template-name: identifier

punctuation: one of
  { } [ ] ( ) ; : ? :: . * += -= *= /= %= &= | ~= != <= >=

reserved-words are not identifiers.

reserved-word: one of
  and and_eq asm auto break
  bitand bitor bool break
  case catch char class
  compl const const_cast continue
default delete do double

dynamic_cast else enum explicit
export extern false float

for friend goto if

inline int long mutable

namespace new not not_eq

operator or or_eq private

protected public register reinterpret_cast

return short signed sizeof

static static_cast struct switch

template this throw true

typecast typedef typeid typename

try typedef typeid typename

volatile wchar_t while xor

xor_eq

non-reserved-words are identifiers.

non-reserved-word: one of
  derived emit entry exit
  file frozen guard implementation
  include interface noguard
  path pool post pre
  prefix pure suffix utility

meta-type-names are reserved-words or identifiers.

29-June-2001
A.2 Lexical conventions

A.2.1 Phase 6 Concatenation Grammar

1. Some meta-type names are also reserved words. The usage as a meta-type name augments usage as a reserved word.
identifier<cat>
  identifier<pp>
  identifier<cat-text-literal><pp>

number-literal<cat>
  number-literal<pp>
  number-literal<cat-text-literal><pp>

string-literal<cat>
  string-literal<pp>
  string-literal<cat-text-literal><pp>
  string-literal<cat-white-space><opt-string-literal><pp>

 tree-literal<cat>
  tree-literal<pp>
  tree-literal<cat-text-literal><pp>

'anything-else'<cat>
  'anything-else'<pp>

 tree-literal<pp>
  at-literal
  dollar-literal
  syntax-macro-literal
  // See Section 4.7

at-literal:
  @ tree-expression.
  @ { tree-expression }

 dollar-literal:
  $ tree-expression.
  $ { tree-expression }
  $ dollar-literal

A.2.2 Phase 7 Tokenization Grammar

'discard':
  whitespace

'reserved-word':
  identifier<cat>

'punctuation':
  'punctuation'<pp>

character-literal:
  character-literal<cat>

floating-literal:
  number-literal<cat>

integer-literal:
  number-literal<cat>

string-literal:
  string-literal<cat>

meta-type-name:
  identifier<cat>

'non-reserved-word':
  identifier<cat>

identifier:
  other-identifier:
    identifier<cat>

tree-literal:
  tree-literal<cat>

// Token is discarded

// If identifier<cat> is a reserved word

// If number-literal<cat> is floating point

// If number-literal<cat> is fixed point

// If identifier<cat> is a meta-type name

// (and not a reserved word)

// If identifier<cat> is a non-reserved word

// If identifier<cat> is anything else
A.4 Expressions

identifier:
  other-identifier
  meta-type-name
  non-reserved-word
  tree-literal

id:
  identifier
  identifier < template-argument-list >
  template identifier < template-argument-list >

nested-id:
  id
  id : nested-id

scoped-id:
  : opt nested-id

special-function-id:
  ~ id
  conversion-function-id
  operator-function-id

nested-special-function-id:
  special-function-id
  id : nested-function-special-id

scoped-special-function-id:
  : opt nested-special-function-id

primary-expression:
  literal
  this
  + identifier
  + operator-function-id
  ++ qualified-id
  ( expression )
  id-expression
deaclator-id

A.5 Statements

statement:
  control-statement
  expression-statement
  compound-statement
  declaration-statement
  try-block
  auto control-statement
  auto meta-expression-statement

control-statement:
  labeled-statement
  selection-statement
  iteration-statement
  jump-statement

2. Resolution of the identifier < context-dependency is discussed in Section 5.8.2.
3. New non-terminal but no changed functionality
A.6 Declarations

declaration:
  block-declaration
  function-definition
  template-declaration
  explicit-instantiation
  explicit-specialization
  linkage-specification
  namespace-definition
  namespace-declaration
  accessibility-specifier
  compound-declaration
  meta-control-declaration
  auto meta-control-declaration
  meta-expression-statement
  auto meta-expression-statement
  auto meta-class-declaration
  auto meta-function-definition
  auto meta-variable-declaration
  syntax-macro-definition
  include-declaration
  file-depency-declaration
  file-placement-declaration
  file-space-declaration

block-declaration:
  simple-declaration
  asm-definition
  namespace-alias-definition
  using-declaration
  using-directive

compound-declaration:
  { declaration-seq_opt }

simple-declaration:
  decl-specifier-seq_opt init-declarator-list_opt ;

decl-specifier:
  storage-class-specifier
  type-specifier
  function-specifier
  friend
  typedef
  access-specifier
  using

storage-class-specifier:
  auto
  register
  static
  extern
  mutable

static:
  static
  ! static

function-specifier:
  inline
  ! inline
  inline / implementation
  inline / interface
  virtual
  ! virtual
  virtual / pure
  explicit

4. using-declaration is generalised and covered by simple-declaration

5. Compatibility can be retained by allowing auto within a function-body.
type-specifier:
  simple-type-specifier
  class-specifier
  enum-specifier
  elaborated-type-specifier
  cv-qualifier

simple-type-specifier:
  +opt nested-name-specifier opt type-name scoped-id
  built-in-type-specifier

built-in-type-specifier:
  char
  wchar_t
  bool
  short
  int
  long
  signed
  unsigned
  float
  double
  void

elaborated-type-specifier:
  class-key +opt nested-name-specifier opt identifier scoped-id
  enum +opt nested-name-specifier opt identifier scoped-id
  typename +opt nested-name-specifier opt identifier opt scoped-id

  typename +opt nested-name-specifier opt identifier < template-argument-list >

tenum-specifier:
  enum +opt identifier opt scoped-id opt { enumerator-list opt }

namespace-declaration:
  namespace scoped-id ;

namespace-definition:
  named-namespace-definition
  unnamed-namespace-definition
  namespace scoped-id opt { namespace-body }

named-namespace-definition:
  original-namespace-definition
  extension-namespace-definition

original-namespace-definition:
  namespace identifier { namespace-body }

extension-namespace-definition:
  namespace original-namespace-name { namespace-body }

unnamed-namespace-definition:
  namespace { namespace-body }

using-declaration:
  using typename opt : : opt nested-name-specifier unqualified-id ;
  using : : unqualified-id ;

linkage-specification:
  extern string-literal { declaration-seq opt }
  extern string-literal declaration

A.7 Declarators

init-declarator-list:
  init-declarator
  init-declarator-list , init-declarator

6. The built-in types are split off to a distinct non-terminal.
init-declarator:
  declarator pure-specifieropt object-statements-clauseopt
declarator initializeropt object-statements-clauseopt
  identifieropt : constant-expression object-statements-clauseopt

cv-qualified:
  const
  volatile
declarator-id:
  ++ operator-expression
  ++ operator declared-name-specifiedtype-name scoped-id
  scoped-special-function-id

function-definition:
  decl-specifier-seqopt declarator pure-specifieropt ctor-initializeropt function-body
  decl-specifier-seqopt declarator pure-specifieropt function-try-block

initializer:
  = initializer-clause
  ( expression-list )
initializer-clause:
  assignment-expression
  | constant-expression assignment-expression
  | initializer-listopt
  | ]

A.8 Classes

class-head:
  class-key identifieropt scoped-idopt base-clauseopt
  class-key nested-name-specifiedidentifier base-clauseopt
class-specifier:
  class-head { member-specificationopt declaration-seqopt }
accessibility-specifier:
  access-specifier :
member-specification:
  member-declaration member-specificationopt
  access-specifier : member-specificationopt
member-declaration:
  decl-specifier-seqopt member-declarator-listopt : function-definitionopt
  qualified-id ;
  using-declaration
  template-declaration
member-declarator-list:
  member-declarator
  member-declarator-list , member-declarator

member-declarator:
  declarator pure-specifieropt
  declarator constant-initializeropt
  identifieropt : constant-expression

pure-specifier:
  = 0
constant-initializer:
  = constant-expression

---

7. The distinction between init-declarator and member-declarator is removed. The ambiguity of a bit-field with a labeled-statement is resolved to the label.
A.9  Derived Classes

base-specifier:
  \(+\_\)nested-name-specifier_opt class-name
  virtual access-specifier_opt \(+\_\)nested-name-specifier_opt class-name
  access-specifier virtual \(+\_\)nested-name-specifier_opt class-name
  scoped-id
  built-in-type-id
  virtual base-specifier
  ! virtual base-specifier
  access-specifier base-specifier
  auto base-specifier

A.10 Special member functions

mem-initializer-id:
  \(+\_\)nested-name-specifier_opt class-name
  identifier
  scoped-id

A.11 Overloading

A.12 Templates

template-declaration:
  export_opt using_opt template < template-parameter-list > declaration

explicit-specialization:
  using_opt template <> declaration

A.13 Exception Handling

A.14 Tree Literals

primary-tree-expression:
  meta-scoped-id
  ( tree-expression )

postfix-tree-expression:
  primary-tree-expression
  postfix-tree-expression ( tree-argument-list_opt )
  postfix-tree-expression [ expression ]
  postfix-tree-expression . scoped-id
  postfix-tree-expression -> scoped-id

expression:
  postfix-tree-expression
  * tree-expression

tree-argument-list:
  tree-argument
  tree-argument-list , tree-argument

tree-argument:
  terminated-tree-argument
  unterminated-tree-argument

tree-statement:
  terminated-tree-argument
  unterminated-tree-argument_opt ;

compound-tree-statement:
  ( tree-statement-seq_opt )

8. A more general rather than changed syntax.

9. Ambiguities are resolved semantically by left to right maximisation of the length of each tree-argument with respect to its required syntax.
10. An ambiguity arising from a try-block at the end of a statement followed by a handler-seq is resolved by maximising the length of the left-hand (try-block) element.

11. The presentation of these productions has huge ambiguities. They demonstrate some of the variety of C++ grammar that can be parsed in a context-free fashion. Very little cannot be. See Appendix C for an actual implementation that avoids the ambiguities.

12. do; is presumed to start an iteration-statement. operator, is presumed to be a declarator-id.
object-statement-seq:
  object-statement
  object-statement-seq_{opt} object-statement

object-statement:
  ;
  initializer ;
  function-used-block
  file-dependency-declaration
  file-placement-declaration
  filespace-declaration
  meta-control-declaration
  auto meta-control-declaration
  auto meta-expression-statement
  auto meta-function-definition
  auto meta-variable-declaration
  derived-clause object-statement
  derived-clause : { object-statement-seq_{opt} }

function-used-block:
ctor-initializer ;
ctor-initializer_{opt} function-body
function-try-block
using file-id-list function-used-block
segment function-used-block

segment:
entry
pre
body
post
exit

derived-clause:
derived ( meta-conditional-expression )

A.16 Meta-Programming

A.16.1 Meta-names

meta-id:
id
  meta-type
  auto
meta-nested-id:
  meta-id
  ~ meta-id
  meta-id :: meta-nested-id
meta-scoped-id:
  ::_{opt} meta-nested-id

A.16.2 Meta-classes

meta-class-id:
  meta-id
  meta-id :: meta-class-id
meta-class-specifier:
  meta-class-key meta-class-id
  meta-class-key meta-class-id base-specifier-clause_{opt} { declaration-seq_{opt} }
meta-class-declaration:
  meta-class-specifier ;
meta-class-key:
class-key
namespace
A.16.3 Meta-types

meta-type:
- meta-type-name
- built-in-type-id
- meta-class-key
- enum
typedef
typename
using

exposed-tree:
[
]

A.16.4 Meta-variables

meta-variable-declaration:
- static opt const opt meta-type meta-scoped-id exposed-tree opt = tree-statement
- static opt const opt meta-type meta-scoped-id exposed-tree opt object-statements-clause
- static opt const opt meta-type ( meta-scoped-id ) exposed-tree opt = tree-statement
- static opt const opt meta-type ( meta-scoped-id ) exposed-tree opt object-statements-clause

A.16.5 Meta-functions, Meta-constructors and Meta-destructors

meta-nested-constructor-id:
- meta-id
- meta-id :: meta-nested-constructor-id

meta-scoped-constructor-id:
- :: opt meta-nested-constructor-id

meta-nested-destructor-id:
- meta-id
- meta-id :: meta-nested-destructor-id

meta-scoped-destructor-id:
- :: opt meta-nested-destructor-id

meta-function-definition:
- static opt meta-type meta-scoped-id ( meta-parameter-list opt ) exposed-tree opt
  compound-tree-statement
- static opt meta-type meta-scoped-id ( meta-parameter-list opt ) exposed-tree opt
  object-statements-clause
- meta-scoped-constructor-id ( ) compound-tree-statement
- meta-scoped-constructor-id ( ) object-statements-clause
- meta-scoped-destructor-id ( ) compound-tree-statement
- meta-scoped-destructor-id ( ) object-statements-clause

meta-parameter-list:
- meta-parameter
- meta-parameter-list , meta-parameter

meta-parameter:
- meta-type identifier exposed-tree opt
- meta-type identifier exposed-tree opt = tree-argument

A.16.6 Meta-statements

meta-control-declaration:
- case constant-expression : declaration
default : declaration
do declaration while ( expression ) ;
for ( for-init-statement condition opt ; expression opt ) declaration
if ( condition ) declaration
if ( condition ) declaration else declaration
switch ( expression ) declaration
while ( condition ) declaration
jump-statement
A.16.7 Meta-expressions

meta-primary-expression:
  literal
  this
  meta-scoped-id
  meta-type meta-nested-id
tree-argument-list opt

meta-postfix-expression:
  meta-primary-expression
  meta-postfix-expression ( tree-argument-list opt )
  meta-postfix-expression [ expressionopt ]
  meta-postfix-expression . declarator-id
  meta-postfix-expression -> declarator-id
  meta-postfix-expression ++
  meta-postfix-expression --

meta-unary-expression:
  meta-postfix-expression
  ++ meta-unary-expression
  -- meta-unary-expression
  * meta-unary-expression
  + meta-unary-expression
  - meta-unary-expression
  ! meta-unary-expression
  ~ sizeof unary-expression

meta-multiplicative-expression:
  meta-unary-expression
  meta-multiplicative-expression * meta-unary-expression
  meta-multiplicative-expression / meta-unary-expression
  meta-multiplicative-expression % meta-unary-expression

meta-additive-expression:
  meta-multiplicative-expression
  meta-additive-expression + meta-multiplicative-expression
  meta-additive-expression - meta-multiplicative-expression

meta-shift-expression:
  meta-additive-expression
  meta-shift-expression « meta-additive-expression
  meta-shift-expression » meta-additive-expression

meta-relational-expression:
  meta-shift-expression
  meta-relational-expression < meta-shift-expression
  meta-relational-expression > meta-shift-expression
  meta-relational-expression <= meta-shift-expression
  meta-relational-expression >= meta-shift-expression

meta-equality-expression:
  meta-relational-expression
  meta-equality-expression == meta-relational-expression
  meta-equality-expression != meta-relational-expression

meta-and-expression:
  meta-equality-expression
  meta-and-expression & meta-equality-expression

meta-exclusive-or-expression:
  meta-and-expression
  meta-exclusive-or-expression ^ meta-and-expression

meta-inclusive-or-expression:
  meta-exclusive-or-expression
  meta-inclusive-or-expression | meta-exclusive-or-expression

meta-logical-and-expression:
  meta-inclusive-or-expression
  meta-logical-and-expression & & meta-inclusive-or-expression

meta-logical-or-expression:
  meta-logical-and-expression
  meta-logical-or-expression | | meta-logical-and-expression

meta-conditional-expression:
  meta-logical-or-expression
  meta-logical-or-expression ? meta-conditional-expression : meta-conditional-expression
A.17 Syntax macros

syntax-macro-definition:
  explicit auto meta-type identifier ( syntax-macro-parameter-list_opt ) exposed-tree_opt
  compound-tree-statement

syntax-macro-parameter-list:
  syntax-macro-parameter
  syntax-macro-parameter-list , syntax-macro-parameter

syntax-macro-parameter:
  meta-type identifier exposed-tree_opt
  identifier
  reserved-word
  punctuation

A.18 Files

string-expression:
  string-literal
  tree-literal

include-declaration:
  using slash-includex_opt slash-utility_opt string-expression ;

slash-include:
  / include

slash-utility:
  / utility

utility:
  emit
  pool
  utility
  frozen

file-dependency-declaration:
  using / implementation = opt file-specifier ;
  using / interface = opt file-specifier ;

file-placement-declaration:
  export / implementation = opt file-specifier ;
  export / interface = opt file-specifier ;
  export / noimplementation ;
  export / utility ;

file-specifier:
  file-name
  file-entity
  file-entity / implementation
  file-entity / interface

file-name:
  string-literal
  file-name / interface
  file-name / implementation
  file-name / template
  file-name / utility
  file-name / guard = string-expression
  file-name / noguard
  file-name / path = string-expression
  file-name / prefix = string-expression
  file-name / suffix = string-expression

file-entity:
  declarator-id
  elaborated-type-specifier
  namespace scoped-id

filesystem-specifier:
  namespace / file file-name compound-declaration
filespace-declaration:
    filespace-specifier ;
B C++ Grammar

The complete C++ grammar implemented using the superset approach outlined for FOG is presented in this appendix.

The presented grammar has been derived automatically from FogParser.y by a sed script to

- remove FOG specific grammar
- remove irrelevant action rules
- simplify relevant action rules
- remove implementation specific $ clutter

The resulting text is acceptable to yacc and has 0 unresolved conflicts.

It is available from

http://www.ee.surrey.ac.uk/Research/CSRG/foa/CxxGrammar.v
In return for not needing to use semantic information, the lexer must support back-tracking, which
is easily achieved by a simple linear buffer, a reference implementation of which may be found in the
accompanying CooxParser.cxx. Back-tracking is used to support:

* Binary search for a consistent parse of the template/arithemtic ambiguity.
  start_search() initiates the search
  advance_search() iterates the search
  end_search() cleans up after a search
  template_test() maintains context during a search
* Lookahead to resolve the inheritance/anonymous bit-field similarity
  mark() saves the starting context
  unmark() pops it
  rewind_colon() restores the context and forces the missing :
* Lookahead to resolve type : function parameter ambiguities
  mark_type() potentially marks the starting position
  mark() marks the pre ( position
  remark() rewinds to the starting position
  unmark() pops the starting position
* Note that lookaheads may nest.
*/

The parsing philosophy is unusual. The major ambiguities are resolved by creating a unified superset
grammar rather than non-overlapping subgrammars. Thus the grammar for parameter-declaration covers an
assignment-expression. Minor ambiguities whose resolution by supersetting would create more
ambiguities are resolved the normal way with partitioned subgrammars.
This eliminates the traditional expression/declaration and constructor/parenthesised declarator
ambiguities at the syntactic level. A subsequent semantic level has to sort the problems out.
The generality introduces four bogus ambiguities and defers the cast ambiguity for resolution
once semantic information is available.

The C++ grammar comprises 561 rules and uses 897 states in yacc, with 0 unresolved conflicts.
23 conflicts from 10 ambiguities are resolved by 8 %prec’s, so that yacc and bison report 0 conflicts.

The ambiguities are:
  1) dangling else resolved to inner-most if
     1 conflict in 1 state on else
  2) < as start-template or less-than
     1 conflict in 1 states on <
  3a) a:b:c resolved to favour a:b::c rather than a::b::c or a::b::c
     1 conflicts in 1 state for ::
  4) pointer operators maximised at end of conversion id/new in preference to binary operators
     2 conflicts in 4 states on * and &
  5a) (a)?b resolved to favour binary a??b rather than cast unary (a)?(b)
  5b) (a)(b) resolved to favour cast rather than call
     8 conflicts in 1 state for the 8 prefix operators: 6 unaries and ( and []
  6) enum name ( resolved to enum-specifier rather than function
     1 conflict in 1 state on 
  7) class name ( resolved to class-specifier rather than function
     1 conflict in 1 state on 
  8) extern "C" resolved to linkage-specification rather than declaration
     1 conflict in 1 state on StringLiteral
9) class X : forced to go through base-clause look-ahead

1 conflict in 1 state on :

10) id : forced to label_statement rather than constructor_head

0 conflicts - but causes a double state for 2)

of which

1 is a fundamental C conflict - always correctly resolved
   can be removed - see the Java spec

2, 3, 4 are fundamental C++ conflicts

2 always consistently resolved by iteration

3 always correctly resolved

4 always correctly resolved

5 is a result of not using type info - deferred for semantic repair

6, 7 are caused by parsing over-generous superset - always correctly resolved

8 is caused by parsing over-generous superset - always correctly resolved
   can be removed at the expense of 7 rules and 5 states.

9 is a look-ahead trick - always correctly resolved
   could be removed by marking one token sooner

10 is caused by parsing over-generous superset - always correctly resolved

The hard problem of distinguishing

class A ( class B : C, D, E )
-- A::B privately inherits C, D and E

class A ( class B : C, D, E ;
-- C is width of anon bit-field

is resolved by using a lookahead that assumes inheritance and rewinds for the bit-field.

The potential shift-reduce conflict on > is resolved by flattening part of the expression grammar
to know when the next > is template end or arithmetic >.

The grammar is SYNTACTICALLY context-free with respect to type. No semantic assistance is required
during syntactic analysis. However the cast ambiguity is deferred and must be recovered
after syntactic analysis of a statement has completed.

The grammar is SYNTACTICALLY context-free with respect to template-names. This is achieved by
organising a binary search over all possible template/arithmetic ambiguities with respect to
the enclosing statement. This is potentially exponentially inefficient but well-behaved in practice.
Approximately 1% of statements trigger a search and approximately 1% of those are misparsed.

requiring the semantic analysis to check and correct once template information is available.
1.5 parse attempts are required on average per ambiguous statement.

The grammar supports type I function declarations at severe impediment to efficiency. A lookahead
has to be performed after almost every non-statement close parenthesis. A one-line plus corollary
change to postfix_expression is commented and strongly recommended to make this grammar as
efficient as the rather large number of reduction levels permits.

Error recovery occurs mostly at the statement/declaration level. Recovery also occurs at
the list-element level where this poses no hazard to statement/declaration level recovery.

Note that since error propagation interacts with the lookaheads for template iteration or
type I function arguments, introduction of finer grained error recovery may repair a false
parse and so cause a misparse.

The following syntactic analysis errors occur, but are correctable semantically:
(cast) unary-op expr is parsed as (parenthesized) unary-op expr

The semantic test should look for a binary/call with a (type) as its left child.
 parenthesized) (arguments) is parsed as (cast) (parenthesized)

The semantic test should look for a cast with a non-type as its left child.

template < and arithmetic < may be cross-parsed (unless semantic help is provided)
approximately 0.01% are misparsed, and must be sorted out - not easy.
* The syntactic analysis defers the following ambiguities for semantic resolution:
  * declaration/expression is parsed as a unified concept
  * Use type and context to complete the parse.
  * -class-name is parsed as unary name
  * The semantic test should look for - with a type as its child.
  * delete[] expr is parsed as delete [[]]expr
  * The semantic test should look for delete with a [[]] cast of its child.
  * operator new/delete[] are parsed as array of operator new/delete
  * The semantic test should look for array of operator new/delete
  * or activate the two extra commented rules in operator
  * template of an explicit instantiation is buried deep in the tree
  * dig it out
  * pure-specifier and constant-initializer are covered by assignment-expression
  * just another of the deferred declaration/expression ambiguities
  * sizeof and typeid don’t distinguish type/value syntaxes
  * probably makes life polymorphically easier
*/

/*

The $prec resolves a conflict in identifier word: which is forced to be a shift of a label for
a labeled-statement rather than a reduction for the name of a bit-field or generalised constructor.
This is pretty dubious syntactically but correct for all semantic possibilities.
The shift is only activated when the ambiguity exists at the start of a statement. In this context
a bit-field declaration or constructor definition are not allowed.
*/
identifier word: Identifier
identifier: identifier word

/*
The $prec resolves the 14.2.3 ambiguity:
Identifier 'C' is forced to go through the is-it-a-template-name test
All names absorb TEMPLATE with the name, so that no template test is performed for them.
This requires all potential declarations within an expression to perpetuate this policy
and thereby guarantee the ultimate coverage of explicit instantiation.
*/
id: identifier
    $prec SHIFT THERE
    identifier template_test {' template_argument_list '}
    identifier template_test '+'
    identifier template_test '+' [ ']
    template_id
    '{' '/ Queue '+' or '+' < as follow on '/
template_test:
global_scope: { ' SCOPE
    TEMPLATE global_scope
    id_scope:
    { ' SCOPE
A : B : C; is ambiguous How much is type and how much name ?
The $prec maximises the (type) length which is the 7.1-2 semantic constraint.
*/
nested_id:
  id
  id_scope nested_id
scoped_id:
  nested_id
  global_scope nested_id

/*
 * destructor_id has to be held back to avoid a conflict with a one's complement as per 5.3.1-9.
 * It gets put back only when scoped or in a declarator_id, which is only used as an explicit member name.
 * Declarations of an unscoped destructor are always parsed as a one's complement.
*/
defstructor_id:
  '-' id
special_function_id:
  conversion_function_id
  operator_function_id
  TEMPLATE special_function_id
nested_special_function_id:
  special_function_id
  id_scope destructor_id
  id_scope nested_special_function_id
global_scope nested_special_function_id

/* declarator_id is all names in all scopes, except reserved words */
declarator_id:
  scoped_id
  scoped_special_function_id
defstructor_id

/* The standard defines pseudo-destructors in terms of type-name, which is class/enum/typedef, of which
 * class-name is covered by a normal destructor. Pseudo-destructors are supposed to support -intil in
 * templates, so the grammar here covers built-in names. Other names are covered by the lack of
 * identifier/type discrimination.
*/
built_in_type_id:
  built_in_type_specifier
  built_in_type_id built_in_type_specifier
pseudo_destructor_id:
  built_in_type_id SCOPE '-' built_in_type_id
  '-' built_in_type_id
  TEMPLATE pseudo_destructor_id
nested_pseudo_destructor_id:
  pseudo_destructor_id
scoped_pseudo_destructor_id:
  id_scope nested_pseudo_destructor_id
  nested_pseudo_destructor_id
global_scope scoped_pseudo_destructor_id

/*---------------------------------------------------------------*/

/*
 * A.2 Lexical conventions
---------------------------------------------------------------*/
/*
 * String concatenation is a phase 6, not phase 7 activity so does not really belong in the grammar.
 * However it may be convenient to have it here to make this grammar fully functional.
 * Unfortunately it introduces a conflict with the generalised parsing of extern "C" which
 * is correctly resolved to maximise the string length as the token source should do anyway.
*/
string:
  StringLiteral
  StringLiteral string -- Perverse order avoids conflicts -- */
literal:
  StringLiteral
  IntegerLiteral
  CharacterLiteral
  FloatingLiteral
string
boolean_literal
FALSE
TRUE

/*-----------------------------------------------*/
* A.3 Basic concepts
*-----------------------------------------------*/
translation_unit:
   declaration_seq.opt

/*-----------------------------------------------*/
* A.4 Expressions
*-----------------------------------------------*/
primary_expression:
   literal
   THIS
   suffix_decl_specified_ids
   -- covered by suffix_decl_specified_ids */
   SCOPE identifier
   -- covered by suffix_decl_specified_ids */
   SCOPE operator_function_id
   -- covered by suffix_decl_specified_ids */
   SCOPE qualified_id
   -- covered by suffix_decl_specified_ids */
   abstract_expression %prec REDUCE_HERE_MOSTLY
   -- prefer binary to unary ops, cast to call */
   id_expression
   -- covered by suffix_decl_specified_ids */

/*
Abstract-expression covers the [] and {} of abstract-declarators.
*/
abstract_expression:
   parenthesis_clause
   "[ [ expressionopt ] ]"
   TEMPLATE parenthesis_clause
/*
SIZEOF '{' type_id '}'
new_expression
  global_scope new_expression
delete_expression
  global_scope delete_expression
DELETE '{' '}' cast_expression
  /* covered by DELETE cast_expression since cast_expression covers ...
  SCOPE DELETE '{' '}' cast_expression
  ... abstract_expression cast_expression and so () cast_expression */
delete_expression:
  DELETE cast_expression
  /* also covers DELETE() cast_expression */
new_expression:
  NEW new_type_id new_initializer.opt
  NEW parameters_clause new_type_id new_initializer.opt
    NEW new_type_id
      NEW '{' type_id '}'
      NEW parameters_clause
        NEW '{' type_id '}'
        NEW parameters_clause parameters_clause new_initializer.opt
    new_type_id:
      type_specifier ptr_operator_seq.opt
      type_specifier new_type_id
new_declarator:
  ptr_operator new_declarator
direct_new_declarator:
  '{' expression '}'
new_initializer.opt:
  /* empty */
    '{' expression_list.opt '}'
  /* cast-expression is generalised to support a [] as well as a () prefix. This covers the omission of DELETE() which when */
  /* followed by a parenthesised expression was ambiguous. It also covers the gcc indexed array initialisation for free. */
cast_expression:
  abstract_expression cast_expression
  /*
    '{' type_id '}' cast_expression
  "-- covered by abstract_expression */
pm_expression:
  cast_expression
  pm_expression DOT_STAR cast_expression
  pm_expression ARROW_STAR cast_expression
  pm_expression
  multiplicative_expression star_ptr_operator pm_expression
    multiplicative_expression
      multiplicative_expression '+' multiplicative_expression
      multiplicative_expression '-' multiplicative_expression
    additive_expression:
      additive_expression
    shift_expression:
      shift_expression
      shift_expression SHR additive_expression
      shift_expression RELATED OPERATOR relational_expression
    relational_expression:
      shift_expression
      relational_expression '<' shift_expression
      relational_expression '>' shift_expression
      relational_expression LE shift_expression
      relational_expression GE shift_expression
    equality_expression:
      equality_expression EQ relational_expression
      equality_expression NE relational_expression
 */
and_expression: equality_expression
exclusive_or_expression: and_expression '\&' and_expression
inclusive_or_expression: exclusive_or_expression '\|' exclusive_or_expression
logical_and_expression: inclusive_or_expression '&&' and_expression
logical_or_expression: logical_and_expression LOG_AND inclusive_or_expression
conditional_expression: logical_or_expression '?:' assignment_expression

/* assignment_expression is generalised to cover the simple assignment of a
braced initializer in order to contribute to the
coverage of parameter-declaration and
init-declaration. */
assignment_expression: conditional_expression
logical_or_expression assignment_operator assignment_expression
logical_or_expression '=' braced_initializer
throw_expression

assignment_operator: '=' | ASS_AND | ASS_AND | ASS_DIV | ASS_MOD | ASS_MUL | ASS_OR | ASS_SHL | ASS_SHR | ASS_SUB | ASS_XOR

/* expression is widely used and usually single-element, so the reductions
are arranged so that a
* single-element expression is returned as is. Multi-element expressions are
parsed as a list that
* may then behave polymorphically as an element or be compacted to an element. */
expression_opt: /* empty */
expression
assignment_expression
expression_list ',' assignment_expression

constant_expression: conditional_expression

/* The grammar is repeated for when the parser stack knows that the next > must end a template. */
templated_relational_expression: shift_expression
templated_relational_expression '<' shift_expression
templated_relational_expression LE shift_expression
templated_relational_expression GE shift_expression
templated_relational_expression EQ templated_relational_expression
templated_relational_expression NE templated_relational_expression
templated_and_expression: templated_and_expression '&&' templated_and_expression
templated_exclusive_or_expression '^^' templated_and_expression
templated_inclusive_or_expression: templated_exclusive_or_expression
templated_inclusive_or_expression '||' templated_inclusive_or_expression

templated_logical_and_expression: templated_inclusive_or_expression
templated_logical_and_expression LOG_AND templated_inclusive_or_expression

templated_logical_or_expression: templated_logical_and_expression
templated_logical_or_expression LOG_OR templated_logical_and_expression

templated_conditional_expression: templated_logical_or_expression
C++ Grammar Meta-compilation for C++
jump_statement:       BREAK ';
                   CONTINUE ';
                   RETURN expression, opt ';
                   GOTO identifier ';

declaration_statement: block_declaration

/* A.6 Declarations */
compound_declaration: { 'nest declaration_seq, opt '}
                      { 'nest declaration_seq, opt util looping, declaration '}
                      { 'nest declaration_seq, opt util looping, declaration '}
                      'bang error ';
                      { unnest(); }
                      { unnest(); UNBang("Bad declaration-seq."); }

declaration_seq, opt:      /* empty */
                      declaration_seq, opt util looping, declaration
                      declaration_seq, opt util looping, declaration
                      'bang error ';
                      { UNBang("Bad declaration."); }

looping, declaration:
                      start_search looped, declaration
                      { end_search(); }

looped, declaration:
                      declaration
                      advance_search '+' looped, declaration
                      advance_search '+'

declaration:
                      block_declaration
                      function_definition
                      template_definition
                      explicit_instantiation
                      explicit_specialization
                      specialised_declaration
                      linkage_specification
                      namespace_definition
                      TEMPLATE specialised_declaration
                      simple_declaration
                      specialised_block_declaration
                      asm_definition
                      namespace_alias_definition
                      using_declaration
                      using_directive
                      TEMPLATE specialised_block_declaration

simple_declaration:
                      { '}
                      { init_declaration ';
                        init_declarations ';
                      }
                      decl_specifier_prefix simple_declaration

/* A decl-specifier following a ptr_operator provokes a shift-reduce conflict for
 * const name
 * which is resolved in favour of the pointer, and implemented by providing versions
 * of decl-specifier guaranteed not to start with a cv_qualifier.

 * decl-specifiers are implemented type-centrically. That is the semantic constraint
 * that there must be a type exploited to impose structure, but actually eliminate
 * very little syntax. built-in types are multi-name and so need a different policy.

 * non-type decl-specifiers are bound to the left-most type in a decl-specifier-seq,
 * by parsing from the right and attaching suffixes to the right-hand type. Finally
 * residual prefixes attach to the left.
*/
suffix, built_in, decl, specifier, raw: built_in_type_specifier
" suffix,built_in, decl, specifier, raw built_in_type_specifier
" suffix,built_in, decl, specifier, raw decl, specifier, suffix
suffix_built_in_declSpecifier:   suffix_built_in_declSpecifier.raw
suffix_named_declSpecifier:   scoped_id
                    | elaborate_typeSpecifier
suffix_named_declSpecifier.bi:   suffix_named_declSpecifier
                    | suffix_built_in_declSpecifier_suffix
suffix_named_declSpecifiers:   suffix_named_declSpecifier_suffix
                    | suffix_built_in_declSpecifier
suffix_named_declSpecifiers.sf:   suffix_named_declSpecifiers_suffix
                    | scoped_special_function_id /* operators etc */
suffix_declSpecified_ids:   suffix_built_in_declSpecifier
                    | suffix_built_in_declSpecifier_suffix
suffix_declSpecified_scope:   suffix_built_in_declSpecifier_suffix
                    | suffix_built_in_declSpecifier_suffix
                    | suffix_built_in_declSpecifier_suffix SCOPE
                    | suffix_built_in_declSpecifier_suffix SCOPE

declSpecifier_affix:   storage_classSpecifier
                    | functionSpecifier
                    | FRIEND
                    | TYPEDEF
                    | cv_qualifier

declSpecifier_suffix:   declSpecifier_affix

declSpecifier_prefix:   declSpecifier_affix
                    | TEMPLATE declSpecifier_prefix

storage_classSpecifier:   REGISTER | STATIC | MUTABLE
                    | EXTERN | auto
                    | *prec SHIFT_THere /* Prefer linkage specification */

functionSpecifier:   EXPLICIT
                    | INLINE
                    | VIRTUAL

typeSpecifier:   simple_typeSpecifier
                    | elaborate_typeSpecifier
                    | cv_qualifier

elaborate_typeSpecifier:   classSpecifier
                    | enumSpecifier
                    | elaborated_typeSpecifier
                    | TEMPLATE elaborate_typeSpecifier

simple_typeSpecifier:   scoped_id
                    | built_in_typeSpecifier

built_in_typeSpecifier:   CHAR | WCHAR_T | BOOL | SHORT | INT | LONG | SIGNED | UNSIGNED | FLOAT | DOUBLE | VOID

/*
 * The over-general use of declaration_expression to cover declSpecifier-seq.opt declarator in a function-definition means that
 * class X {};  
 * could be a function-definition or a classSpecifier.
 * enum X {};  
 * could be a function-definition or an enumSpecifier.
 * */
• The function-definition is not syntactically valid so resolving the false conflict in favour of the
• elaborated_type_specifier is correct.

| elaborated_type_specifier: | elaborated_class_specifier
| | elaborated_enum_specifier
| | TYPENAME scoped_id

| elaborated_enum_specifier: | ENUM scoped_id %prec SHIFT_THERE
| enum_specifier: | ENUM scoped_id enumerator_clause
| | ENUM enumerator_clause
| enumerator_clause: | '(' enumerator_list_ecarb
| | '(' enumerator_list enumerator_list_ecarb
| | '(' enumerator_list ',' enumerator_definition_ecarb
| enumerator_list_ecarb: | bang error ')
| | bang error ')
| enumerator_definition_ecarb: | '{' empty ')
| | '{' empty ')
| enumerator_definition_filler: | enumerator_definition_filler
| | enumerator_definition_filler
| | enumerator_list_head enumerator_definition
| enumerator_list: | enumerator_list
| | enumerator = constant_expression
| | identifier
| namespace_definition: | NAMESPACE scoped_id compound_declaration
| namespace_alias_definition: | NAMESPACE compound_declaration
| | NAMESPACE scoped_id 's scoped_id ';
| using_declaration: | USING declarator_id ';
| | USING TYPENAME declarator_id ';
| usingDirective: | USING NAMESPACE scoped_id ';
| asm_definition: | ASM '(' 'string' ') ';
| linkage_specification: | EXTERN string loopinDeclaration
| | EXTERN string compound_declaration

/*
 * A.7.8 Declarators
 */

/* this declaration is named init_declaration to reflect the embedded decl-specifier-seq-opt */

init_declarations: init_declarations init_declaration

| init_declaration: | assignment_expression
| | assignment_expression 's initializer_clause
| | assignment_expression '(' expression_list ')'

/*
 * this declaration is named direct_declarator */

star_ptr_operator: star_ptr_operator cv_qualifier

nested_ptr_operator: id_scope nested_ptr_operator

ptr_operator: ' &
nested_ptr_operator
ptr_operator_seq:
global_scope nested_ptr_operator
ptr_operator
ptr_operator_ptr_operator_seq

/* Independently coded to localise the shift-reduce conflict: sharing just needs another $prec */
ptr_operator_seqopt:
/* empty */
$prec SHIFT_THERE
/* Maximise type length */
cv_qualifier_seqopt:
/* empty */
ptr_operator_ptr_operator_seqopt
cv_qualifier:
cv_qualifier_seqopt cv_qualifier
CONST | VOLATILE /* cvQualifier */

/*type_id*/
typeSpecifier abstract_declarator.opt
typeSpecifier type_id
-- also covered by parameter declaration */

/*abstract_declarator*/
abstract_declarator.opt:
/* empty */
ptr_operator abstract_declarator.opt
direct abstract_declarator
/* empty */
direct abstract_declarator

direct abstract_declarator

direct abstract_declarator
direct abstract_declarator opt parenthesis_clause
direct abstract_declarator opt [ ]'
direct abstract_declarator opt { constant_expression }'

/*

parenthesis_clause:
parameters_clause cv_qualifier_seqopt
parameters_clause:
parameters_clause cv_qualifier_seqopt exception_specification

parameters_clause:
{ (' parameter_declaration_clause ' ) }'

/* parameter_declaration_clause also covers init_declarator, type_id, declarator and abstract_declarator. */
parameter_declaration_clause:
/* empty */
parameter_declaration_list
parameter_declaration_list ELLIPSIS
parameter_declaration_list ' , ' parameter_declaration

/ * A typed abstract qualifier such as
   class * ...
   * looks like a multiply, so pointers are parsed as their binary operation equivalents that
   * ultimately terminate with a degenerate right hand term. */
abstract_pointer_declaration:
multiplicative_expression star_ptr_operator ptr_operator_seqopt
abstract_parameter_declaration:
abstract_pointer_declaration
and_expression ' & '
abstract_pointer_declaration
special_parameter_declaration:
special_parameter_declaration
abstract_parameter_declaration ' , ' assignment_expression
ELLIPSIS
parameter_declaration:
assignment_expression
special_parameter_declaration
declSpecifier_prefix parameter_declaration

/* The grammar is repeated for use within template */
*/
tempered_parameter_declaration:  
  templated_assignment_expression  
  templated_abstract_declaration  
  templated_abstract_declaration '->' templated_assignment_expression

templated_abstract_declaration:  
  decl_specifier_prefix templated_parameter_declaration  
  abstract_pointer_declaration  
  templated_and_expression '->'  
  templated_and_expression '->' abstract_pointer_declaration

/* function_definition includes constructor, destructor, implicit int definitions too.  
 * A local destructor is successfully parsed as a function-definition but the - was treated as a unary operator.  
 * constructor_head is the prefix ambiguity between a constructor and a member-init-list starting with a bit-field.  
 */

function_definition:  
  ctor_definition  
  func_definition

func_definition:  
  assignment_expression function_try_block  
  assignment_expression function_body  
  decl_specifier_prefix func_definition

cctor_definition:  
  constructor_head function_try_block  
  constructor_head function_body  
  decl_specifier_prefix ctor_definition

constructor_head:  
  bit_field_init_declaration  
  constructor_head '->' assignment_expression

function_try_block:  
  TRY function_block handler_seq

function_block:  
  ctor_initializer_opt function_body

function_body:  
  compound_statement

/* An = initializer looks like an extended assignment_expression.  
 * An () initializer looks like a function call.  
 * initializer is therefore flattened into its generalised customers.  
 */

initializer:  
  '->' initializer_clause  
  '( expression_list ')'  
  -- flattened into caller

initializer_clause:  
  assignment_expression  
  braced_initializers  
  bracedInitializer:

braced_initializers:  
  'value_initializer_list',  
  'value_initializer_list', 'value_initializer_list', '

looping_initializer_clause:  
  looping_initializer_clause 'looping_initializer_clause'  
  initializer_list 'looping_initializer_clause'  
  initializer_list 'looping_initializer_clause'  
  UNBANG("Bad initializer_clause.");

looped_initializer_clause:  
  looped_initializer_clause 'looped_initializer_clause'  
  initializer_clause  
  start_search looped_initializer_clause  
  end_search();  
  advance_search '+' looped_initializer_clause  
  advance_search '-'

/* A.5 Classes
-------------------------

* An anonymous bit-field declaration may look very like inheritance:  
* class A : B = 3;
* class A : B;
* The two usages are too distant to try to create and enforce a common prefix so we have to resort to  
* a parser hack by backtracking. inheritance is much the most likely so we mark the input stream context
and try to parse a base-clause. If we successfully reach a { the base-clause is ok and inheritance was
the correct choice so we unmark and continue. If we fail to find the { an error token causes back-tracking
 to the alternative parse in elaborated_class_specifier which regenerates the : and declares unconditional success.

C++ Grammar Meta-compilation for C++

* colon_mark:  
  
  elaborated_class_specifier:  
  class_key scoped_id
  class_key scoped_id colon_mark error
  class_key scoped_id colon_mark base_specifier_list '{'
  
  class_specifier_head:  
  class_key ';
  base_specifier_list '{'
  class_key scoped_id id
  
  class_key:  
  CLASS | STRUCT | UNION
  
  class_specifier:  
  class_specifier_head member_specification.opt '{'
  class_specifier_head member_specification.opt '}
  member_specification.opt '{'
  member_specification.opt '}'
  [ UNBANG("Bad member_specification.opt."); ]
  
  member_specification.opt:  
  /* empty */
  member_specification.opt until looping_member_declaration
  member_specification.opt until looping_member_declaration '}'
  [ UNBANG("Bad member-declaration."); ]
  
  looping_member_declaration:  
  start_search looped_member_declaration
  advance_search '{' looped_member_declaration
  advance_search '}
  
  looped_member_declaration:  
  member_declaration
  accessibility_specifier
  single_member_declaration
  function_definition
  '/
  function_definition '/
  qualified_id '/
  using_declaration
  template_declarations
  
  / The genericity of constructor names (there need be no parenthesised argument list) means that that
  name : f(g), h(i)
  could be the start of a constructor or the start of an anonymous bit-field. An ambiguity is avoided by
  parsing the ctor-initializer of a function definition as a bit-field.
  
  simple_member_declaration:  
  /' assignment_expression '/
  constructor_head '/
  member_init_declarations '/
  
  member_init_declarations:  
  decl_specifier_prefix simple_member_declaration
  assignment_expression '/
  member_init_declaration
  constructor_head '/
  bit_field_init_declarations '/
  
  member_init_declaration:  
  assignment_expression
  assignment_expression '/
  initialiser_clause
  assignment_expression '/
  expression_list '/
  // covered by another set of call arguments
  
  accessibility_specifier
  bit_field_declaration:  
  assignment_expression '/
  bit_field_width
  
  bit_field_width:  
  logical_or_expression '/
  expression '/
  bit_field_width
  
  bit_field_init_declarations:  
  bit_field_init_declarations '/
  initializer_clause
/* A.9 Derived classes */

/* base_clause: */
/* base_specifier_list: */
/* base_specifier */
/* base_specifier_list ',' base specifier */
/* base_specifier */
/* scoped_id */
/* access specifier base specifier */
/* VIRTUAL base specifier */
/* access specifier: */
/* PRIVATE | PROTECTED | PUBLIC */

/* A.10 Special member functions */

/* conversion_function_id: */
/* conversion_type_id: */
/* type specifier ptr_operator_seq.opt */

/* Ctor-initialisers can look like a bit field declaration, given the generalisation of names: */
/* Class(Type) : m1(1), m2(2) { } */
/* NonClass(bit_field) : int(2), second_variable, ... */
/* The grammar below is used within a function try_block or function_definition. */
/* See simple_member_declaration for use in normal member function_definition. */

/* ctor_initializer.opt: */
/* empty */

/* mem_initializer: */
/* mem_initializer_list: */
/* mem_initializer_list bang error */
/* mem_initializer_list_head: */
/* mem_initializer_list bang error ',' */

/* A.11 Overloading */

/* operator_function_id: */

/* It is not clear from the ANSI standard whether spaces are permitted in delete[]. If not then it can */
/* be recognised and returned as DELETE_ARRAY by the lexer. Assuming spaces are permitted there is an */
/* ambiguity created by the over generalised nature of expressions. operator new is a valid declare-id */
/* which we may have an undimensioned array of. Semantic rubbish, but syntactically valid. Since the */
/* array form is covered by the declarator consideration we can exclude the operator here. The need */
/* for a semantic rescue can be eliminated at the expense of a couple of shift-reduce conflicts by */
/* removing the comments on the next four lines. */

/* operator: */
/* NEW */
/* DELETE */
/* NEW */
/* DELETE */
/* NEW [' ']' */
/* DELETE [' ']' */

-- Covered by array of OPERATOR NEW */
-- Covered by array of OPERATOR DELETE */
C++ Grammar

Meta-compilation for C++

March 1, 2002

Page 311

/*----------------------------------------*/
/* A.12 Templates */

/*----------------------------------------*/
template_declaration: template_parameter_clause declaration

template_parameter_clause: EXPORT template_declaration

template_parameter_list: TEMPLATE "<" template_parameter_list ">"

template_parameter: template_parameter_list ";" template_parameter

template_parameter: simple_type_parameter

simple_type_parameter: simple_type_parameter ";" type_id

template_type_parameter: template_type_parameter ";" identifier

template_parameter_list: template_type_parameter ";" identifier

bang error

{ UNBANG("Bad template-parameter."); }

-- covered by parameter_declaration */

CLASS identifier

-- covered by parameter_declaration */

/*----------------------------------------*/

/*----------------------------------------*/

/*----------------------------------------*/
template_type_parameter:  template_parameter_clause CLASS
| template_id:  template_parameter_clause CLASS identifier
| template_argument:  TEMPLATE identifier '<' template_argument_list '>'
| template_argument_list:   template_argument
| template_argument:   template_argument_list ',' template_argument
| template_name:   template_parameter_declaration
| error -- covered by template_parameter_declaration */
| /* Generalised naming makes identifier a valid declaration, so TEMPLATE identifier is too.
* The TEMPLATE prefix is therefore folded into all names, parenthesis_clause and decl specifier_prefix.
* */
| /* explicit_instantiation:  TEMPLATE declaration */
| explicit_specialization:  TEMPLATE '<' '>' declaration
| /*------------*/
| try_block:  TRY compound_statement handler_seq
| function_try_block:    handler handler_seq
| handler:  handler handler_seq
| exception_declaration:  parameter_declaration
| throw_expression:  THROW
| throw_assignment_expression:  THROW
| throw('')
| throw('(' type_id_list ')')
| type_id_list:  type_id
| type_id_list '',' type_id
| /* Back-tracking and context support
* */
| advance_search:  error
| bang:  /* empty */
| mark:  /* empty */
| nest:  /* empty */
| start_search:  /* empty */
| start_search1:  /* empty */
| util:  /* empty */
| /* yerrno, yyclear; advance_search(); } /* Rewind and queue '+' or '-' '#' */
| /* set flag to suppress 'parse error' */
| /* Push lookahead and input token stream context onto a stack */
| /* Push a declaration nesting depth onto the parse stack */
| /* Reset binary search context */
| /* Create/reset binary search context */
| /* Get current utility mode */
FOG Grammar

The complete FOG grammar implemented using the superset approach outlined for FOG is presented in this appendix.

The presented grammar has been derived automatically from FogParser.y by a sed script to

- remove C++ specific grammar
- remove irrelevant action rules
- simplify relevant action rules
- remove implementation specific $ clutter

The resulting text is acceptable to yacc and has 0 unresolved conflicts.

It is available from

http://www.ee.surrey.ac.uk/Research/CSRG/foa/FogGrammar.v
This is a yacc-able parser for the entire FOG grammar with no unresolved conflicts. /*
The parse is SYNTACTICALLY consistent and requires no template or type name assistance.
The grammar in the C++ standard notes that its grammar is a superset of the true
grammar requiring semantic constraints to resolve ambiguities. This grammar is a really big
superset unifying expressions and declarations, eliminating the type/non-type distinction.
and iterating to find a consistent solution to the template/arith,metaic < ambiguity.
As a result the grammar is much simpler, but requires the missing semantic constraints to be
performed in a subsequent semantic pass, which is of course where they belong. This grammar will
support conversion of C++ tokens into an Abstract Syntax Tree. A lot of further work is required to
make that tree useful.

The principles behind this grammar are described in my thesis on Meta-Compilation for C++, which
may be found via http://www.computing.surrey.ac.uk/research/dsrg/fog/FogThesis.html.

Author: E.D.Willink Ed.Willink@rrl.co.uk
Date: 15-Jun-2001

The lexer (and/or a preprocessor) is expected to identify the following

Punctuation:
/*

*/
Punctuation sequences
/*
term <keyword> ARROW ARROW_STAR DEC EQ GE INC LE LOG_AND LOG_OR NE SHL SHR

term <keyword> ASS_ADD ASS_AND ASS_DIV ASS_MOD ASS_MUL ASS_OR ASS_SHL ASS_SHR ASS_SUB ASS_XOR

term <keyword> DOT_STAR ELLIPSIS SCOPE

*/
Reserved words
/*

term <access_specifier> PRIVATE PROTECTED PUBLIC

term <built_in_id> BOOL CHAR DOUBLE FLOAT INT LONG SHORT SIGNED UNSIGNED VOID WCHAR_T

term <class_key> CLASS ENUM NAMESPACE STRUCT TYPENAME UNION

term <cv_qualifiers> CONST VOLATILE

term <decl_specifier_id> AUTO EXPLICIT EXPORT EXTERN FRIEND INLINE MUTABLE REGISTER STATIC TEMPLATE TYPEDEF USING VIRTUAL

term <keyword> ASM BREAK CASE CATCH COND_CAST CONTINUE DEFAULT DELETE DO DYNAMIC_CAST

term <keyword> ELSE FALSE FOR GOTO IF NEW_OPERATOR REINTERPRET_CAST RETURN

term <keyword> SIZEOF STATIC_CAST SWITCH THIS THROW TRUE TRY TYPEID WHILE

*/

Parametric values.
/*

term <character_literal> CharacterLiteral

term <floating_literal> FloatingLiteral

term <identifier> Identifier

term <integer_literal> IntegerLiteral

term <number_literal> NumberLiteral

term <string_literal> StringLiteral

*/

FOG non-reserved word identifier extensions
/*

term <built_in_id> BuiltInTypeSpecifier

term <meta_type> MetaType

term <name> TrueLiteral


The parser needs to treat '0' as distinct from IntegetLiteral in the hope that pure-specifier can
be distinguished. It isn't. Semantic rescue from a constant-expression is necessary.

The lexer is not required to distinguish template or type names, although a slight simplification to the
grammar and elaboration of the action rules could make good use of template name information.

In return for not needing to use semantic information, the lexer must support back-tracking, which
is easily achieved by a simple linear buffer, a reference implementation of which may be found in the
accompanying CxxParser.cxx. Back-tracking is used to support:

- Binary search for a consistent parse of the template/arithemtic ambiguity.
- start_search() initialises the search
- advance_search() iterates the search
- end_search() cleans up after a search
- template_test() maintains context during a search

Lookahead to resolve the inheritance/anonymous bit-field similarity
- mark() saves the starting context
- unmark() pops it
- rewind_colon() restores the context and forces the missing :

Lookahead to resolve type 1 function parameter ambiguities
- mark_typep() potentially marks the starting position
- mark() marks the pre (position
- remark() rewrites to the starting position
- unmark() pops the starting position

Note that lookahead may nest.

The parsing philosophy is unusual. The major ambiguities are resolved by creating a unified superset
grammar rather than non-overlapping subgrammars. Thus the grammar for parameter-declaration covers an
assignment-expression. Minor ambiguities whose resolution by supersetting would create more
ambiguities are resolved the normal way with partitioned subgrammars.
This eliminates the traditional expression/declaration and constructor/parenthesised declarator
ambiguities at the syntactic level. A subsequent semantic level has to sort the problems out.
The generality introduces four bogus ambiguities and defers the cast ambiguity for resolution
once semantic information is available.

The FOG grammar comprises 958 rules and uses 1565 states in yacc, with 0 unresolved conflicts.
31 conflicts from 15 ambiguities are resolved by 15 %prec's, so that yacc and bison report 0 conflicts.

The ambiguities are:
1) dangling else resolved to inner-most if
   1 in 2 states on else
2) < as start-template or less-than
   1 conflict in 1 states on <
3) a :: b :: c resolved to favour a::b::c rather than a::b::c or a::b::c
   1 conflicts in 1 state for ::
4) pointer operators maximised at end of conversion id/new in preference to binary operators
   2 conflicts in 4 states on * and &
5a) (a|b) resolved to favour binary a|b rather than cast unary (a|(0b)
5b) (a|(O) resolved to favour cast rather than call
   8 conflicts in 1 state for the 8 prefix operators: 6 unaries and ( and [
   6) enum name resolved to enum-specifier rather than function
   1 conflict in 1 state on 
   7) class name ( resolved to class-specifier rather than function
   1 conflict in 1 state on 
   8) extern "C" resolved to linkage-specification rather than declaration
   1 conflict in 1 state on StringLiteral
   9) class X forced to go through base-clause look-ahead
   1 conflict in 1 state on:
   10) id : forced to label_statement rather than constructor_head
   1 conflict in 1 state on
   11) access-specifier : forced to access-declaration rather than anon bit-field
   1 conflict in 1 state on
   12) inline/ and virtual/ forced to switch rather than divide treatment
   1 conflict in 2 states on /
   13) using StringLiteral forced to include declaration not simple declaration
   1 conflict in 1 states on StringLiteral
   14) handler_seq maximised avoiding ambiguity in compound_tree_statement
   1 conflict in 1 states on catch
15) built_in_type_id maximised resolving ambiguity for auto unsigned int : : a
   1 conflict in 1 states on BuiltInTypeSpecifier
   of which
   1 is a fundamental C conflict - always correctly resolved
      can be removed - see the Java spec
   2, 3, 4 are fundamental C++ conflicts
   2 always consistently resolved by iteration
   3 always correctly resolved
   4 always correctly resolved
   5 is a result of not using type information - deferred for semantic repair
   6, 7 are caused by parsing over-generous superset - always correctly resolved
   8 is caused by parsing over-generous superset - always correctly resolved
   can be removed at the expense of 7 rules and 5 states.
   9 is a look-ahead trick - always correctly resolved
      could be removed by marking one token sooner
   10 is caused by parsing over-generous superset - always correctly resolved
   11 is caused by parsing over-generous superset - always correctly resolved
   12 is caused by parsing over-generous superset - always correctly resolved
   13 is caused by parsing over-generous superset - always correctly resolved
   14 is a genuine conflict - always correctly resolved by definition
      more enthusiastic parsing of the } or ; statement end could fix this
   15 is a fundamental FOG conflict comparable to 3
      always correctly resolved

The hard problem of distinguishing
1. class A { class B : C, D, E } -- A::B privately inherits C, D and E
2. class A { class B : C, D, E, F } -- C is width of anon bit-field
   is resolved by using a lookahead that assumes inheritance and rewrites for the bit-field.

The potential shift-reduce conflict on > is resolved by flattening part of the expression grammar
   to know when the next > is template end or arithmetic >.

The grammar is SYNTACTICALLY context-free with respect to type. No semantic assistance is required
during syntactic analysis. However the cast ambiguity is deferred and must be recovered
after syntactic analysis of a statement has completed.

The grammar is SYNTACTICALLY context-free with respect to template-names. This is achieved by
organising a binary search over all possible template/arithemtic ambiguities with respect to
the enclosing statement. This is potentially exponentially inefficient but well-behaved in practice.
Approximately 1% of statements trigger a search and approximately 1% of those are misparsed.
requiring the semantic analysis to check and correct once template information is available.
1.5 parse attempts are required on average per ambiguous statement.

The grammar supports type 1 function declarations at severe impediment to efficiency. A lookahead
has to be performed after almost every non-statement close parenthesis. A one-line-plus corollary
change to postfix_expression is commented and strongly recommended to make this grammar as
efficient as the rather large number of reduction levels permits.

Error recovery occurs mostly at the statement/declaration level. Recovery also occurs at
the list-element level where this poses no hazard to statement/declaration level recovery.
Note that since error propagation interacts with the lookaheads for template iterate or
type 1 function arguments, introduction of finer grained error recovery may repair a false
parse and so cause a misparse.

The following syntactic analysis errors occur, but are correctable semantically:
(cast) unary-op expr is parsed as (parenthesised)binary-op expr
The semantic test should look for a binary/call with a (type) as its left child.
(parenthesised) (arguments) is parsed as (cast)(parenthesised)
The semantic test should look for a cast with a non-type as its left child.
template < and arithmetic < may be cross-parsed (unless semantic help is provided)
approximately 0.01% are misparsed, and must be sorted out - not easy.

The syntactic analysis defers the following ambiguities for semantic resolution:
declaration/expression is parsed as a unified concept
use type and context to complete the parse.
-class-name is parsed as unary- name
The semantic test should look for - with a type as its child.
delete[] expr is parsed as delete []expr
The semantic test should look for delete with a [] cast of its child.
operator new/delete[] are parsed as array of operator new/delete
The semantic test should look for array of operator new/delete
or activate the two extra comment rules in operator
template of an explicit instantiation is buried deep in the tree
dig it out
pure-specifier and constant-initializer are covered by assignment-expression
just another of the deferred declaration/expression ambiguities
sizeof and typeid don't distinguish type/value syntaxes
probably makes life polymorphically easier*

/*
%nonassoc SHIFT THERE
%nonassoc SCOPE ELSE INC DEC '; ' ' ' ' ' '(' ' '{' ' '<' ' ':' StringLiteral
'//' CATCH BuiltInTypeSpecifier
%nonassoc REDUCE HERE MOSTLY
%nonassoc '{' /*%nonassoc REDUCE HERE */
%start translation_unit
*/
\*
\* The %prec resolves a conflict in identifier_word: which is forced to be a shift of a label for
\* a labeled-statement rather than a reduction for the name of a bit-field or generalised constructor.
\* This is pretty dubious syntactically but correct for all semantic possibilities.
\* The shift is only activated when the ambiguity exists at the start of a statement. In this context
\* a bit-field declaration or constructor definition are not allowed.
\*/
identifier_word:
  Identifier
    MimeType
      DERIVED | FILE | GUARD | IMPLEMENTATION
      INCLUDE | INTERFACE | NOGUARD | NOIMPLEMENTATION
    OVERLOAD | PATH | PREFIX | PURE | SUFFIX
    segment
    utility
    identifier_word %prec SHIFT_HERE
    Treeliteral
\*/
\* The %prec resolves the 14.2-3 ambiguity:
\* Identifier '<' is forced to go through the is-it-a-template-name test
\* All names absorb TEMPLATE with the name, so that no template_test is performed for them.
\* This requires all potential declarations within an expression to perpetuate this policy
\* and thereby guarantee the ultimate coverage of explicit instantiation.
\*/
id:
  identifier %prec SHIFT_HERE /* Force < through test */
    identifier template_test '+' template_argument_list '>'
    identifier template_test '+' '>' (ERRMSG("Empty template-argument-list");)
    identifier template_test '<' %prec SHIFT_HERE /* requeued < follows */
    template_id '<' /* Queue '+' or '-' < as follow on */
    (template_test());
    (IS_DEFAULT;)
    (IS_TEMPLATE;)


global_scope:
  SCOPE
    TEMPLATE global_scope
    id SCOPE
\*/
\* A :: B :: C is ambiguous. How much is type and how much name?
\* The %prec maximises the (type) length which is the 7.1-2 semantic constraint.
\*/
nested_id:
  id %prec SHIFT_HERE /* Maximize length */
    id_scope nested_id
  nested_id
  global_scope nested_id
\*/
\* destructor_id has to be held back to avoid a conflict with a one's complement as per 5.3.1-9,
\* It gets put back only when scoped or in a declarator_id, which is only used as an explicit member name.
\* Declarations of an unscoped destructor are always parsed as one's complement.
\*/
destructor_id:
  '-' id
  TEMPLATE destructor_id
  conversion_function_id
  operator_function_id
  TEMPLATE special_function_id

nested_special_function_id:
  special_function_id
    id_scope destructor_id
    id_scope nested_special_function_id

scoped_special_function_id:
  nested_special_function_id
    id_scope nested_special_function_id
    global_scope nested_special_function_id
/* declarator-id is all names in all scopes, except reserved words */
defclarator_id: scoped_id scoped_special_function_id destructor_id

/* The standard defines pseudo-destructors in terms of type-name, which is class/enum/typedef, of which
 * class-name is covered by a normal destructor. pseudo-destructors are supposed to support -int() in
 * templates, so the grammar here covers built-in names. Other names are covered by the lack of
 * identifier/type discrimination.
 */
built_in_type_id: built_in_typeSpecifier
pseudo_destructor_id: built_in_type_id built_in_typeSpecifier
                     built_in_type_id "-" built_in_type_id
                     TEMPLATE pseudo_destructor_id
nested_pseudo_destructor_id: id_scope nested_pseudo_destructor_id
scoped_pseudo_destructor_id: nested_pseudo_destructor_id
global_scope scoped_pseudo_destructor_id

/*****************************/

/ A.2 Lexical conventions
/*****************************/
/*
 * String concatenation is a phase 6, not phase 7 activity so does not really belong in the grammar.
 * However it may be convenient to have it here to make this grammar fully functional.
 * Unfortunately it introduces a conflict with the generalised parsing of extern "C" which
 * is correctly resolved to maximise the string length as the token source should do anyway.
 */
string: StringLiteral
	/*string:
 | StringLiteral
 | StringLiteral string -- Perverse order avoids conflicts -- */
/* literal:
 | IntegerLiteral
 | CharacterLiteral
 | FloatingLiteral
 | string
 | boolean literal
 | string
 | TreeLiteral
 | FALSE
 | TRUE

/ A.3 Basic concepts
/*****************************/

translation_unit: declaration_seq.opt

/ expression grammar */
	/* tree_expression
 | '{' tree_expression ecarb
 | '}' bang error
 | '}' bang error ecarb
*/

/ A.4 Expressions
primary_expression covers an arbitrary sequence of all names with the exception of an unscoped destructor, which is parsed as its unary expression which is the correct disambiguation (when ambiguous). This eliminates the traditional A(B) meaning A B ambiguity, since we never have to tack an A onto the front of something that might start with ( The name length got maximised ab initio. The downside is that semantic interpretation must split the names up again.

Unification of the declaration and expression syntax means that unary and binary pointer declarator operators:
int * name
are parsed as binary and unary arithmetic operators (int) * (*name). Since type information is not used ambiguities resulting from a cast {cast} *(value)
are resolved to favour the binary rather than the cast unary to ease AST clean-up.
The cast-call ambiguity must be resolved to the cast to ensure that {a}(b) can be parsed.

The problem of the functional cast ambiguity
name(arg)
as call or declaration is avoided by maximising the name within the parsing kernel. So
primary_id_expression picks up
extern long int const var = 5;
as an assignment to the syntax parsed as "extern long int const var". The presence of two names is parsed so that "extern long into const" is distinguished from "var" considerably simplifying subsequent semantic resolution.

The generalised name is a concatenation of potential type-names (scoped identifiers or built-in sequences) plus optionally one of the special names such as an operator-function-id, conversion-function-id or destructor as the final name.

/*
 primary_expression:
   literal
   THIS
   suffix_decl_specified_ids
   SCOPE identifier
   SCOPE operator_function_id
   SCOPE qualified_id
   abstract_expression %prec REDUCE_HERE_MOSTLY
   id_expression -- covered by suffix_decl_specified_ids */
   -- covered by suffix_decl_specified_ids */
   -- covered by suffix_decl_specified_ids */
   -- covered by suffix_decl_specified_ids */
   -- covered by suffix_decl_specified_ids */

   abstract_expression covers the ( and ) of abstract-declarators.
   */
   abstract_expression:
      parenthesis_clause
      ' expression.opt ']
      TEMPLATE parenthesis_clause
      */
   Type I function parameters are ambiguous with respect to the generalised name, so we have to do a lookahead following any function-like parentheses. This unfortunately hits normal code, so kill the -- lines and add the ++ lines for efficiency.
   Supporting Type I code under the superset causes perhaps 25% of lookahead parsing. Sometimes complete class definitions get traversed since they are valid generalised type I parameters!

typel_parameters:
   "-----" parameter_declaration_list ;'
   "-----" typel_parameters parameter_declaration_list ;'
mark_typel:
   /* empty */
   { mark_typel(); yyclearin; }
postfix_expression:
   primary_expression
   /*
   "++++++" postfix_expression parenthesis_clause */
   "-----" postfix_expression parenthesis_clause mark_typel ;'
   "-----" postfix_expression parenthesis_clause mark_typel ;
   typel_parameters mark ;' error
{ yerror; yclearin; remark_type1(); unmark(); unmark(); }
postfix_expression parenthesis_clause mark_type1 '*' type1_parameters mark error
{ yerror; yclearin; remark_type1(); unmark(); unmark(); }
postfix_expression parenthesisClause mark_type1 '*' error
{ yerror; yclearin; remark_type1(); unmark(); unmark(); }

/*-----*/
postfix_expression expression[opt] {'| expression[opt] '|
/*-----*/
derector_id {'| expression[opt] '|
/*-----*/
simple_typeSpecifier {'| expression_list[opt] '|
/*-----*/
postfix_expression declaration_id
/*-----*/
postfix_expression TEMPLATE declaration_id
/*-----*/
postfix_expression scoped_pseudo_destructor_id
/*-----*/
postfix_expression ARROW declaration_id
/*-----*/
postfix_expression ARROW TEMPLATE declaration_id
/*-----*/
postfix_expression ARROW scoped_pseudo_destructor_id
/*-----*/
postfix_expression INC
/*-----*/
postfix_expression INC
/*-----*/

TYPE ID parameters_clause
/*-----*/

expression_list[opt] {'| empty '|
/*-----*/
expression_list
expression_list[opt] {'| assignment_expression
expression_list '| assignment_expression
/*-----*/

Unary_expression:
/*-----*/

postfix_expression INC cast_expression
/*-----*/
DEC cast_expression
/*-----*/
ptr_operator cast_expression
/*-----*/
'&' cast_expression
/*-----*/
decl_specifier_seq '*' cast_expression
/*-----*/
decl_specifier_seq '&' cast_expression
/*-----*/
suffix_decl_specified_scope star ptr_operator cast_expression /* covers e.g int ::type::* const t = 4 */

/*-----*/

cast_expression
/*-----*/
'.' cast_expression
/*-----*/
';' cast_expression
/*-----*/
'~' cast_expression
/*-----*/
'+' cast_expression
/*-----*/
'-' cast_expression
/*-----*/
'SIZEOF unary_expression
/*-----*/
SIZEOF {'| type_id '|
/*-----*/

new_expression
/*-----*/
global_scope new_expression
/*-----*/
delete_expression
/*-----*/
delete_expression
/*-----*/
delete_expression
/*-----*/
delete_expression
/*-----*/
new_expression:
/*-----*/
new_expression:
/*-----*/
new_expression:
/*-----*/
new_type_id new_initializer[opt]
/*-----*/
new_parameters_clause new_type_id new_initializer[opt]
NEW parameters_clause
NEW '{ type-id ')
NEW parameters_clause_parameters_clause new_initializer.op
NEW '{ type-id ') new_initializer
NEW parameters_clause '{ type-id ')
NEW parameters_clause_parameters_clause_parameters_clause

new_type_id:
  type_specifier ptr_operator_seq.op
  type_specifier new_declarator
  type_specifier new_type_id
new_declarator:
  ptr_operator new_declarator
direct_new_declarator:
  '{ expression ')
direct_new_declarator '{ constant_expression ')
new_initializer.op:
  '/ empty */
  '{ expression_list.op ')

/* cast-expression is generalised to support a [ ] as well as a ( ) prefix. This covers the omission of DELETE[ ] which when
  followed by a parenthesised expression was ambiguous. It also covers the gcc indexed array initialisation for free.
*/
cast_expression:
  unary_expression
  abstract_expression cast_expression
  '{ type-id ') cast_expression

pm_expression:
  cast_expression
  pm_expression DOT_STAR cast_expression
  pm_expression ARROW_STAR cast_expression
multiplicative_expression:
  pm_expression
  multiplicative_expression star_ptr_operator pm_expression
  multiplicative_expression pm_expression
  multiplicative_expression pm_expression
additive_expression:
  multiplicative_expression
  additive_expression '+' multiplicative_expression
  additive_expression '-' multiplicative_expression
shift_expression:
  additive_expression
  shift_expression SHR additive_expression
  shift_expression SHR additive_expression
relational_expression:
  shift_expression < shift_expression
  shift_expression > shift_expression
  relational_expression LE shift_expression
  relational_expression GE shift_expression
equality_expression:
  relational_expression
  equality_expression EQ relational_expression
  equality_expression NE relational_expression
and_expression:
  equality_expression
  and_expression ' & ' equality_expression
exclusive_or_expression:
  and_expression
  exclusive_or_expression ' ^ ' and_expression
inclusive_or_expression:
  exclusive_or_expression
  inclusive_or_expression ' | ' exclusive_or_expression
logical_and_expression:
  inclusive_or_expression
  logical_and_expression LOG_AND inclusive_or_expression
logical_or_expression:
  logical_and_expression
  logical_or_expression LOG_OR logical_and_expression
conditional_expression:
  logical_or_expression
  conditional_expression ' ? ' expression ':' assignment_expression
/* assignment-expression is generalised to cover the simple assignment of a braced initializer in order to contribute to the */
/* coverage of parameter-declaration and init-declaration. */
/*
*/
 assignment_expression: conditional_expression
  | logical_or_expression assignment_operator assignment_expression
  | logical_or_expression '{' braced_initializer
  | throw_expression
 assignment_operator: '=' | ASS_ADD | ASS_AND | ASS_DIV | ASS_MOD | ASS_MUL | ASS_OR | ASS_SHL | ASS_SHR | ASS_SUB | ASS_XOR

/* expression is widely used and usually single-element, so the reductions are arranged so that a */
/* single-element expression is returned as is. Multi-element expressions are parsed as a list that */
/* may then behave polymorphically as an element or be compacted to an element. */
/* expression.opt: */
  | empty */
 expression: assignment_expression
  | expression_list ',' assignment_expression
 constant_expression: conditional_expression

/* The grammar is repeated for when the parser stack knows that the next > must end a template. */

templated_relational_expression: shift_expression
  | templated_relational_expression '<' shift_expression
  | templated_relational_expression 'LE' shift_expression
  | templated_relational_expression 'EQ' templated_relational_expression
  | templated_relational_expression 'NEQ' templated_relational_expression
  | templated_relational_expression 'LT' templated_relational_expression
  | templated_relational_expression 'GT' templated_relational_expression
 templated_equality_expression: templated_equality_expression '&&' templated_equality_expression
  | templated_equality_expression '||' templated_equality_expression
 templated_and_expression: templated_and_expression '&&' templated_and_expression
  | templated_and_expression '||' templated_and_expression
 templated_exclusive_or_expression: templated_exclusive_or_expression '&&' templated_exclusive_or_expression
  | templated_exclusive_or_expression '||' templated_exclusive_or_expression
 templated_inclusive_or_expression: templated_inclusive_or_expression '&&' templated_inclusive_or_expression
  | templated_inclusive_or_expression '||' templated_inclusive_or_expression
 templated_logical_and_expression: templated_logical_and_expression 'LOG_AND'
  | templated_logical_and_expression 'LOG_OR'
 templated_logical_or_expression: templated_logical_or_expression 'LOG_AND' templated_logical_or_expression
  | templated_logical_or_expression 'LOG_OR' templated_logical_or_expression
 templated_conditional_expression: templated_logical_or_expression '?'
  | templated_logical_or_expression ':' templated_assignment_expression
 templated_assignment_expression: templated_conditional_expression
  | templated_assignment_expression assignment_operator templated_assignment_expression
 templated_expression: templated_assignment_expression
  | templated_expression_list ',' templated_assignment_expression
 templated_expression_list: templated_assignment_expression
  | templated_assignment_expression ',' templated_assignment_expression
/** A.5 Statements */
* Parsing statements is easy once simple_declaration has been generalised to cover expression_statement.
*/
looped_statement: start_search looped_statement
| looped_statement: advance_search '=' looped_statement
| state: control_statement
expression_statement: -- covered by declaration_statement
compound_statement: try_block
AUTO control_statement
AUTO meta_expression_statement
labeled_statement: labeled_statement
| iteration_statement: identifier_word ':' looped_statement
expression.opt: DEFAULT ':' looped_statement
| compound_statement: '"' statement_seq.opt '"'
statement_seq.opt: '"' statement_seq.opt looped_statement '##' bang error'"' [UNBANG("Bad statement-seq.");]
| empty
| empty
| empty

/* The dangling else conflict is resolved to the innermost if. */
selection_statement: IF '"' condition '"' looped_statement $prec SHIFT_THERE
| IF '"' condition '"' looped_statement ELSE looped_statement
| SWITCH '"' condition '"' looped_statement
| empty
condition: parameter_declaration_list
| expression: type_specifier_seq declarator '=' assignment_expression
| empty
| empty
iteration_statement: WHILE '"' condition '"' looped_statement
DO looped_statement WHILE '"' expression '"' ;
| FOR '"' for_init_statement condition_opt '"' expression_opt '"' looped_statement
for_init_statement: simple_declaration
expression_statement: -- covered by simple_declaration
jump_statement: BREAK '"'
| CONTINUE '"'
| RETURN expression.opt '"'
| GOTO identifier '"'
declaration_statement: block_declaration
*/

/* A.6 Declarations */
compound_declaration: '"' nest declaration_seq.opt '"'
| unnest();
*/
/* empty */

looping_abbreviation;

lopped_abbreviation:
  advance_search ‘+’ looped_abbreviation
  advance_search ‘-’

lined_abbreviation;

declaration;

  template_abbreviation
  explicit_abbreviation
  explicit_specialization
  specialized_abbreviation
  accessibility_abbreviation
  compound_abbreviation
  meta_control_abbreviation
  AUTO meta_control_abbreviation
  AUTO meta_class_abbreviation
  AUTO meta_expression_abbreviation
  AUTO meta_function_abbreviation
  syntax_macro_abbreviation
  include_abbreviation
  file_dependency_abbreviation
  file_placement_abbreviation
  filesystem_abbreviation
  linkage_specification
  namespace_abbreviation
  namespace_abbreviation
  TEMPLATE specialized_abbreviation
  simple_abbreviation
  specialized_block_abbreviation
  asm_abbreviation
  namespace_alias_abbreviation
  using_abbreviation
  using_directive
  TEMPLATE specialized_block_abbreviation

  ';'
  init_abbreviation
  constructor_head
  assignment_expression
  init_abbreviations
  decl_specifier_prefix simple_abbreviation

/* A decl-abbreviation following a ptr_operator provokes a shift-reduce conflict for
 * const name
 * which is resolved in favour of the pointer, and implemented by providing versions
 * of decl-abbreviation guaranteed not to start with a cv_abbreviation.
 * decl-abbreviations are implemented type-centrically. That is the semantic constraint
 * that there must be a type exploited to impose structure, but actually eliminate
 * very little syntax. built-in types are multi-name and so need a different policy.
 * non-type decl-abbreviations are bound to the left-most type in a decl-abbreviation-seq,
 * by parsing from the right and attaching suffixes to the right-hand type. Finally
residual prefixes attach to the left.

```c++
/*
suffixBuiltInDeclSpecifier, raw: built_in_typeSpecifier
  suffixBuiltInDeclSpecifier, raw built_in_typeSpecifier
  suffixBuiltInDeclSpecifier, raw declSpecifier_suffix
  suffixBuiltInDeclSpecifier
  TEMPLATE suffixBuiltInDeclSpecifier
  scoped_id
  elaborate_typeSpecifier
  suffixNamedDeclSpecifier declSpecifier_suffix
  suffixNamedDeclSpecifier, raw
  suffixNamedDeclSpecifier, bi
  suffixNamedDeclSpecifier, raw suffixNamedDeclSpecifier, bi
  suffixNamedDeclSpecifier, bi
  suffixNamedDeclSpecifier, bi
  suffixNamedDeclSpecifier, bi
  suffixNamedDeclSpecifier, sf
  suffixNamedDeclSpecifier, sf
  suffixNamedDeclSpecifier
  suffixNamedDeclSpecifier, sf
  scopeSpecialFunction_id
  /* operators etc */
suffixNamedDeclSpecifications
  scope_specialFunction_id
  suffixNamedDeclSpecifications
  scope_specialFunction_id
  suffixNamedDeclSpecifications
  suffixBuiltInDeclSpecifier suffixNamedDeclSpecifications.sf
  suffixNamedDeclSpecifications.sf
  suffixNamedDeclSpecifications
  suffixBuiltInDeclSpecifier suffixNamedDeclSpecifications.scope
  suffixBuiltInDeclSpecifier scope

declSpecifier_affix:
  storage_classSpecifier
  functionSpecifier
  FRIEND
  TYPEDEF
  cv_qualifier

  /* The bogus conflict between public: as an anonymous bit-field and member-specification is resolved to the member-specification. */
  accessSpecifier
  #prec SHIFT THERE

  /* using-declaration is generalised to cover a much more general concept of re-use, so using treated like typedef.
   * Unfortunately this gives the same conflict on string as for linkage specification, so the #prec forces using followed
   * by a string to be treated as an include rather than a declaration. */
   USING
   #prec SHIFT THERE

declSpecifier_suffix:
  declSpecifier_affix
  AUTO

declSpecifier_prefix:
  declSpecifier_affix
  TEMPLATE declSpecifier_prefix

storage_classSpecifier:
  REGISTER | STATIC | MUTABLE | EXTERN | #prec SHIFT THERE
  '!' | STATIC

functionSpecifier:
  EXPLICIT
  #prec SHIFT THERE
  HIDE
  #prec SHIFT THERE
  HIDE
  HIDE

  /* Prefer linkage specification */
  /* Prefer INLINE / IMPLEMENTATION */
  /* Prefer VIRTUAL / PURE */

typeSpecifier:
  simple_typeSpecifier
```
elaborate_typeSpecifier:

cv_qualified

/* The following augment typeSpecifier rather than cv_qualified to avoid a conflict on | between
  * a * const b and a * const b which requires a 2-token lookahead to resolve. */
  
  "' CONST
  "' VOLATILE

elaborate_typeSpecifier: class_specifier
  | enum_specifier
  | \n
simple_typeSpecifier: elaborated_typeSpecifier
  | TEMPLATE elaborated_typeSpecifier
  | scoped_id
  | \n
built_in_typeSpecifier: BuiltInTypeSpecifier

/* The over-general use of declaration_expression to cover decl-specifier-seq.opt declarator in a function-definition means that
  * class X { }
  * could be a function-definition or a class-specifier.
  * enum X { }
  * could be a function-definition or an enum-specifier.
  * The function-definition is not syntactically valid so resolving the false conflict in favour of the
  * elaborated_typeSpecifier is correct.
  */

elaborated_typeSpecifier: elaborated_classSpecifier
  | elaborated_enumSpecifier
  | typename scoped_id

elaborated_enumSpecifier: ENUM scoped_id type prec SHIFT_THERE

enum_specifier: ENUM scoped_id enumerator_clause
  | ENUM enumerator_clause

enumerator_clause: "' enumerator_list_ecarb
  | "' enumerator_list enumerator_list_ecarb
  | "' enumerator_list '" enumerator_definition_ecarb

enumerator_list_ecarb: bang error '}'

enumerator_definition_ecarb: bang error '}'

enumerator_definition_filler: "' empty '}

enumerator_list: enumerator_definition_filler
  | enumerator_definition

enumerator_definition: enumerator_list ',' enumerator_definition_filler

enumerator: identifier

namespace_definition: NAMESPACE scoped_id compound_declaration

namespace_alias_definition: NAMESPACE compound_declaration

namespace_statement: NAMESPACE scoped_id '=' scoped_id '

namespace_declaration: NAMESPACE scoped_id ';'

using_directive: USING NAMESPACE scoped_id ','

asm_definition: ASM '" string '")

linkage_specification: EXTERN string looping_declaration
/* | EXTERN string compound_declaration -- covered by declaration */
/* A.7 Declarators */
/* init-declarator is named init_declarator to reflect the embedded decl-specifier-seq.opt */
init_declarations:
    assignment_expression , , init_declarator
    init_declarations , , init_declarator
    init_object_declarator , , init_declarator
    constructor_head , , bit_field_init_declarator
    constructor_head , , init_object_declarator
    assignment_expression

init_declarator:
    assignment_expression = ' initializer_clause
    assignment_expression = (' expression_list )
    bit_field_init_declarator
    init_object_declarator
    assignment_expression object_statements_clause
    bit_field_init_declarator object_statements_clause

/* declarator:
    /* direct_declarator: */
    star_ptr_operator:
    nested_ptr_operator:
    ptr_operator:
    ptr_operator_seq:
    /* Independently coded to localise the shift-reduce conflict: sharing just needs another $prec */
    ptr_operator_seq_opt:
        /* empty */
    ptr_operator ptr_operator_seq_opt
    cv_qualifier_seq_opt:
        /* empty */
    cv_qualifier:
    cv_qualifier:
    type_id:
    type_id:
    /* abstract_declarator: */
    abstract_declarator_opt:
    direct_abstract_declarator_opt:
        /* empty */
    direct_abstract_declarator
    direct_abstract_declarator:
        direct_abstract_declarator_opt parenthesis_clause
        direct_abstract_declarator_opt ( [ ] )
        direct_abstract_declarator_opt ( " constant_expression " )
        ' ( abstract_declarator )'
    /*
    parenthesis_clause: */
    parameters_clause cv_qualifier_seq_opt
    parameters_clause cv_qualifier_seq_opt exception_specification
parameters_clause: (' parameter_declaration_clause ')
  
    /* parameter_declaration_clause also covers init_declaration, type_id, declarator and abstract_declarator. */
  
    parameter_declaration_clause: ' empty ' 
    parameter_declaration_list ELLIPSIS 
    parameter_declaration_list ' , ' parameter_declaration 
    parameter_declaration_list: parameter_declaration 
      parameter_declaration_list ELLIPSIS 
      parameter_declaration_list ' , ' parameter_declaration 

    /* A typed abstract qualifier such as */
    /* Class * ... */
    /* looks like a multiply, so pointers are parsed as their binary operation equivalents that */
    /* ultimately terminate with a degenerate right hand term. */

    abstract_pointer_declaration: ptr_operator_seq
      multiplicative_expression star_ptr_operator ptr_operator_seq.opt
    
    abstract_parameter_declaration: abstract_pointer_declaration
      and_expression '&' abstract_pointer_declaration
    
    special_parameter_declaration: abstract_parameter_declaration
      abstract_parameter_declaration ' = ' assignment_expression
      ELLIPSIS
    
    parameter_declaration: assignment_expression
      special_parameter_declaration
      decl_specifier_prefix parameter_declaration 
    
    /* The grammar is repeated for use within template */
    /* */

    templatized_parameter_declaration: templatized_assignment_expression
      templatized_abstract_declaration
      templatized_abstract_declaration ' = ' templatized_assignment_expression
    
    templatized_abstract_declaration: decl_specifier_prefix templatized_parameter_declaration
      templatized_and_expression '&' templatized_and_expression ' & ' abstract_pointer_declaration
    
    /* function_definition includes constructor, destructor, implicit int definitions too. */
    /* A local destructor is successfully parsed as a function-declaration but the - was treated as a unary operator. */
    /* constructor_head is the prefix ambiguity between a constructor and a member-init-list starting with a bit-field. */
    /* */

    function_definition: ctor_definition
      func_definition
    
      assignment_expression function_try_block
    
      assignment_expression function_body
    
      decl_specifier_prefix func_definition
    
    ctor_definition: constructor_head function_try_block
      constructor_head function_body
    
      decl_specifier_prefix ctor_definition
    
    constructor_head: bit_field_init_declaration
      constructor_head ' ; ' assignment_expression
    
    function_try_block: TRY function_block handler_seq
    
    function_block: ctor_initializer.opt function_body
    
    function_body: compound_statement
    
    /* An = initializer looks like an extended assignment_expression. */
    /* An () initializer looks like a function call. */
/* initializer is therefore flattened into its generalised customers. */
"initializer" = 'expression_list'  
  assignment_expression  
  braced_initializer  
looping_initializer_clause = 'looping_initializer_clause'  
  "bang error"  
  "bang error"

/* A B Classes */

/* An anonymous bit-field declaration may look very like inheritance: */
  class A : B = 3;
  class A : B ;
  The two usages are too distant to try to create and enforce a common prefix so we have to resort to
  a parser hack by backtracking. Inheritance is much the most likely so we mark the input stream context
  and try to parse a base-clause. If we successfully reach a ( the base-clause is ok and inheritance was
  the correct choice so we unmark and continue. If we fail to find the ( an error token causes back-tracking
  to the alternative parse in elaborated_class_specifier which regenerates the : and declares unconditional success.
*/

/* Base class clause */
  colon_mark : class_key scoped_id %prec SHIFT THERE  
  class_key scoped_id colon_mark error  
  class_key : 'base_specifier_list' '{'  
  class_key '{' class_key scoped_id '{'  
  class_key '{' class_key : CLASS | STRUCT | UNION
  class_key : class_specifier_head nest declaration_seq.opt '{'  
  class_key : 'class_specifier_list' '{'  
  class_key '{' class_key '{'  
  class_key '{'  
  accessibilitySpecifier : accessSpecifier '{'  
  bit_field_declaration : assignment_expression '{' bit_field_width  
  bit_field_width : 'bit_field_width'  
  logical_or_expression '{' expression '{' assignment_expression -- has SA conflict w.r.t later = */
  logical_or_expression '{' expression '{' assignment_expression  
  logical_or_expression '{' expression '{' assignment_expression  
  bit_field_declaration '{' bit_field_width '{' bit_field_width  
  bit_field_declaration '{' bit_field_width '{' bit_field_width  

/* A S Derived classes */
  base_specifier_list : base_specifier

/* flattened */
baseSpecifier: baseSpecifierList, baseSpecifier
  scopedId
accessSpecifier baseSpecifier
  VIRTUAL baseSpecifier
  '!' VIRTUAL baseSpecifier
AUTO baseSpecifier
built_in_type_id
/
* A.10 Special member functions
/
conversionFunctionId: OPERATOR conversionType_id
conversionType_id: typeSpecifier ptr_operator_seq.opt
typeSpecifier conversionType_id
/
* Ctor-initialisers can look like a bit field declaration, given the generalisation of names:
  * Class(type) : m1(1), m2(2) ...
  * NonClass(bit_field) : int(2), second_variable, ...
  * The grammar below is used within a function_try_block or function_definition.
  * See simple_member_declaration for use in normal member function_definition.
/
cctorInitializer: /* empty */
cctorInitializer: 'm' memInitializer_list
memInitializer_list: memInitializer
memInitializer_list_head: memInitializer_list_head memInitializer
memInitializer_list_head: memInitializer_list bang error '
memInitializer: memInitializer_id '(' expression_list.opt ')
memInitializer_id: scopedId
/
* A.11 Overloading
operatorFunctionId: OPERATOR operator
/*
  It is not clear from the ANSI standard whether spaces are permitted in delete[]. If not then it can
  be recognised and returned as DELETE_ARRAY by the lexer. Assuming spaces are permitted there is an
  ambiguity created by the over generalised nature of expressions. operator new is a valid delator-id
  which we may have an undimensioned array of. Semantic rubbish, but syntactically valid. Since the
  array form is covered by the declarator consideration we can exclude the operator here. The need
  for a semantic rescue can be eliminated at the expense of a couple of shift-reduce conflicts by
  removing the comments on the next four lines.
/
operator: /*++++*/ NEW
          /*++++*/ DELETE
  /*---*/ NEW
  /*---*/ DELETE %prec SHIFT_THERE
  /*---*/ NEW '[' ']
  /*---*/ DELETE '[' ']
  /*---*/ '%"'
  /*---*/ '%"'
  /*---*/ '%"'
  /*---*/ '%"'
-- Covered by array of OPERATOR NEW */
-- Covered by array of OPERATOR DELETE */
```plaintext
/* A.12 Templates */

template_declarator: 
  template_parameter_clause declaration
  EXPORT template_declarator

/* This extension is only defined for USING, but we need to use declSpecifier_prefix to avoid conflicts. */

template_parameter_clause: 
  declSpecifier_prefix template_declarator
  TEMPLATE '<' template_parameter_list '>

template_parameter_list: 
  template_parameter_list ',' template_parameter

template_parameter: 
  template_parameter_list ',' template_parameter
  simple_type_parameter 
  simple_type_parameter '=' type_id
  template::template_type_parameter '=' identifier
  template::template_parameter_declaration

simple_type_parameter: 
  CLASS
  CLASS identifier
  TYPEID
  TYPEID identifier
  -- covered by parameter_declaration */

templated_type_parameter: 
  template::template_parameter_clause CLASS
  -- covered by parameter_declaration */
```

Page 332

March 1, 2002
template_id:
  <!-- covered by templated_parameter_declaration -->
  /* template-id is evaluated using a templated...expression so that > resolves to end of template. */
  /
  template_argument:
  template_id
  /
  template_argument_list:
    template_list '/' template
  /
  template_argument:  
    template_argument
    template_argument_list '/' template
  /
  template_name:  
    template
    template_list ' ' template
  /
  error
  /
  /* Generalised naming makes identifier a valid declaration, so TEMPLATE identifier is too. */
  /* The TEMPLATE prefix is therefore folded into all names, parenthesis_clause and declSpecifier_prefix. */
  /
  explicitInstantiation:  
    TEMPLATE declaration */
  /
  explicitSpecialization:  
    TEMPLATE ' ' declaration */
  /
  /* This extension is only defined for USING, but we need to use declSpecifier_prefix to avoid conflicts. */
  /
  declSpecifier_prefix explicit_specialization
  /
  /--------------------------
  / A.13 Exception Handling
  /--------------------------

try_block:  
  TRY compound_statement handler_seq

  function_try_block:  
    try_block
  /
  /* A handler_seq may follow a try_block in a compound_tree_statement such as: */
  /
  /* if (a) try ( ) catch(a) ( ) catch(b) ( ) catch(c) ( ) ... */
  /* we resolve the conflict by maximising the handler sequence. */
  /
  handler_seq:  
    handler
    handler handler_seq
  /
  handler:  
    handler_handler_seq
  /
  handler_handler_seq:  
    parameter_declaration
  /
  throw_expression:  
    THROW
  /
  templated_throw_expression:  
    THROW templated_assignment_expression
  /
  exception_specification:  
    THROW ' ( ' type_id_list ' , ' type_id
  /
  type_id_list:  
    type_id
    type_id_list '/ type_id

  /--------------------------
  / A.14 Tree literals
  /--------------------------

primary_tree_expression:  
  meta_scoped_id
  /
  primary_tree_expression:  
    '[' ' tree_expression ']
  /
  postfix_tree_expression:  
    postfix_tree_expression [ ' , ' ]
    postfix_tree_expression [ ' ( ' constant_expression ' ) ']
    postfix_tree_expression [ ' ( ' tree_argument_list_opt ' ) ']
    postfix_tree_expression [ ' , ' scoped_id]
    postfix_tree_expression ARROW scoped_id
tree_expression:  
  postffix_tree_expression
  **\* tree_expression**

  /* tree_argument_list.opf are carefully coded to avoid conflicts between the components of a constructor_head at the start of a function_definition
  * and the equivalent discrete elements. There is no need to resolve a conflict on ",", which is fortunate because it couldn't work. */

  tree_argument_list_opt:  
  tree_arguments.head
  tree_arguments.head ',', tree_argument_list_opt
  tree_argument.ctors
  constructor_head
  decl_specifier_prefix tree_argument.ctors
  tree_argument.head:  
  **\* empty *\*/
  tree_argument.most
  assignment_expression
  func_definition
  tree_argument.ctors,comma most
  constructor_head ',', tree_argument.most
  decl_specifier_prefix tree_argument.ctors,comma most
  tree_argument.most:
  terminated_tree_argument 
  terminated_tree_argument most
  terminated_tree_argument most ';'
  tree_argument.miss
  decl_specifier_prefix
  Assignement_expression
  /*
  bit_field_init_declaratin
  function_definition
  init_object_declaratin
  special_parameter_declaration
  decl_specifier_prefix assignment_expression
  decl_specifier_prefix tree_argument.miss
  */

  looping_unterminated_tree_argument:  
  start_search looped_unterminated_tree_argument
  ( end_search(); )

  looped_unterminated_tree_argument:  
  unterminated_tree_argument
  advance_search ' ' looped_unterminated_tree_argument
  advance_search ' ' */

  /* Omission of the preceding line which causes two reduce/reduce conflicts is justified provided the
  * looped_unterminated_tree_argument rules are only used within a compound_tree_statement, where the alternate
  * looping search for a tree_statement preceding this search. Since the cascaded advance_search ' '.
  * is only used to terminate a total failure of the search for a plausible template/arithmetic syntax, it doesn't
  * matter, apart from minor error reporting niceties, whether it is the statement or unterminated argument search
  * that is deemed to have failed.
  */

  looping_tree_statement:  
  start_search looped_tree_statement
  ( end_search(); )

  looped_tree_statement:  
  tree_statement
  advance_search ' ' looped_tree_statement
  advance_search ' ' */

  tree_statement:
  terminated_tree_argument
  unterminated_tree_argument ';'

  compound_tree_statement:
  ( ( tree_statement_seq_opt )
  ( 'tree_statement_seq_opt looping_unterminated_tree_argument ')' )
  ( 'tree_statement_seq_opt looping_unterminated_tree_argument '# bang error ') [ UNBANG('Bad compound-tree-statement.') ]; )

  ( 'tree_statement_seq_opt looping_tree_statement '# bang error ' )
/* Terminated syntax has an unambiguous end and does not need a ; as a meta-variable initializer. */
terminated_tree_argument:
  asm_definition
  compound_tree_statement
  declaration_statement ;
  explicit_instantiation
  expression_statement ;
  file_dependency_declaration
  file_placement_declaration
  include_declaration
  iteration_statement
  jump_statement
  labeled_statement
  linkage_specification
  namespace_alias_definition
  namespace_declaration
  parameter_declaration ;
  selection_statement
  template_declaration
  using_directive
  AUTOU meta_control_statement
  AUTO meta_expression_statement
  AUTO meta_function_declaration
  OPERATOR ;

/* Unterminated syntax has no obvious end and/or must have a ; as a meta-variable initializer. */
unterminated_tree_argument:
  unterminated_tree_argument.most

unterminated_tree_argument.most:
  simple_tree_declaration
  accessibilitySpecifier
  accessSpecifier
  baseSpecifier
  builtIn_typeSpecifier
  classSpecifier
  condition
  cvQualifier
  declSpecifier
  enumSpecifier
  enumerator_definition
  exception_declaration
  exceptionSpecification
  fileSpaceSpecifier
  function_definition
  function_try_block
  handler_seq
  initializer_clause
  mem_initializer
  AUTO meta_classSpecifier
  parameter_declaration

  [ UNBANG("Bad compound-tree-statement."); ]
  [ UNBANG("Bad tree-statement."); ]

  /* empty */
  tree_statement_seq.opt:
  tree_statement_seq.opt looping_tree_statement
  tree_statement_seq.opt looping_tree_statement ;
  /* bang error */
  [ UNBANG("Bad compound-tree-statement."); ]
  [ UNBANG("Bad tree-statement."); ]
```
/*
  simple_tree_declaration  -- not part of .most */
  simple_type_parameter
  storage_class_specifier
  template_argument
  template_parameter
  try_block
  type_id
  type_parameter
  reserved_id
  token.punct

  AUTO
  CATCH
  CLASS
  DO
  ENUM
  NAMESPACE
  OPERATOR
  STRUCT

  TEMPLATE
  THROW
  TYPE_NAME

  UNION

  reserved_id:
  ASCII | BREAK | CASE | CONST_CAST | CONTINUE | DEFAULT | DELETE | DYNAMIC_CAST | ELSE | FOR
  GOTO | IF | NEW | REINTERPRET_CAST | RETURN | sizeof | STATIC_CAST | SWITCH | TRY | TYPEID | WHILE

  EXPORT

  token.punct:
  SCOPE | SHR | SHR | EQ | NE | LE | GE | LOG_AND | LOG_OR | INC | DEC | ARROW | ARROW_STAR | DOT_STAR

  simple_tree_declaration:
  declSpecifierPrefix
  init_declarations
  constructor_head_1, assignment_expression
  init_declarations
  special_parameter_declaration
  declSpecifierPrefix simple_tree_declaration

/*
  * A.15 Object statements
  *-----------------------------------------------*/

  object_statements_clause:
    '{' '(' object_statement_seq_opt ')' '}'
    '{' '(' object_statement_seq_opt_opt looping_object_statement '}'
    { UNBANG("Bad object-statements-clause.");}

  object_statement_seq_opt:
    /* empty */
    object_statement_seq_opt opt looping_object_statement
    object_statement_seq_opt opt looping_object_statement /* bang error */
    { UNBANG("Bad object-statement."); }

  looping_object_statement:
    start_search looped_object_statement
  object_statement:
    advance_search ++ looped_object_statement
```
object_statement:
  advance_search '
  
  function_used_block
  
  initializer_clause');
  expression_list'));
file_dependency_declaration
file_placement_declaration
filespace_specifier semi
meta_control_statement
AUTO meta_control_statement
AUTO meta_expression_statement
AUTO meta_function_definition
derived_clause object_statement
derived_clause : ('object_statement_seq.opt')
function_block
function_try_block
ctor_initializer ;'
USING file_id_list function_used_block
segment function_used_block

/* A.16 Derivation rules

* A 17.1 meta-names
* id
* meta_id
* meta_simple_type
* AUTO
* meta_id SCOPE
* meta_nested_id
* meta_id ;' meta_id
* meta_scope meta_nested_id
* global_scope meta_nested_id
* A 17.2 meta-classes
* meta_class_specifier:
* meta_class_key meta_nested_id compound_declaration
* meta_class_key meta_nested_id : ' base_specifier_list compound_declaration
* A 17.3 meta-types
* The MetaType names are not reserved words so form part of identifier and consequently scoped_id */
/* The %prec maximises the length of e.g. unsigned int when followed by e.g int::s */
meta_class_key:
  class_key
NAMESPACE
/* A 17.4 meta-variables */

/* A 17.5 meta-functions, meta-constructors and meta-destructors */

/* A 17.6 meta-statements */

/* A 17.7 meta-expressions */
meta_postfix_expression '([' expression '])' object_statements_clause_semi
meta_logical_or_expression assignment_operator line tree_statement
CONST meta_expression_statement
STATIC meta_expression_statement
'.' STATIC meta_expression_statement

/A 18 Syntax macros

syntax_macro_definition: EXPLICIT AUTO meta_type identifier '{' syntax_macro_parameter_list '}' compound_tree_statement
EXPLICIT AUTO meta_type identifier '{' syntax_macro_parameter_list '}' '[' ']' compound_tree_statement
EXPLICIT AUTO meta_type identifier '{' '}' compound_tree_statement
EXPLICIT AUTO meta_type identifier '{' '{' '}' compound_tree_statement

syntax_macro_parameter_list: syntax_macro_parameter syntax_macro_parameter_list ',' syntax_macro_parameter
syntax_macro_parameter: syntax_macro_parameter
meta_type_identifier meta_type_identifier '{' '}'
identifier reserved_id
token.punct
':'
';'
'{'
'}'

beng error

{ UNBANG("bad syntax-macro-parameter."); }

/A 19 files

include_declaration: USING string
USING '//' INCLUDE string_expr
USING '//' INCLUDE '//' utility string_expr
USING '//' utility string_expr
ENTITY
FROZEN
POOL
UTILITY

file_dependency_declaration: using_implementation_semi
using_interface_semi
using_implementation: using_implementation_semi
USING '//' IMPLEMENTATION file_use
USING '//' IMPLEMENTATION '==' file_use
using_interface: using_interface_semi
USING '//' INTERFACE file_use
USING '//' INTERFACE '==' file_use
file_use: file_id
file_entity

file_placement_declaration: export_implementation_semi
export_interface semi
EXPORT /* NOIMPLEMENTATION semi
EXPORT /* UTILITY semi
export_implementation:
EXPORT /* IMPLEMENTATION implementation_file
EXPORT /* IMPLEMENTATION = '{' implementation_file
export_interface:
EXPORT /* INTERFACE interface_file
EXPORT /* INTERFACE '{' interface_file
implementation_file:
file_id
template
interface_file:
file_id
template
file_name:
string
file_name /* INTERFACE
file_name /* IMPLEMENTATION
file_name /* TEMPLATE
file_name /* utility
file_name /* GUARD = '{' string Expr
file_name /* NOGUARD
file_name /* PATH = '{' string Expr
file_name /* PREFIX = '{' string Expr
file_name /* SUFFIX = '{' string Expr
declarator_id
elaborated_typeSpecifier
NAMESPACE scoped_id
file_id:
file_name
file_entity /* IMPLEMENTATION
file_entity /* INTERFACE
file_id_list:
file_id
file_id_list '{' file_id
namespaceSpecifier:
NAMESPACE /* FILE file_name compound_declaration

// Error handling aids
-------------

scarr:
'}' /*
  bang_error ')' /*
  bang_error ')' /*
  ( UNBang("Extra text ignored before '}'")); }

semi:
'}' /*
  bang_error ')' /*
  ( UNBang("Missing '}'")); }

// Back-tracking and context support
-------------

advance_search:
error ( yyerrtok; yyclearin; advance_search(); ) /* Rewind and queue '+' or '-' '+' */
line: /* empty */ /* Get current line context */
mark: /* empty */ /* Push lookahead and input token stream context onto a stack */
nest: /* empty */ /* Push a declaration nesting depth onto the parse stack */
start_search: /* empty */ /* Create/reset binary search context */
start_search1: /* empty */ /* Create/reset binary search context */
util: /* empty */ /* Get current utility mode */
D Command Line

The FOG command line is

```
  fog <tokens> <files>
```

where <files> is one or more input files, conventionally using the extension .fog.
- may be used to indicate that the standard input be used as an input file.
- <tokens> is any combination of the following, with or without spacing between a
token such as -I and a subsequent text argument denoted as *

D.1 Miscellaneous options

- `help` Display usage help.
- `q` Suppress the program identification message.

D.2 Preprocessor options

- `-D*` Define a preprocessor macro value.
- `-I*` Source include file path(s) (defaults to current directory).

D.3 Variant C++ options

- `-long_long_type` Treat long long as built-in (Sun C++ language extension).
- `-mbc#` Bytes in a multi-byte character(4).
- `-no_access` Diagnose access declarations (ARM C++ compatibility).
- `-no_bool_type` Do not treat bool as built-in (ARM C++ compatibility).
- `-no_namespace` Treat namespace as a synonym for class.
- `-no_specialisation_prefix` Do not require template<> for specialisation (ARM C++ compatibility).
- `-no_using` Emit using-declarations as access-declarations (ARM C++ compatibility).
- `-no_wchar_t_type` Do not treat wchar_t as built-in (ARM C++ compatibility).

D.4 Generated C++ options

- `-anony_prefix*` Prefix for "anonymous" names (default is _anony_).
- `-c++` Behave more like a C++ compiler.
- `-comment_line_numbers` Enclose #line numbers as comments in emitted files.
- `-extern_prefix*` Prefix for extern linkage names (default is _extern_).
- `-no_line_numbers` Omit #line numbers from emitted files.
- `-nobanner` Suppress emitted comment banners (to ease regression testing).
- `-t#` Columns per tab in source files (default 8).
- `-template_parameter_prefix` Prefix for normalised template parameter names (default is _“__”_).
- `-unnest` Emit nested classes after rather than within enclosing class.

D.5 Output file options

- `-cd*` Emitted implementation file directory path.
- `-cp*` Emitted implementation file prefix.
- `-cs*` Emitted implementation file suffix.
- `-ctd*` Emitted template implementation file directory path.
- `-ctp*` Emitted template implementation file prefix.
-cts*  Emitted template implementation file suffix.
-f      Force file emission (bypass redundancy comparisons).
-global* (File)name of the global namespace.
-hd*    Emitted interface file directory path.
-hp*    Emitted interface file prefix.
-hs*    Emitted interface file suffix.
-htd*   Emitted template interface file directory path.
-htp*   Emitted template interface file prefix.
-hts*   Emitted template interface file suffix.
-log*   Log file name (duplicates standard error).
-max_errors# maximum number of errors before program termination (100).
-max_warnings# maximum number of warnings before program termination (0).
-nc     Notify emitted file names that are created.
-ne     Notify emitted file names that are suppressed through equivalence.
-o*     File name for make dependencies between source and emitted files.

D.6 Diagnostic options

-readonly Just read source files to gather token count statistics.
-statistics Emit program performance statistics.
-z2h    Display each token passed between lexer and hash parser.
-z2l    Display each token passed between lexer and locate parser.
-z2m    Display each token passed between lexer and main parser.
-z2r    Display each token passed between lexer and replace parser.
-z2s    Display each token passed between lexer and substitute parser.
-za     Display changes to the activity status of declarations.
-zd     Delete all objects rigorously on exit (for testing with purify).
-zf     Display file name as each entity is (re)positioned.
-zi     Display each input and macro line.
-zl     Display the behaviour of the lexer.
-zp     Display changes to the purity status.
-zs     Display changes to the default parser scope.
-zt     Display each token passed between lexer and main parser.
-zu     Display changes to the composed entity and parser default utility.
-zx     Display full hex address of each object in diagnostics.
-zy     Display yacc parser progress.
-zz     Repeat certain failed invocations after generating an error to aid debug.

D.7 Predefined macros

#define __STDC__ 0
#define __cplusplus 0
E  Built-In Functionality

This appendix describes the built-in functionality, or more accurately the potential built-in functionality of FOG, since only about half of what is described has actually been implemented and because usage of FOG for a variety of practical applications will probably reveal requirements for further built-in support.

E.1  Built-in Meta-classes

E.1.1  auto

All meta-classes (and meta-namespaces) ultimately inherit from the `::auto` metaclass, which has no functionality. Its positioning at the root enables meta-program code to affect all classes by composition with its initially empty meta-constructor or meta-destructor. For instance:

```cpp
auto auto::-auto()
{
    std::diagnostic(©This);
}
```

generates a diagnostic message for every meta-class in an application, since the meta-destructor of the root meta-class is inherited by and consequently executed during meta-destruction of every meta-class including those for built-in types.

E.2  Built-in Meta-namespaces

E.2.1  std

A variety of generic support facilities are provided by built-in meta-functions. These built-in functions are incorporated as part of the `std` meta-namespace. This avoids cluttering the global meta-namespace directly, or indirectly through introduction of a new namespace. The `std` meta-namespace already exists and is otherwise empty since C++ reserves `std` for language support but has no meta-functionality.

```cpp
auto bool std::ambiguous(expression aName)
auto bool std::defined(expression aName)
```

These two meta-functions test for the presence and multi-presence of declarations. They return non-zero if the expression is ambiguous (has multiple definitions) or is defined (has at least one definition).

These meta-functions may be used as predicates to avoid errors in subsequent code. They take no account of whether a declaration is enabled or not. Thus a declaration for use only in leaf classes is regarded as defined at the root class and all its derived classes, even though the declaration is disabled at non-leaves.

```cpp
auto token std::find(expression aName) []
```

Returns a possibly empty list of all declarations visible in the meta-name-space

```cpp
auto void std::diagnostic(string aString)
auto void std::error(string aString)
auto void std::warning(string aString)
```

These meta-functions provide the only method for communication between a meta-program and the programmer. The string argument is emitted to standard out (and any log-file) classified as either a diagnostic warning or error message. Warning messages are prefixed by "WARNING --" and increment the overall warning count.

Error messages are prefixed by "ERROR --" and increment the overall error count. Obviously an extension to support meta-streams and `std::cerr` would be more powerful.
auto string std::get_cpp(string aString)
An almost universally available preprocessor extension is the ability to pass macro
definitions with an invoking command line such as
    cc -DDEBUG_LEVEL=4 ...
FOG also supports -D as a command line option (see Appendix D).
FOG provides access to the preprocessor definition namespace via the get_cpp
built-in meta-function, which takes the name to be looked up in the Cpp names­
space as an argument and returns its value.
    auto int debugLevel = std::get_cpp("DEBUG_LEVEL");
    auto if (debugLevel > 4 ) /* ... */;
auto string std::get_env(string aString)
Definitions may be acquired from the programming environment by using
std::get_env, which just invokes the POSIX getenv routine.
    auto string logName = std::get_env("LOGNAME");
auto string std::date()
auto string std::file()
auto string std::time()
These functions return the current date, file and time and replace the ANSI C
preprocessor symbols __DATE__, __FILE__ and __TIME__.
auto token std::parse(string aString)
auto token std::parse_tokens(token someTokens[])
auto token std::tokenize(string aString) []
The parse meta-function provides necessary support for character- and token­
level substitution. It performs lexical and syntactical analysis of aString to return
the equivalent syntax tree.
parse is equivalent to successive calls of tokenize and then parse_tokens, to
perform lexical and syntactic analysis respectively. It is not clear how much, if any
of this functionality is necessary or even desirable.

E.3 Built-in Meta-variables

The following meta-variables are built-in (to the token meta-type and so inherited
by all meta-types).

Namespace
Identifies the current namespace, which is necessary to ensure that declarations
occurring nominally within one class can be rescoped to be placed elsewhere.

class ThisClass
{
    class NestedClass {};
    class {Namespace}::SiblingClass {};
};

This works whether ThisClass is a class in the traditional unnamed global
namespace or a class in a named namespace.

When invoked directly within a namespace, Namespace identifies the namespace,
not its enclosing namespace.

OuterNamespace

OuterNamespace differs from Namespace when invoked for a namespace by
returning the immediately enclosing namespace, if there is one or the global
namespace otherwise.

Thus ${Namespace}::$({OuterNamespace}) first locates the current namespace
and then locates its immediately enclosing namespace. Eventually after sufficient
iterations OuterNamespace always returns the global namespace.
It is an error to traverse potential declarations beyond their defined ancestry.

**OuterScope**

Similarly **OuterScope** differs from **Scope** when invoked for a scope (class, struct, union) by returning the immediately enclosing scope, if there is one or the namespace otherwise.

Thus `$({Scope})::{OuterScope}` first locates the current scope and then locates its immediately enclosing scope. Eventually after sufficient iterations **OuterScope** always returns the namespace.

It is an error to traverse potential declarations beyond their defined ancestry.

**Scope**

Identifies the current scope, which may be a class, filespace, linkage, namespace, struct or union.

When invoked directly within a scope, **Scope** identifies the scope, not its enclosing scope.

**Super**

**Super** identifies the primary base class. It is a short form for (and much more efficient than) **Scope**::`bases()`[0].

Use of **Super** for a class without a base-class is an error. The base-class determined by **Super** ignores base classes declared using `auto`, and so is not exactly equivalent to **Scope**::`bases()`[0], which could resolve a base meta-class.

```cpp
class Y : auto XI, public X2
{
  // Super is X2
  // bases()[0] is X1
};
```

**This**

**This** identifies the current declarative region which is the same as **Scope**, when invoked within the context of a class or namespace. However, when invoked within the context of a variable or function, **This** refers to the variable or function, and provides access to object-scoped meta-declarations in preference to occluded meta-declarations from the class scope.

### E.4 Built-in Meta-functions

The inheritance relationships between the built-in meta-types are described in Section 4.1.2 and shown below using indentation.

```cpp
token
  expression
    character
    number
      double
        signed
        unsigned
    bool
    string
```
token
expression
name
keyword
decl_specifier
cv_qualifier
identifier
meta_type
class_key
reserved
using_directive
initializer_clause
template_argument
token
declaration/statement
expression_statement
specifier
base_specifier
enumerator_definition
file_dependency_specifier
file_placement_specifier
namespace_alias_definition
object_specifier
function_specifier
meta_function_specifier
meta_parameter_specifier
parameter_specifier
template_parameter_specifier
template_type_parameter_specifier
type_parameter_specifier
value_parameter_specifier
typedef_specifier
using_declaration
variable_specifier
scope_specifier
filesystem_specifier
linkage_specification
meta_class_specifier
namespace_definition
type_specifier
built_in_type_specifier
class_specifier
elaborated_type_specifier
enum_specifier
token
entity
base
enumerator
namespace_alias
object
exception
function
meta_function
meta_parameter
meta_variable
parameter
typedef
using
variable
token
  entity
    scope
      filespace
      linkage
      meta_class
      namespace
      type
        built_in
        class
        enum
        struct
        typename
        union

token
  object_statement
  exception_specification
  handler
  iterator
  modifier
    array_modifier
    function_modifier
    pointer_modifier
    reference_modifier
    scoped_modifier
  void
  punctuation
  tree_literal

All meta-types inherit from token, where
  • polymorphism between scalars and lists is established
  • default implementations of all built-in functions provided
  • meta-type type testing predicates are implemented

The built-in meta-functions are described in the following sections, with the
description at the lowest level in the inheritance hierarchy at which use of the
meta-function is meaningful. Use below that level for a predicate testing meta­
function just returns false. For other meta-functions an error message is
generated.

E.4.1 array_modifier

E.4.2 base and baseSpecifier

auto class base::base()
auto class baseSpecifier baseSpecifier::base()
auto bool base::is_auto()
auto bool baseSpecifier::is_auto()
auto bool base::is_private()
auto bool baseSpecifier::is_private()
auto bool base::is_protected()
auto bool baseSpecifier::is_public()
auto bool base::is_public()
auto bool baseSpecifier::is_public()
auto bool base::is_virtual()
auto bool baseSpecifier::is_virtual()
E.4.3  
**built_in and built_in_type_specifier**

E.4.4  
**character**

```cpp
auto character::operator character()  
auto character::operator identifier()  
auto character::operator number()  
auto character::operator string()
```

E.4.5  
**class and class_specifier**

E.4.6  
**class_key**

E.4.7  
**cv_qualifier**

E.4.8  
**decl_specifier**

E.4.9  
**declaration**

see statement

E.4.10  
**elaborated_type_specifier**

```cpp
auto class_key elaborated_type_specifier::class_key() []  
auto bool elaborated_type_specifier::is_auto()  
auto bool elaborated_type_specifier::is_class()  
auto bool elaborated_type_specifier::is_namespace()  
auto bool elaborated_type_specifier::is_struct()  
auto bool elaborated_type_specifier::is_typename()  
auto bool elaborated_type_specifier::is_union()
```

E.4.11  
**entity and specifier**

```cpp
auto string entity::implementation_file()  
auto string specifier::implementation_file()  
auto string entity::interface_file()  
auto string specifier::interface_file()
```

E.4.12  
**enum and enum_specifier**

```cpp
auto enumerator enum::enumerators() []  
auto enumerator_definition enum_specifier::enumerators() []
```

E.4.13  
**enumerator and enumerator_definition**

```cpp
auto number enumerator::value()  
auto expression enumerator_definition::value()
```

E.4.14  
**exception and exception_declaration**

E.4.15  
**exception_specification**

E.4.16  
**expression**

```cpp
auto expression expression::value()
```
E.4.17  filespace and filespaceSpecifier

E.4.18  function, functionModifier and functionSpecifier

auto exception function::exceptions() []
auto exception_declaration function_modifier::exceptions() []
auto exception_declaration function specifier::exceptions() []
auto parameter function::parameters() []
auto parameter_declaration function_modifier::parameters() []
auto parameter_specifier function specifier::parameters() []
auto function specifier function::signature() []
auto function specifier function_modifier::signature() []
auto function specifier function specifier::signature() []

E.4.19  handler

E.4.20  Identifier

auto identifier::operator character()
auto identifier::operator identifier()
auto identifier::operator number()
returns the result of a text to numeric conversion, which usually involves an error message.
auto identifier::operator string()

E.4.21  iterator

A typical idiomatic use of an iterator is shown in the following example

    auto for (iterator i = $MyClass::variables(); i; ++i)
        auto if (i->is_static())
            const char *MyClass::names[] = { "$i->name()" };
            const char *MyClass::names[] = { 0 };
    in which composition of array elements is used to build a null terminated list of the names of the member variables of MyClass.

    Note that an iterator maintains a copy of the identities of the elements of the iteration domain, but not of their contents. Therefore addition of an addition base class during a traversal of base classes will not be detected by the iteration.
    Modification of an iteration element prior to traversal does affect the iteration.

    auto iterator::iterator()
        constructs an iterator already out-of-domain.
    auto iterator::iterator(token [])
        constructs an iterator to iterate over and from the start of the exposed list. The identities of the elements in the list are copied.
    auto iterator::iterator(iterator)
        constructs a copy of an iterator, which involves a copy of the identities of the elements in the iteration domain and of the position within the domain.
    auto void iterator::operator=(token [])
        assigns an iterator to iterate over and from the start of the exposed list. The identities of the elements in the list are copied.
    auto void iterator::operator=(iterator)
        assigns a copy of an iterator, which involves a copy of the identities of the elements in the iteration domain and of the position within the domain.
    auto iterator::operator number()
        returns true as long as the iterator remains within the iteration domain.
auto void iterator::operator++() advances the iterator through the iteration domain, setting out-of-domain once the top edge passed.

auto void iterator::operator--() rewinds an iterator one step back through the iteration domain, setting out-of-domain once the bottom edge passed.

auto token iterator::operator->() returns the current element in the iteration domain, generating an error if out of domain.

auto token iterator::operator*() returns the current element in the iteration domain, generating an error if out of domain.

E.4.22 keyword

E.4.23 linkage and linkage_specification

auto string linkage::value()
auto string linkage_specification::value()

E.4.24 meta_class and meta_class_specifier

E.4.25 meta_function and meta_function_specifier

auto meta_parameter meta_function::meta_parameters() []
auto meta_parameterSpecifier
meta_function_specifier::meta_parameters() []

E.4.26 meta_parameter and meta_parameter_specifier

E.4.27 meta_type

E.4.28 meta_variable and meta_variable_specifier

auto token meta_variable::value()
auto token meta_variable_specifier::value()

E.4.29 modifier

auto bool modifier::is_array_modifier()
auto bool modifier::is_const()
auto bool modifier::is_function_modifier()
auto bool modifier::is_pointer_modifier()
auto bool modifier::is_reference_modifier()
auto bool modifier::is_scoped_modifier()
auto bool modifier::is_volatile()

E.4.30 name

auto name name::full_name()
auto name name::name()

E.4.31 namespace and namespace_definition

E.4.32 namespace_alias and namespace_alias_definition

auto namespace namespace_alias::value()
auto name namespace_alias_definition::value()
E.4.33 number

auto number::operator character()
auto number::operator identifier()
auto number::operator number()
auto number::operator string()

E.4.34 object and object_specifier

auto bool object::is_const()
auto bool object_specifier::is_const()
auto bool object::is_static()
auto bool object_specifier::is_static()
auto bool object::is_volatile()
auto bool object_specifier::is_volatile()
auto meta_function object::meta_functions() []
auto meta_function object_specifier::meta_functions() []
  returns the immediate list of member meta-functions
auto meta_variable object::meta_variables() []
auto meta_variable object_specifier::meta_variables() []
  returns the immediate list of member meta-variables
auto modifier object::modifiers() []
auto modifier object_specifier::modifiers() []
  returns the list of declarator modifiers
auto type object::type()
auto type object_specifier::type()

E.4.35 object_statement

auto bool object_statement::is_boundary()
auto bool object_statement::is_leaf()
auto bool object_statement::is_pure()
auto bool object_statement::is_root()

E.4.36 parameter and parameter_specifier

E.4.37 pointer_modifier

E.4.38 punctuation

E.4.39 reference_modifier

E.4.40 reserved

E.4.41 scope and scope_specifier

auto base scope::all_bases() []
auto base_specifier scope_specifier::all_bases() []
  returns the transitive list of base-specifiers (including meta-bases)
auto scope scope::all_classes() []
auto scope scope_specifier::all_classes() []
  returns the transitive list of nested classes
auto function scope::all_functions() []
auto function scope_specifier::all_functions() []
  returns the transitive list of member functions
auto type scope::all_types() []
auto type scope_specifier::all_types() []
  returns the transitive list of member types
auto typedef scope::all_typedefs() []
auto typedef_specifier scope_specifier::all_typedefs() []
returns the transitive list of member typedefs
auto variable scope::all_variables() []
auto variable_specifier scope_specifier::all_variables() []
returns the transitive list of member variables
auto base scope::bases() []
auto base_specifier scope_specifier::bases() []
returns the immediate list of base-specifiers (including meta-bases)
auto class_key scope::class_key() []
auto class_key scope_specifier::class_key() []
auto scope scope::classes() []
auto scope_specifier scope_specifier::classes() []
returns the immediate list of nested classes
auto entity scope::friends() []
auto specifier scope_specifier::friends() []
auto function scope::functions() []
auto function_specifier scope_specifier::functions() []
returns the immediate list of member functions
auto bool scope::is_auto() []
auto bool scope_specifier::is_auto() []
auto bool scope::is_boundary() []
returns true if there is a pure-virtual function in an immediate base class but no
pure-virtual in this class
auto bool scope::is_class() []
auto bool scope_specifier::is_class() []
auto bool scope::is_leaf() []
returns true if there are no derived classes
auto bool scope::is_pure() []
returns true if there is a pure-virtual function
auto bool scope::is_struct() []
auto bool scope_specifier::is_struct() []
auto bool scope::is_union() []
auto bool scope_specifier::is_union() []
auto typedef scope::typedefs() []
auto typedef_specifier scope_specifier::typedefs() []
returns the immediate list of member typedefs
auto type scope::types() []
auto type_specifier scope_specifier::types() []
returns the immediate list of member types
auto variable scope::variables() []
auto variable_specifier scope_specifier::variables() []
returns the immediate list of member variables

E.4.42 scoped_modifier
E.4.43 specifier
E.4.44 statement
E.4.45 string

Identical strings are represented by the same metaobject, so content comparison
is performed by operator== and operator!=.
auto string::operator character()
auto string::operator identifier()
auto string::operator number()
auto string::operator string()

E.4.46  template_parameter and template_parameter_specifier

E.4.47  token

The token meta-type provides a default implementation for all built-in meta­
functions. Predicate meta-functions such as is_virtual() that return a bool
value are implemented to return false. All other meta-functions generate an error
message.

auto entity token::find_entity(expression anExpression) []
returns all visible entities (names or types)

auto object token::find_name(expression anExpression) []
returns all visible names

auto type token::find_type(expression anExpression) []
returns all visible types

auto token token::get(expression anExpression)
returns the meta-declaration visible in the meta-name-space, generating an error
if undefined or ambiguous.

auto entity token::get_entity(expression anExpression)
returns the visible entity (name or type), generating an error if undefined or
ambiguous.

auto object token::get_name(expression anExpression)
returns the visible name, generating an error if undefined or ambiguous.

auto type token::get_type(expression anExpression)
returns the visible type, generating an error if undefined or ambiguous.

auto bool token::is_exposed()
returns true if token comprises an exposed list.

auto bool token::is_list()
returns true if token comprises an encapsulated or exposed list.

auto bool token::is_meta_type()
returns true if token is at least as specialised as meta_type.

auto unsigned token::length()
returns the number of elements in an encapsulated or exposed list.

auto meta_type token::meta_type()
returns the describing meta-type.

auto token token::sub_list(unsigned start, signed length) []
returns an exposed list comprising the abs(length) elements from start to
start+length exclusive, generating an error if any element out-of range.

auto token token::operator[](unsigned index)
returns the index element, generating an error if out-of range.

auto token token::operator+(token tokens[]) []
return a new list comprising tokens appended to this list.

auto void token::operator+=(token tokens[]) []
appends tokens.
E.4.48 type and type_specifier

E.4.49 typedef and typedef_specifier

auto type typedef::value()
auto type_specifier typedef_specifier::value()

E.4.50 typename

E.4.51 using and using_declaration

E.4.52 using_directive

E.4.53 variable and variable_specifier

auto expression variable::value()
auto expression variable_specifier::value()

E.4.54 void

The void meta-type is used for invalid and zero values.
Implementation

The presentations of the FOG extensions to C++ in Chapter 3, their semantics in Chapter 4, and a novel parsing approach in Chapter 5 are all fairly substantial and so a number secondary issues are relegated to this appendix.

A brief discussion of the difficulties of enhancing C++ syntax is followed by a description of some syntax extensions that were considered and why they were not implemented.

We then describe how the superset grammar approach resolves specific C++ parsing difficulties and outline the activities needed during the semantic processing necessary to recover lost syntactic resolution.

Finally the syntax extensions to support file placement and include file dependencies are presented.

F.1 Syntax Implementation

Providing additional syntax in C++ without introducing new reserved words or totally esoteric meanings for punctuation is rather difficult, since most simple syntax using non-reserved words is covered by a simple-declaration.

The multi-pass implementation used set rather than export, and use rather than using for the file syntaxes. This caused no ambiguity at the declaration level since / cannot appear except as an initializer in a declaration.

    set / implementation = "file";

The above of course is a syntactically valid expression, and a little provocative given the presence of set as a template name in the Standard Template Library.

Migration to the superset grammar resulted in a generalised parser being used for declarations and so set became difficult to disambiguate syntactically. The syntax was therefore changed to its current form. export and using are not so very far away from the intended meanings of specify-output-file, and specify-input-dependency.

Overloading reserved words is undesirable and confusing as exemplified by the many meanings of static. The reuse of auto to mean meta is equally unsatisfactory. Other new over loadings are relatively clear, since the reserved word is followed by a switch.

F.1.1 !const and !volatile

Provision of !static for more explicit control of composition suggests that !const and !volatile should also be provided. However the situation is not quite the same. const and volatile form part of a function signature and so there is no possibility that composition should ever interpret a missing const as const. A missing const always means !const. This reasoning makes !const unnecessary but does not preclude its provision as a documentation aid.

Provision of !const causes implementation problems too, since an ambiguity arises in a generalised parse between

    (type) ! a    // Cast of complement
    (type) ! const // Very degenerate parameter-declaration-clause and cv-qualifiers

Two tokens of lookahead are required to resolve the ambiguity.

Supporting !const as an extension of type-specifier rather than cv-qualifier solves this problem allowing usage everywhere except following parameter-declaration-clauses. However this support was a little irregular and since the sole purpose was as a documentation aid, it was decided to omit !const and !volatile.
F.1.2 Member variable delegation

Larger objects may be built from smaller objects using inheritance or aggregation. Inheritance has the convenient property that the entire interface of the base object is visible as part of the larger object, whereas aggregation makes none of the interface available. The implementor is faced with an all or nothing choice for delegation.

A *re-using-declaration* that mentions a member-variable could be interpreted as a directive to support delegation so that:

```cpp
class Proxy
{
    Client *_member_variable;
    using _member_variable;
};
```

automatically synthesises delegation routines such as

```cpp
int f(double b) { return _member_variable->f(b); }
```

for every accessible function of `_member_variable`. More selective synthesis could be achieved by naming functions

```cpp
using _member_variable->f;
```

A further extension was considered whereby the client could group a number of functions to establish a view:

```cpp
class Client
{
    namespace/view ProxyView
    {
        int f(double b);
    }
};
```

so that all functions identified as part of the `ProxyView` would automatically be delegated by:

```cpp
using _member_variable->ProxyView;
```

This extension then ensures that addition of a further function to `Client::ProxyView` automatically adds a delegating function to the Proxy.

This is useful, but vulnerable to practical considerations:

- it may be desirable to handle null tests in the delegation routines
- it may be necessary to add *this as an extra argument during delegation

Customized formatting of the synthesised routine is not easily handled by a standardised approach. Customized formatting is available via meta-programming and so the concept of using a member variable and delimiting part of an interface through a view is no longer supported. The effect can be achieved by:

```cpp
class Client
{
    int f(double b);
    int g(double b);
    void g();
    auto declaration ProxyView[] = { f, g(double) };
};
class Proxy : auto Client
{
    Client *_member_variable;
};
auto Proxy::Proxy()
{
    $delegate(_member_variable, Client::ProxyView);
};

An appropriate implementation of delegate can then be written or accessed from a meta-library to iterate over the declarations in Client::ProxyView to synthesize delegation routines via _member_variable.

[The meta-inheritance of Proxy from Client establishes a meta-compilation order dependency so that meta-construction of Client occurs before Proxy, ensuring that any member-functions declared by Client's meta-constructor exist before Proxy's meta-constructor synthesises its delegation functions.]

This example is far from bomb-proof; functions added to Client during meta-main execution or by later meta-constructors will not receive delegate treatment. An implementation proof against arbitrary composition requires Client to provide a meta-function to register contributions to ProxyView, and to register classes interested in viewing the contents of ProxyView. The registration function can notify registered viewers when any change occurs, and the meta-destructor for the client can verify that no functions have appeared without passing through the registration function. This approach is reliable but slow and it is readily supported by standard meta-functions from a meta-library. Alternative metaobject protocols such as those available with CLOS or OpenC++ provide more direct hooks. FOG could support the option for a user-definable meta-function:

    auto declaration Client::add_function(declaration);

which would be invoked for each addition and return a possibly modified declaration or even a completely different set of declarations. The modest complexity and inefficiency of detecting, maintaining and invoking such functions does not seem to be justified for simple applications.

F.1.3 Pattern names

AspectJ [Lopes98] supports the addition of code to all functions whose name matches some pattern, which is very useful for adding tracing code, since an "entering function x" diagnostic is easily attached to everything. It is not clear whether a more partial pattern match is useful without imposing a potentially awkward lexical convention on function names just to satisfy the pattern match.

Support for pattern matching in AspectJ is relatively easy since it is Java-based and so there is no overloading and the * character is free for use in patterns. Direct adoption of the same policy in FOG would not be possible since overloads need resolution and * is used for pointers. The problem of pattern syntax is soluble by expressing pattern names as strings. Thus

    void *print**(ostream&, ***)

might select all functions whose name starts with print, that return void and have an ostream& as a first parameter.

An alternative solution is available by meta-programming. A pattern-matching routine filters the set of all member function names and invokes a customised meta-function for each of the filtered names. Much of this functionality can be provided by a meta-library, and would be considerably assisted by a built-in meta-function to support an elemental pattern match.

    auto::~auto()
    {
        $std::map(customMetaFunction,
            $std::filter($functions(), "print**");
    }
for which `std::map()` and `std::filter()` represent to-be-implemented components of a meta-library, `functions()` is built-in to FOG. `customMetaFunction` performs the per-function meta-programming, and might also be a standard meta-library component. Invocation from the meta-destructor of `auto` ensures that the custom functionality is applied to all classes. This an area for further research. It would appear that most of the solution lies in the domain of a meta-library rather than fundamental FOG functionality.

F.1.4 `#line` directive

The `#line` directive is not used in source programs generated by human beings. It provides a very simple but useful mechanism for automatic source code generators to ensure that compilers and debuggers refer to the original source lines rather than some scrambled intermediate. `#line` performs this role adequately and needs no replacement, although an extension with a more cryptic free format spelling could be considered to free the `#` token once Cpp has been discontinued. The line-literal would be discarded along with whitespace in translation phase 7.

```
line-literal:
  ~ { line-context-seq }
line-context-seq:
  line-context
  line-context-seq line-context
line-context:
  domain optionally file-line optionally line-number
domain:
  identifier
file-line:
  string-literal
line-number:
  decimal-literal
```

The optional `domain` supports definition of line numbers for more than one source domain, with the list of contexts supporting multiple contexts. For instance code passed first through yacc++ and then cfront needs to report both yacc input and yacc output line numbers, so that an enhanced cfront might include a `line-literal` such as

```
~({"Grammar.y" 21 cxx "yacc.tab.c" 127})
```

The first domain is unspecified and defaults to source. With this information, enhanced debugging systems and their users can select the appropriate file upon which to perform source-level debugging.

F.2 Resolution of parsing difficulties

This section reviews the specific problems that arise in parsing C++ and shows how they are resolved using the multi-pass or superset grammar approach.

F.2.1 Context-free problems

The C++ (and C) grammar violates the requirement for a context-free grammar, since "New context-dependent keywords are introduced into a program by `typedef`, `namespace`, `class`, `enumeration` and `template declarations" (§A.1-1). When an identifier is encountered, semantic information is needed, since there is a context-dependency on type names and on template names.

```
typedef-name: identifier
namespace-name: original-namespace-name namespace-alias
```
If the grammar defined by the standard is to be followed very closely, semantic information is apparently needed to classify identifiers into one of

- class-name
- enum-name
- identifier
- namespace-alias
- original-namespace-name
- template-name
- typedef-name

However quite what constitutes context-dependency in the grammar depends on how much the grammar is intended to specify. A complete grammar might include all the language constraints on definition/reference ordering, template instantiation and function overloading. From such a strict perspective almost all languages are context-dependent (at least when implemented using a first order grammar).

When a grammar is solely concerned with the conversion of a token stream into an Abstract Syntax Tree to support a subsequent semantic analysis, the requirements on the grammar are much less stringent. In Section 5.7.1 it was shown that type information was not necessary. In Section 5.6.2 it was shown that a lack of template name information could also be accommodated, although the consequent complexity might not justify that approach.

The context-dependencies are therefore reviewed from the less ambitious standpoint of AST creation.

### F.2.1.1 #include anomaly

The arguments of the C++ #include directive use non-standard forms. In the

```cpp
#include "string"
```

form, there is no recognition of escape sequences. And in the

```cpp
#include <file>
```

form, the angle brackets act as string delimiters, rather than template delimiters or arithmetic operators.

### Resolution

This context-dependence is readily resolved within the lexical analysis processing by switching the lexer into an alternate state starting at the recognition of a #include and continuing to the end of the line. While in this alternate state a different tokenization policy is adopted,
F.2.1.2 Type information

A full semantic interpretation of a C++ program obviously requires a knowledge of the types. Unfortunately this information is also needed for a complete syntactic disambiguation of

- declaration/expression ambiguity (Section F.2.5.1)
- parenthesised-call/cast-parenthesis ambiguity (Section F.2.5.2)
- parenthesised-binary/cast-numeric ambiguity (Section F.2.5.2)
- call/functional-cast ambiguity (Section F.2.5.3)
- new placement/initializer ambiguity (Section F.2.5.5)
- sizeof type/value ambiguity (Section F.2.5.6)
- typeid type/value ambiguity (Section F.2.5.7)
- template argument type/value ambiguity (Section F.2.5.8)

F.2.1.3 < as template-start or greater than (§14.2-3)

A misparse resulting from the lack of template context is difficult to resolve because the two meanings of < and > do not result in localised errors to the tree structure. The arithmetic operators are infix binary operators and have no requirements for associated punctuation, whereas the template brackets must be paired.

A problem arises for an expression such as

\[ V < W < X > ( Y ) > ( Z ) \]

which, in C, would be four unambiguous comparisons. However in C++, there are alternate meanings depending upon which of V or W are template names.

Correct determination of template names requires that names be resolved in the correct scope and may require a template to be instantiated.

\[ a->b<int>::c<... \]

a in current scope.
b in scope of a.
c in scope of b<int>.

Resolution

Correcting a template misparse is an inconvenient but not a particularly difficult AST rearrangement. Although a back-tracking search for a syntactically consistent interpretation is of exponential complexity, the implementation and results presented in Section 5.8.2 show that this does not arise in practice.

F.2.1.4 > as template-end or greater than (§14.2-3)

Within a template, the meaning of an unnested > changes to close the template rather than perform an arithmetic operation. This is not a context-dependency, since the interpretation is dependent on the preceding parse context. The parser knows whether it is in a template and so the grammar can be written to resolve the conflict.

Resolution

The expression rules with higher precedence than > are duplicated to omit the arithmetic > behaviour. This duplicated behaviour is used whenever parsing within a template. As a result the parser keeps track of in/out of template context and
Implementation

Meta-compilation for C++

distinguishes between > nested within parentheses or brackets as part of its normal operation. The cost is about 25 extra rules.

F.2.1.5 Meta-types

The use of meta-types in Section 3.1.5.5 introduces a form of context dependency. In the declaration

```cpp
auto expression e = a + b & c;
```

the syntax used following the `=` is determined by the expression meta-type. This dependency could be eliminated by flattening the grammar to support each meta-type individually in every relevant rule.

However when the same dependency exists in parsing a function argument

```cpp
auto bool meta_function(expression e) { ... }
if (meta_function(a + b & c))
```

It appears that the semantic knowledge of the meta-type of the meta-function parameter must influence the syntactic parsing.

Resolution (multi-pass)

The multi-pass implementation of FOG was syntax-driven, using the meta-type of a meta-function or meta-variable to guide the parse. This only required loose coupling between syntactic and semantic processing, since meta-function and meta-variable definitions can change only at the end of a statement or declaration.

Resolution (superset)

The superset implementation of FOG parses for generic syntax elements, and so syntactic and semantic processing are isolated and there is no meta-type context dependency.

F.2.2 Trivial Ambiguities

The C++ grammar "accepts a superset of valid C++ constructs" (§A-1). Two simple examples of the ambiguities that arise from the overlap between subgrammars are described below.

F.2.2.1 Empty statement

In the syntax for a statement:

```cpp
statement:
  expression-statement
  simple-declaration   // As part of a declaration-statement
...
expression-statement:
  expressionopt ;

simple-declaration:
  decl-specifier-seqopt init-declarator-listopt ;
```

both `expression-statement` and `simple-declaration` provide a cover for the empty statement comprising just a semicolon. The ambiguity is trivial, but must be eliminated in order to create an unambiguous grammar for an automated parser tool.

F.2.2.2 Template parameter

In the syntax for template parameters
Meta-compilation for C++

F.2.3 Syntactic Ambiguities

[Roskind91] identified the major C++ ambiguities that existed prior to templates. This section provides an update to the list and shows how each can be resolved using a superset parse. This section describes only those ambiguities that are syntactically ambiguous. The next two sections discuss further apparent ambiguities: one that is not ambiguous at all, and others that may be deferred for resolution after syntactic analysis.

F.2.3.1 Dangling else (§6.4.1)

The dangling else ambiguity arises in languages with no end of if marker:

```c++
if (a)
  if (b)
    ...
else // else (!a) or (!b) ?
  ...
```

The ambiguity is resolved by definition to the inner-most if, requiring the parser to shift the else on to the parser stack and to continue parsing the if (b) statement, rather than reducing the stack, completing the if (b) statement, and continuing the if (a) statement.

Resolution

Resolution of the dangling else ambiguity is implemented by using a %prec rule.

```c++
selection_statement:
  "if" '(' condition ')' statement %prec SHIFT_THERE
  | "if" '(' condition ')' statement "else" statement
  | "switch" '(' condition ')' statement
```

The SHIFT_THERE precedence specified for the reduction of the shorter rule is lower than that of the else token and so the shift of the else is favoured.

F.2.3.2 < as template-start or less than (§14.2-3)

[Roskind91] does not report this ambiguity because he did not implement templates.

The template-name problem is a context-dependency and has been described in Section F.2.1.3.

F.2.3.3 Multiply nested scope (§7.1-2)

[Roskind91] does not report this ambiguity because he did not implement arbitrary scope nesting.

There is an ambiguity between

A::B::C

and

A::B::C

and

A::B::C
which is resolved by language definition to favour the longest possible decl-specifier-seq as the type.

Resolution

A single %prec rule in the FOG grammar resolves the conflict that arises from this ambiguity.

\[\text{id\_scope: id "::"} \]
\[\text{nested\_id: id} \quad \text{%prec SHIFT\_THERE} \quad \text{id\_scope nested\_id} \]
\[\text{scoped\_id: nested\_id} \quad \text{"::" nested\_id}\]

F.2.3.4 new-type-id (§5.3.4-2), conversion-function-id (§12.3.2-4)

The name of a new-type-id in a new-expression

\[\text{new-expression:} \quad \text{opt new new-placement\_opt new-type\_id new-initializer\_opt} \]
\[\text{opt new new-placement\_opt ( type\_id ) new-initializer\_opt}\]

new-type-id:
\[\text{type-specifier-seq new-declarator\_opt}\]

new-declarator:
\[\text{ptr\_operator new-declarator\_opt}\]
\[\text{direct\_new-declarator}\]
\[\text{direct\_new-declarator:} \quad [ \text{expression} ] \quad \text{direct\_new-declarator [ constant-expression ]}\]

and of a conversion-function-id

conversion-function-id:
\[\text{operator conversion-type\_id}\]

conversion-type-id:
\[\text{type-specifier-seq conversion-declarator\_opt}\]

conversion-declarator:
\[\text{ptr\_operator conversion-declarator\_opt}\]

may each end in a * or &, which can cause an ambiguity with respect to a subsequent expression.

\[\text{new int ** * p;} \quad // (new (int **)) * (p)\]
\[\text{new int **) * (*p) \quad // (new (int *)) * (*p)\]
\[\text{new int **) * (**p) \quad // (new (int)) * (**p)\]
\[\text{&operator int ** + p;}\]

Each is resolved by definition to maximise the length of the type name.

Resolution

A %prec resolves the ambiguity in accordance with the language specification, using the one production ptr_operator_seq_opt to implement conversion-declarator_opt and part of new-declarator_opt.

\[\text{ptr\_operator\_seq\_opt:} \quad \text{/* empty */} \quad \text{%prec SHIFT\_THERE} \quad \text{/* Maximise type length */} \quad \text{ptr\_operator ptr\_operator\_seq\_opt}\]

F.2.3.5 Array of operator ambiguity

operator new and operator new[] (and operator delete and operator delete[]) are valid declarator-ids. It is unclear whether

\[\text{int operator new [ ];}\]
declares an array or a scalar.

Resolution

The ambiguity is removed by
• excluding operator new[] and operator delete[] from the grammar
• accepting a missing array dimension in an expression

Semantic processing identifies the array form from the parsed array of scalar form.
Alternatively, unnecessary semantic effort can be avoided by retaining the array forms and using two %precs to resolve the two consequent shift-reduce conflicts.

F.2.4 Deep Ambiguities

There are some C++ constructs that require a significant amount of lookahead to determine which of two alternative syntaxes is in use.

F.2.4.1 Bit-field or Inheritance

[Roskind91] identifies an ambiguity following

```cpp
class A { class B : [Roskind91]
        which could form part of an anonymous bit field
    const int C = 3;
    class A { class B : C, D, E = 5; }; // C is a bit-field width, D,E are variables
```

or a base class

```cpp
class C {};
class D {};
class E {};
class A { class B : C, D, E {}; }; // A::B inherits privately from C, D and E
```

class A is not really part of the ambiguity. It just serves to avoid the semantic quibble that there are no bit-fields at global scope. There is in fact no ambiguity anyway, because the inheritance declaration must eventually lead on to an open brace whereas the bit-field can never be followed by an open brace.

The problem is the need to lookahead through an arbitrary long comma-separated list of names until some keyword (such as public) or punctuation (such as *) clarifies the name list, or until eventually the trailing punctuation resolves the ambiguity.

Resolution

The superset grammar assumes that class A : is the start of a class declaration, and back-tracks to the : if the base-specifier-list is not terminated by a {. This incurs only a very minor performance loss, since the use of redundantly qualified anonymous bit-fields is surely rare, and so back-tracking may never occur in practice.

```cpp
class_specifier:
    class_key scoped_id '{' member_specification.opt '}'
    class_key scoped_id ':' mark base_specifier_list
    '{' unmark member_specificationopt '}'
    class_key '{' member_specificationopt '}'
    class_key ': ' base_specifier_list
    '{' member_specificationopt '}'
    "template" class_specifier
```
elaborated_typeSpecifier:
    class_key scoped_id '::' mark error { rewind_colon(); }
| class_key scoped_id %prec SHIFT_THERE
| "enum" scoped_id %prec SHIFT_THERE
| "typename" scoped_id
| "template" elaborated_typeSpecifier

The %prec on class_key scoped_id resolves two conflicts. It forces a following : to go through the inheritance lookahead test, and suppresses the spurious interpretation as a function-name when followed by a { (Section F.2.6.2).

Attempting to resolve this problem by parsing for a shared prefix with a more generalised expression syntax proves to be rather difficult, since the generalisation to share a prefix allows the constructor initializer list to provide a third alternative. This construct also ends in a { and so

    class A : name {};

would satisfy the generalised syntax of both constructor and class inheritance. Generalising the syntax further is not possible since a constructor takes a list of statements whereas a class takes a list of declarations. A statement and a declaration cannot be unified since the syntax for the label of a goto statement is highly ambiguous with respect to an anonymous bit field (see Appendix F.2.7.2).

F.2.4.2 Type I functions

The original form of C function declarations is not normally supported by C++ compilers. The syntax presents challenges in avoiding conflicts, and does not tie in well with the generalised name solution of the superset.

Resolution

The grammars in Appendix B and Appendix C implement Type I function declarations but only at a severe (25%) cost to the parsing efficiency. To avoid ambiguities, a lookahead parse is performed following almost any closing parenthesis not at statement level. The severe inefficiencies most commonly follow typedefed pointers to functions:

    typedef A (*B)(C);

for which (C) is a valid generalised first parameter, and it is only after parsing many subsequent declarations, which may include complete class definitions, that the missing function-body is eventually detected. Initiating a lookahead search after every close parenthesis interacts very badly with the initiation of binary searches to resolve template ambiguities. It is advisable to constrain the generality by maintaining a type I enabled flag and type I active flag so that type I lookahead is initiated less often and so that incompatible syntaxes (such as templates) terminate the lookahead more rapidly.

F.2.5 Semantic Ambiguities

The traditional parsing approach needs to resolve semantics during syntactic analysis and so encounters ambiguities that need type information. These ambiguities are all deferred until the semantic analysis by the superset parse.

F.2.5.1 Declaration/Expression ambiguity (§6.8)

Section 5.5.3.2 discussed the ambiguity whereby

    T(a);
could be:

- an expression-statement invoking the function or constructor $T$ with argument $a$
- a declaration of a variable of type $T$ and redundantly parenthesised name $a$

Resolution

The superset grammar eliminates the ambiguity but requires semantic processing to resolve type-dependent problems identified in Section F.2.1.2 and Section F.2.6. [The multi-pass FOG parser resolved the ambiguity by parsing declarations in the first pass and expressions in the second pass. Statements within functions were not parsed and so no semantic corrections were required.]

F.2.5.2 Parenthesised-call / cast-parenthesis
parenthesised-binary / cast-unary ambiguity

A full semantic interpretation of a C++ program obviously requires a knowledge of the types. Unfortunately this information is also needed for a correct syntactic interpretation of an expression using a C cast followed by a unary operator or call.

```c
(T)-5       // This is a cast if T is a type
(t)-5       // This is a subtraction if t is not a type
(T)(5)      // This is a cast if T is a type
(t)(5)      // This is a function call if t is not a type
```

Resolution for binary operator

Without type information, the above cases cannot be distinguished. At most one of the two possibilities can be parsed correctly. The misparse must be detected later and corrected. The superset grammar misparses the unary operator as the binary operator, since the subsequent semantic correction to change a binary operator into a cast is simpler than changing a unary into a binary:

- The change from binary requires replacing the binary node by a cast node and inserting a unary operator on the leading child node. This involves only the erroneous node and its children.
- A change from unary to cast would require inserting a cast node in the parent hierarchy of the erroneous node, and parent traversal is not normally supported by tree algorithms.

Preferring the binary operation is probably slightly more efficient. The use of C casts is discouraged in C++ and so the need to make a correction to a cast should be rare.

The misparse resulting from the lack of type information is tractable because the incorrect parse results in a small easily resolved error in the parse tree.

Resolution for parenthesis

When a possible cast is followed by a parenthesised expression, the resolution has to be in the opposite direction, favouring the cast, since the presumption of a function-call would preclude the possibility of a subsequent non-parenthesised term:

```c
(a) (b) (c) (d) e (f) (g) (h);
```

[The multi-pass implementation of FOG used back-tracking to resolve ambiguities. The cast ambiguity was resolved by establishing a mark following any open parenthesis and then attempting to parse a cast. If the cast failed a nested expression was parsed. This use of back-tracking was inefficient, required a syntax rearrangement to isolate the leading parenthesis and resolved the
ambiguity in the opposite direction. Since expressions were only used within the context of declarations, inadequacies were not significant.]

F.2.5.3 Call/functional-cast ambiguity
As described in Section 5.7.1.4, the functional-cast is totally subsumed by a call and so has been eliminated from the superset grammar.

F.2.5.4 Destructor name/one's complement ambiguity (§5.3.1-9)
An unqualified destructor name cannot appear in an expression because of the ambiguity with a complement operator.

    ~ X ();       // This is ~(X()) even if X is a class

Unqualified destructors are therefore omitted from the superset parse.

Resolution
The missing unqualified destructor name in a declaration must be recovered from its complement expression form. The semantic processing must check all complement nodes to detect the misparse.

F.2.5.5 new-placement/new-initializer ambiguity
Type information is needed to distinguish between an omitted new-placement and an omitted new-initializer when

    new (a)(b)

is parsed against

\[
\begin{align*}
\text{new-expression:} & \\
& : ^* \text{new-placement}_\text{opt} \text{ new-type-id new-initializer}_\text{opt} \\
& : ^* \text{new new-placement}_\text{opt} \ ( \text{type-id} ) \text{ new-initializer}_\text{opt}
\end{align*}
\]

\[
\begin{align*}
\text{new-placement:} & \\
& ( \text{expression-list} )
\end{align*}
\]

\[
\begin{align*}
\text{new-initializer:} & \\
& ( \text{expression-list}_\text{opt} )
\end{align*}
\]

Resolution
The parser creates an AST node with two child expressions for the ambiguous case. Semantic processing determines that if the first child is a type, then the new-placement has been omitted, or alternately if non-type that the new-initializer defaults.

[The multi-pass implementation of FOG parsed for new-placement present on a first pass and absent on a second.]

F.2.5.6 sizeof ambiguity
Type information is required to distinguish the overlap between the sizeof a value and of a type. The syntactic ambiguity resulting from the lack of type information may be resolved by recognising that the generalised syntax accepts a parenthesised parameter-declaration as a unary-expression. The syntax for parenthesised type-id is therefore covered by the syntax accepted as a unary-expression.

\[
\begin{align*}
\text{unary-expression:} & \\
& \ldots \text{sizeof } \text{unary-expression} \\
& \ldots \text{sizeof } \ ( \text{type-id} ) \\
& \ldots
\end{align*}
\]
Resolution

The parser creates an AST node with a child expression. Semantic processing determines which syntax is in use.

[The multi-pass implementation of FOG used two passes to parse the two alternatives without changing the grammar.]

F.2.5.7 typeid ambiguity

Type information is required to distinguish the overlap between the typeid of a value and that of a type. In the absence of type information, the superset parse of expression and type-id is available as a parameter-declaration-clause.

postfix-expression:

...   typeid ( expression- t ype-i-d   
    typeid ( type-id )  
    typeid ( parameter-declaration-clause )  
   ...

Resolution

The parser creates an AST node with a child expression. Semantic processing determines which syntax is in use.

[The multi-pass implementation of FOG used two passes to parse the two alternatives without changing the grammar.]

F.2.5.8 Template argument type/value ambiguity

The appropriate template specialisation cannot be selected without knowledge of inheritance.

Successful parsing of the arguments of a template require type information to distinguish type and value parameters.

Resolution

Templates do not need to be specialised during the primary parse. An AST node describing the template-argument-list is created which the subsequent semantic processing elaborates.

A superset grammar that covers

template-argument:

   assignment-expression
   type-id
   template-id

can be used to create AST nodes during the syntax analysis that can be interpreted later once type information is available. The generalised parameter-declaration covers template-argument.

[The multi-pass implementation of FOG used two passes to attempt value and type parsing. Value and type were tried independently for each argument of a multi-argument template resulting in exponential complexity.]

F.2.6 New C++ ambiguities

A strict superset of the declaration and expression syntaxes should introduce no new ambiguities. However parsing is eased by taking a rather larger superset. In particular a very general policy is adopted for names. As a result some new ambiguities are created.
F.2.6.1 **ctor-initializer or Named Bit-field**

The generalised name makes

name
valid as the **declarator** in a **function-definition** with the result that there is deep ambiguity between

\[
\text{type name : m(i), n(j), p(l) } \// \text{ A constructor}
\]
\[
\text{type name : m(i), n(j), p(l); } \// \text{ A bit-field, and variables}
\]

Resolution

The superset grammar uses a shared prefix for the two cases, so that no decision is made until the disambiguating punctuation is reached. Use of a shared prefix requires considerable generalisation, tolerating **assignment-expressions** as each **ctor-initializer**, and an **assignment-expression** rather than an **identifier** as the bit-field name.

All valid syntaxes are parsed correctly. Many invalid syntaxes are accepted and need diagnosis at the semantic level.

The shared prefix appears as **constructor_head** in the grammar of Appendix B. The prefix causes greater problems in the implementation of **tree_statement**, since, as described at the end of Section 3.1.1, there is an ambiguity between the use of a comma to separate multiple components of a single element, and its use to separate multiple elements. The implementation in Appendix C is carefully structured to avoid shift-reduce conflicts.

F.2.6.2 **class-specifier or enum-specifier as function-definition name**

The generalised name makes

\[
\text{enum X}
\]
valid as the **declarator** in a **function-definition** with the result that

\[
\text{enum X();}
\]
is accepted as both an **enum-specifier** and **function-definition**.

Resolution

The resolution of the bit-field or inheritance ambiguity (Appendix F.2.4.1) has the beneficial side effect of solving this ambiguity for

\[
\text{class X();}
\]
The ambiguity for **enum** produces a shift-reduce conflict, which is resolved in favour of the only valid possibility: the **enum-specifier**. No semantic repair is necessary.

Elaboration of the grammar to be more restrictive on function names might be possible for C++, although there is a risk of introducing conflicts between **function-definitions** and **simple-declarations**. Such elaboration is not possible in FOG where the extended **re-using-declaration** syntax accepts function names without parentheses.

F.2.6.3 **delete[] ambiguity**

Introduction of the abstract array declarator [] as part of a **primary-expression** creates an ambiguity in

\[
\text{delete[](p)}
\]
between
  • the intended array operation
  • deletion of an abstract array of functions.

Resolution
The array form is removed from the grammar and supported by generalising a cast expression to accept a bracketed as well as a parenthesised prefix. The resulting bracketed cast ambiguities are resolved in exactly the same way as parenthesised cast ambiguities, save for the benefit that the resolution is correct except for the array delete which must be detected semantically.

\[
\text{cast_expression: unary_expression}
  \begin{align*}
    &| \\
    \text{abstract_expression: parenthesis_clause}
    &| '[]' \quad \text{expression.opt 1}'
  \\
  \text{parenthesis_clause: parameters_clause cv_qualifier_seq.opt}
    &| \text{exception_specification.opt}
\]

Generalisation of the cast also covers the gcc indexed array initializer extension described in Section 3.1.4.2.

F.2.6.4 linkage-specification ambiguity
The generalised name parsing supports interpretation of

\[
\text{extern "$C$";}
\]

as a conventional declaration since a string-literal satisfies the syntax of a generalised name.

Resolution
This is a false ambiguity resolved by unconditional treatment of the keyword extern followed by a string-literal as the pre-amble for a linkage-specification.

F.2.7 Extra FOG ambiguities
The presentation of the FOG extensions in Chapter 3 identified many of the ambiguities associated with the extensions and parsing approaches to avoid them. All ambiguities that cannot easily be avoided are identified in this section.

F.2.7.1 built-in-type-id maximised
The lack of distinction between meta-type-names and meta-names leads to two ambiguities when a meta-name follows a meta-type as in a meta-variable-declaration.

\[
\text{auto short long :: int :: a = 0;}
\]

This could be
  • a meta-declaration of ::a with meta-type short long:int
  • a meta-declaration of ::int::a with meta-type short long
  • a meta-declaration of long::int::a with meta-type short
  • a meta-expression assigning to short long::int::a

Resolution
These ambiguities are resolved by two of the C++ disambiguation principles: maximise to the left, and prefer declarations to expressions.
The elemental meta-type-name is therefore maximised syntactically as short
long even though this is a semantic error. The entire name is maximised as short
long::int::a to select the expression interpretation.

F.2.7.2 label preferred to anonymous bit-field

There is an ambiguity between a label and bit-field:

label: int(value); // label then simple-declaration
type : int(value); // anonymous bit field

which does not arise in C++ since labels and bit-fields are syntactically exclusive.
Syntax generalisation in FOG removes this exclusivity introducing a parsing conflict.

Resolution

The conflict is resolved to preserve C++ syntax, by preferring the label interpretation as part of a statement. The same %prec that forces an identifier <
to be shifted for a template test also forces a shift to prefer a labeled-statement
whenever a labeled-statement is a syntactic option.

id: identifier %prec SHIFT_THERE

labeled_statement: identifier '::' looping_statement

F.2.7.3 handler-seq maximised

The generic syntax parsing for a tree statement supports a handler-seq following a
try-block.

void f() try {} catch (a) {} catch (b) {} catch (c) {}

leading to an ambiguity as to where the function ends and a subsequent handler-seq
begins.

Resolution

The ambiguity is resolved by definition to maximise the length of the left hand
syntax element.

handler_seq: handler %prec SHIFT_THERE

| handler handler_seq

F.2.7.4 access-specifier

The false ambiguity resulting from the generalised name interpretation of

public : x(y);

as an implicit int anonymous bit-field was discussed in Section 3.1.3.2.

Resolution

The ambiguity is resolved by a %prec to favour the accessibility-specifier.

decl_specifier_affix: ...

| access_specifier %prec SHIFT_THERE

accessibility_specifier: access_specifier '::'

F.2.7.5 inline/ and virtual/ cannot be expressions

The false ambiguities resulting from the generalised name parsing of

name1 inline / interface ( name2 ) ;
were discussed in Sections 3.1.3.5 and 3.1.3.6.

Resolution

The ambiguities are resolved by `%prec` to favour the switch.

```cpp
functionSpecifier: "explicit" %prec SHIFT_THERE
  | "inline" %prec SHIFT_THERE
  | "virtual" %prec SHIFT_THERE
  | '!' "inline"
  | "inline" '/' "implementation"
  | "inline" '/' "interface"
  | '!' "virtual"
  | "virtual" '/' "pure"
```

F.2.7.6 **using string-literal is not an expression**

A false ambiguity arises from the generalised name interpretation of

```
using "string";
```

as a re-using-declaration since a string-literal satisfies the generalised syntax of a name and a re-using-declaration is parsed as a generalised declaration or expression.

Resolution

The ambiguity is resolved by treating the keyword `using` followed by a string-literal as an include-declaration unconditionally.

```cpp
declSpecifierAffix: ... "using" %prec SHIFT_THERE
includeDeclaration: "using" string ...
```

F.3 **Semantic checks**

The syntactic analysis tolerates a very generalised syntax that merges declarations and expressions and requires no knowledge of template or type names. The analysis builds an Abstract Syntax Tree that must then be processed to incorporate semantic information.

The semantic processing comprises four parts; resolution of syntax ambiguities, correction of misparses, validation of semantics and implementation. Each part is naturally performed in a distinct pass over the AST adding little to the substantial amount of processing needed for a complex language such as C++.

The origin of most of the deferred ambiguities has been described in Sections F.2.5, F.2.6 and F.2.7. Their resolution during semantic analysis is considerably eased by operation on the AST where the whole of the construct to be analysed is available, whereas the more conventional approach must struggle with constraints of parser lookahead and shift-reduce conflicts. Both approaches ultimately require the same decision code.

F.3.1 **Resolution**

A conventional C++ parse uses semantic information to resolve most if not all ambiguities during the syntax parse. The superset parse uses a generalised syntax to avoid semantic leakage. As a result the parse is incomplete and requires ambiguities deferred from the syntactic analysis to be resolved during semantic analysis. The most significant ambiguity is the declaration/expression ambiguity, but there are a number of other minor type related problems to resolve.
F.3.1.1 Declaration/Expression (§6.8)

The usage of the same AST nodes for declarations and expressions must be resolved. The superset grammar parses declarations as a slightly generalised expression, and so when appropriate, the declaration must be determined from an expression tree. The context in which a tree is used sometimes determines whether a declaration or expression is required. For instance, an initializer for a default argument can only be an expression. In most situations, such as a statement within a function, an ambiguity must be resolved.

Some aspects of the ambiguity can and must be resolved by an accurate syntax check. For instance * const can only occur in a declaration, whereas && can only occur in an expression. Further aspects can be resolved when type information is available as it is for actual declarations. The syntax resolution for an actual declaration can therefore perform a strong check, whereas only a weak check can be performed for a potential declaration.

Resolution occurs by a dataflow propagation of a set of boolean flags that indicate whether the AST satisfies a number of syntactic hypotheses. These hypotheses are propagated from the leaves to the root, applying the constraints applicable at each node so that the syntaxes satisfied by the tree can be determined.

The current partially functional implementation propagates a bit vector of 217 hypotheses ranging from EPSILON, IDENTIFIER, CLASS_NAME, ELABORATED_TYPESpecifier, via PARAMETER_DECLARATOR and PARAMETER_DECLARATION_CLAUSE, to BIT_FIELD_DECLARATION or META_FUNCTION_DECLARATION.

Maintenance of these hypotheses at the 20 or so interesting tree nodes is relatively straightforward since few hypotheses propagate through each node type. Most operator nodes propagate only the BASIC_EXPRESSION hypothesis. Unary operators such as & * and ~ that have meaning in declarators also propagate the various DECLARATOR hypotheses. Tree nodes for the binary operators * and & resolve their alternate parse for a left-hand type and right-hand declarator. Tree nodes involving parentheses are where the conventional ambiguities are resolved and where there is some implementation complexity, but only in the number of ifs that map incoming to outgoing hypotheses.

The overall operation of the tree nodes of course restores the grammar which was folded into the expression grammar; there are separate hypotheses for DECLARATOR and DIRECT_DECLARATOR. However, because of the very different semantic constraints associated with different names, it is necessary to maintain independent hypotheses for ABSTRACT, CONVERSION, FUNCTION, PURE_FUNCTION, INIT, META, PARAMETER or BIT_FIELD_DECLARATIONs and DECLARATORS and sometimes DIRECT_DECLARATORS, DECLARATION_ids and DECLARATOR_ids. It is not appropriate to merge all DECLARATOR hypotheses as a single parameterised hypothesis, since each hypothesis is potentially independent and propagated up the tree in parallel. Sharing state between hypotheses could cause cross-talk and therefore fail to achieve sufficient precision to resolve very finely balanced declaration/expression ambiguities correctly.

The same propagation algorithm is used for weak and strong hypotheses. The difference lies at the leaves. For a strong determination, an identifier node is assessed to determine whether the identifier satisfies each of a class, enum namespace, typedef and template hypothesis. For a weak determination, all hypotheses are satisfied, since there is no context to contradict the hypothesis.
All hypotheses are propagated in parallel, and so a single pass over the tree identifies all satisfied syntaxes. Ambiguity resolution amongst those of interest determines whether to

- correct misparses in the expression metaobject
- create a potential declaration specifier metaobject

The corrections that may be required are outlined in Appendix F.3.2. Creation of a potential declaration specifier involves a further tree traversal towards the naming node where the appropriate metaobject can be created, and then decorated as it is returned through intervening nodes.

F.3.1.2 new-placement/new-initializer ambiguity

See Section F.2.5.5.

F.3.1.3 sizeof and typeid ambiguity

The reuse of the same AST nodes for types and values probably simplifies the implementation.

F.3.1.4 pure-specifier

The

```
pure-specifier:
  = 0
```

syntax is covered by

```
constant-initializer:
  = constant-expression
```

so that if even the lexer distinguishes the lexeme '0' from the number 0, there is little possibility of identifying the distinction in a generalised parse.

Resolution

The superset parse does not resolve pure-specifiers or even constant-initializer. The semantic pass must identify each from the more general assignment-expression, taking care to distinguish character streams such as 00 or 0x0 from 0.

F.3.1.5 explicit-instantiation

The generalised name binds template close to the name, rather than as a prefix to a declaration. Semantic processing must therefore locate the keyword deeper in the AST than might be expected. explicit-instantiation does not appear as a distinct production.

F.3.1.6 Implicit int

[Roskind91] questions the enthusiasm with which compilers have stopped supporting the deprecated implicit int for C functions with no declared return type.

Resolution

The purely syntactic superset parse does not distinguish implicit int from a constructor. Implicit int is therefore parsed successfully. The semantic pass may easily distinguish implicit int from a constructor, since a constructor has a name that matches its scope. Any other name is implicit int.
F.3.2 Correction

Misparses resulting from incorrect assumptions during the syntactic analysis must be corrected, or accounted for during subsequent processing.

F.3.2.1 Template/arithmetic ambiguity

The binary search to identify a consistent syntax for the template or arithmetic interpretation of identifier followed by < identifies the wrong syntax for approximately 0.01% of statements (Section 5.8.2). These errors must be corrected. Detection of the errors just requires the semantic test that should have been performed during lexical analysis to be performed upon the AST. If the test result is inconsistent with the tree structure, then the tree must be rearranged. This is not particularly easy, since the two interpretations have distinctly different tree structures, and rearrangements of nodes associated with arithmetic operators must account for arithmetic precedences. The complexity is reduced a little by the use of generic syntax elements, and so there is no difference between the subtree for an expression and that for a template-argument. There is no template-argument meta-type.

F.3.2.2 Parenthesised-call / cast-parenthesis

Parenthesised-binary / cast-unary ambiguity

Semantic correction of the AST is required where a binary operator that is also a unary operator has a type as its first child. This indicates that the binary operator should be replaced by a C-style cast, and that the equivalent unary operator be applied to the first term in the tree headed by the second child.

The reverse correction occurs where the parenthesised function name in a function call was misparsed as a cast. An apparent cast to a non-type should be corrected to a parenthesised function call.

F.3.2.3 Call/functional-cast ambiguity

A function call invoking a non-class type is recognised as a functional-cast.
A function call invoking a class type is recognised as a constructor.

F.3.2.4 Destructor name/one’s complement ambiguity (§5.3.1-9)

A function call to a complemented class-name in a declaration is detected as a destructor declaration.

F.3.2.5 Array of operator ambiguity

Arrays of operator new or operator delete are recognised as the array forms operator new[] and operator delete[].

F.3.2.6 delete[] ambiguity

An [] prefix following a delete operator is recognised as denoting delete[].

F.3.3 Validation

The syntax generalisations to establish a simple superset grammar accept many nonsensical constructs.

double root2 = static sqrt(2) const throw(); // declaration clutter
void b extern = int a char; // unlimited generality
A::A() : this(0), 5+?, -3, int(...) [] // expression/name clutter
The form of each generalised expression and declaration must be carefully validated to diagnose the numerous anomalies that should normally be trapped by syntax analysis.

This has advantages in terms of error diagnosis. Because the syntax accepts so much, there is a good chance that many common semantic errors will survive the syntactic analysis, build a plausible tree, at which point the difference from a closely matching possibility can be reported. An accurate syntax parse is liable to encounter the ubiquitous unclassified "parse error" more often, or require extra elaboration in the grammar to cover probable user errors.

F.3.4 Implementation

The semantic analysis of the AST for each parsed statement must update the symbol table to keep track of the enlarged program and make any additional declarations accessible to subsequent statements.

For some unpleasant statements such as typedefs containing more than one declarator, or function parameter lists, the semantic analysis update must perform the symbol table updates so that names introduced early in the statement have appropriate visibility later in the same statement. Achieving this during syntactic analysis, and before expression/declaration ambiguities have been resolved, requires very careful coding of the symbol table update to reflect the tentative semantic analysis of an incomplete syntactic analysis.

Again these are exactly the same decisions that need to be performed by any accurate C++ compiler, however their implementation is eased by operation in the context of the AST rather than within the straitjacket imposed by avoiding conflicts while looking infinitely far ahead in the parser grammar.

F.4 File Syntaxes

Additional syntax supports

- include declarations (replacing #include)
- specification of and allocation to output file names
- specification of generated include file dependencies for scopes
- specification of generated include file dependencies for code

This syntax contributes four new categories of declaration, only three of which are supported as parts of classes. Include declarations are only valid at global scope. The integration of these declarations with the main grammar is described in Section 3.1.5. The syntax to support function-specific and function-body-specific declarations is described in Section 3.1.4.7.

F.4.1 Target File Names

The names of target files are used in two ways.

- to create the file in the file system
- to reference the file in a generated #include directive

These two names need not be exactly the same. The name used to create the file must be complete and specified with respect to the current user directory. Names appearing in generated #include directives need only be resolvable by the subsequent compiler, with respect to a potentially different current working directory and using a search list of include file paths.

The file name used to refer to a file in a generated #include directive is composed from 3 parts

prefix name suffix

The name is normally that of a class or namespace.
The way in which *prefix* and *suffix* components are determined is summarised in Table F.1.

<table>
<thead>
<tr>
<th>Component</th>
<th>File</th>
<th>Template</th>
<th>Default</th>
<th>Environment Variable</th>
<th>Command Line Token</th>
<th>Source Code Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>path</strong></td>
<td>Interface</td>
<td>Path</td>
<td>Fog::interface_path</td>
<td>-hd /path</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Template</td>
<td>Path</td>
<td>Fog::template_interface_path</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
<td>Path</td>
<td>Fog::implementation_path</td>
<td>-cd</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Template</td>
<td>Path</td>
<td>Fog::template_implementation_path</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>prefix</strong></td>
<td>Interface</td>
<td>Prefix</td>
<td>Fog::interface_prefix</td>
<td>-hp /prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Template</td>
<td>Prefix</td>
<td>Fog::template_interface_prefix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
<td>Prefix</td>
<td>Fog::implementation_prefix</td>
<td>-cp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Template</td>
<td>Prefix</td>
<td>Fog::template_implementation_prefix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>suffix</strong></td>
<td>Interface</td>
<td>Suffix</td>
<td>.hxx Fog::interface_suffix</td>
<td>-hs /suffix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Template</td>
<td>Suffix</td>
<td>.H Fog::template_interface_suffix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
<td>Suffix</td>
<td>.cxx Fog::implementation_suffix</td>
<td>-cs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Template</td>
<td>Suffix</td>
<td>.C Fog::template_implementation_suffix</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table F.1 File Name Component Contributions**

The *suffix* is normally one of `.cxx`, `.hxx`, `.c`, `.H` depending upon whether the file is associated with the implementation or interface of a non-template or template class. These defaults may be changed by defining the corresponding environment variable. This default setting may in turn be overridden by a command line token. Finer-grained settings can be supplied using the appropriate switch in the source code syntax of a *file-name*.

The *prefix* may be used to locate nested include files such as *sys/stdlib.h*. The *prefix* has no default, but may be given a value from an environment variable, command line option or source code switch. For template prefixes (and paths) the environment variable option is determined first from a template-specific name, and if that is undefined the non-template name is used.

When files are created, a further path prefix may be added. If no prefix has been defined then the file is created as

*prefix name suffix*

otherwise the file is created as

*path separator prefix name suffix*

where *path* is the specified path, and *separator* is a file system dependent joining character. Only `/` is implemented, which is suitable for any file system with a C interface.
F.4.2 Target File Identities

When the name of a file is specified, it may be qualified to override the default naming or structuring policies.

file-name:
  string-literal
  file-name / interface
  file-name / implementation
  file-name / template
  file-name / utility
  file-name / guard = string-literal
  file-name / noguard
  file-name / path = string-literal
  file-name / prefix = string-literal
  file-name / suffix = string-literal

The file name is specified as a normal string (or concatenation yielding a string), rather than the subtly different syntax for a string in a #include directive. File names containing escape sequences must therefore be appropriately escaped.

It is assumed that the string-literal incorporates prefix, name and suffix components, unless /interface or /implementation is specified, in which case the string-literal should comprise just the name component.

/ interface
/ implementation
  Specify that a prefix and a suffix should be applied to the name using values determined in accordance with Table F.1.

/ template
  Specifies the use of the template, rather than non-template, policy for any path, prefix or suffix.

/ utility
  Specifies a utility level of an output file. Only /utility (or /frozen) is meaningful, as a further assurance that a file should not be emitted.

/ guard = string-literal
  Specifies the spelling of the include file guard, overriding the default derived from the file name and its suffix.

/ noguard
  Specifies that there should be no include file guard in the target file.

/ path = string-literal
  specifies the path to be prefixed to the name when creating the file. If neither /interface nor /implementation is used, the implementation rather than interface path is used in accordance with Table F.1.

/ prefix = string-literal
/ suffix = string-literal
  specify an override for the prefix and/or suffix parts of the file name, but are only used if one of /interface or /implementation has been used to specify an algorithmic contribution to the name.

F.4.3 Target File Placement

The target file for a particular scope may be changed by a file-placement-declaration.
file-placement-declaration:
  export / implementation =opt file-specifier ;
  export / interface =opt file-specifier ;
  export / noimplementation ;

file-specifier:
  file-name
  file-entity
  file-entity / implementation
  file-entity / interface

file-entity:
  declarator-id
  elaborated-type-specifier
  namespace scoped-id

export/implementation specifies that the implementation (non-inline function bodies and initialised variables) should be located in file-specifier. Similarly export/interface specifies that the interface should be located in file-specifier.

Each file-specifier may be a file-name (the qualified name of a file) or file-entity (the name of some declared entity). In the case of a file-entity, the required file is either the implementation or the interface file to which the declarations of the entity are emitted. An explicit distinction is made by use of /implementation or /interface to qualify the entity name. An implicit distinction is made in the absence of an explicit qualifier, by using the implementation or /interface qualifying the export keyword.

The default placement recursively locates declarations with their enclosing class. Top level classes and namespaces are located in a file named from the declaration name. This policy may be overridden to support

- arbitrary placement

```cpp
class Base {
  export / implementation "Bases.cxx"; // emit implementation to Bases.cxx
  export / interface "Bases.hxx"; // emit interface to Bases.hxx
};
```

- placement of a derived class with its base:

```cpp
class Derived : public Base {
  export / implementation Base; // emit implementation with
  // implementation of Base
  export / interface Base;       // emit interface with interface of Base
};
```

- placement of a local class entirely within an implementation file:

```cpp
class Local {
  export / implementation Base;       // emit implementation with
  // implementation of Base
  export / interface Base / implementation; // emit interface with
  // implementation of Base
};
```

export/noimplementation specifies that there should be no implementation, as is often intended to be the case for private constructors and assignment operators. Specification of export/noimplementation ensures that any violation of this intent as a result of composition is detected.

F.4.4 File-spaces

A file-space is a set of declarations grouped in order to share file placement. A file-space therefore supports placement of selected declarations in a specific pair
of implementation and interface files. A file-space is not a declarative region, and so declarations appearing within a file-space form part of the enclosing scope.

filespace-specifier:
    namespace / file file-name compound-declaration

filespace-declaration:
    filespace-specifier ;

The implementation and interface files are determined on the assumption that file-name constitutes a suitable name for application of prefix and suffix components in accordance with Table F.1. Since both interface and implementation file names are determined algorithmically, the use of /interface or /implementation on the file-name is redundant. A file-placement-declaration may be used to change one of both of the file names. The interface file is only used for namespaces and function definitions inline in the interface, since the declarations of a class must necessarily occur within the one class: they cannot be partitioned across multiple interface files.

In

    class A
    {
        namespace/file "FirstRegion"
        {
            // ...
        };
        namespace/file "SecondRegion"
        {
            export/implementatio B;
            // ...
        };
    }

declarations within FirstRegion form part of class A and are typically emitted to A.hxx, FirstRegion.hxx and FirstRegion.cxx. Declarations within SecondRegion are similarly part of class A, but are typically emitted to A.hxx, SecondRegion.hxx and B.cxx.

File-spaces may be used to compose a file structure upon the global namespace or upon C source, and to place explicit-instantiations.

F.4.5 Target File Dependencies

Files emitted by FOG have include file guards around declarations and file inclusions that are arranged to ensure that C++ requirements for forward references are satisfied. Include file references and forward declarations are generated by analysis, and as a result are often tighter (better) than those produced by hand.

The current implementation of FOG performs a very limited analysis of function bodies, so FOG may miss dependencies. A perfect analysis could not guarantee to catch all dependencies, given the inadequate type information available for template arguments of templates that remain uninstantiated.

Target file dependency declarations are therefore provided to allow missed declarations to be specified.

file-dependency-declaration:
    using / implementation =opt file-specifier ;
    using / interface =opt file-specifier ;

These concepts correspond to "uses (for interface)" and "uses (for implementation)" in [Booch91]. However, whereas a full model requires all dependencies to be specified, FOG only requires the specification of those that it fails to deduce.
A dependency declaration may be used as a class member-declaration to specify a dependency for all declarations within the class

class Base
{
    using/implementation "stdlib.h"; // Entire implementation uses stdlib.h
    using/interface string;  // interface uses string
};

Class level specification is very heavy handed, providing little indication of where the dependency is triggered and poor support for declarations originating from algorithm-centric source code. This may provoke spurious includes to be generated if the class implementation is spread into many files. FOF therefore supports declaration level specification of dependencies:

As an object-statement to restrict the dependency to a single function, irrespective of its derivation context

class Base
{  
    public void f()
    
    : {  
        using/implementation ofstream;  
        
        [ ... ];  
        //Base::f implementation uses ofstream
        
    } ;
};

or as part of a function-used-block to further restrict the dependency to the derivation contexts in which the associated function-body is used.

class Base
{  
    public void f()
    
    : {  
        using ofstream [ ... ];  
        
        //Function contribution uses ofstream
        
    } ;
};

F.4.6 File inclusion

A replacement syntax for #include is provided by overloading the using keyword and recognising that using followed by a string has no meaning in C++.

#include-declaration:
using slash-includeopt slash-utilityopt string-literal ;

slash-include:
/ include

slash-utility:
/ utility

utility:
  pool
  emit
  utility
  frozen

The utility indicates whether the declaration belongs to a frozen free-standing external class utility, to a pool of declarations, or whether the declaration is to be emitted. The default behaviour is for all declarations arising from #include files to be treated as frozen utility declarations that must not be changed or re-emitted.

The replacement syntax for #include provides qualifiers to select the nature of included declarations. The replacement syntax is only valid as a global-scope declaration.

Inclusion only occurs upon the first encounter, and so include file guards are unnecessary.
using "file.h";  // Include file.h preserving prevailing utility
using/utility "string.h";  // Include string.h as utility declarations
using/pool "shared.h";  // Include shared.h as pooled declarations

The utility or pool attributes apply throughout the included file and its nested
inclusions. The prevailing mode is restored after the include completes.

Declarations read while in utility mode provide information that enables FOG to
correctly analyse and emit the wanted code, but do not directly cause emission of
the utility code. However, utility classes may indirectly contribute to emitted code
by providing derivation rules or meta-programs that do contribute to classes that
are emitted. Any attempt to change the functionality of utility classes can be
diagnosed and rejected.

The new include syntax is restricted to top-level declarations, unlike the #include
directive which could potentially appear in the middle of an expression. The
rationale behind this is a corollary of adopting an include-just-once policy to avoid
the need for include file guards. This implies that a new-style include cannot
contribute to multiple scopes and so should not contribute to any. Every included
declaration should be part of the global name-space. Perhaps a little of the old
behaviour should be restored by introducing using/reinclude as part of any
declaration or statement.
<table>
<thead>
<tr>
<th>Term</th>
<th>Page_numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof</td>
<td>369, 376</td>
</tr>
<tr>
<td>template argument</td>
<td>370</td>
</tr>
<tr>
<td>template keyword</td>
<td>181</td>
</tr>
<tr>
<td>tree-arguments</td>
<td>41</td>
</tr>
<tr>
<td>typeid</td>
<td>370, 376</td>
</tr>
<tr>
<td>unqualified destructor</td>
<td>44</td>
</tr>
<tr>
<td>using</td>
<td>374</td>
</tr>
<tr>
<td>ambiguity string-literal</td>
<td>372, 374</td>
</tr>
<tr>
<td>ambiguous</td>
<td>73, 345</td>
</tr>
<tr>
<td>anonymous bit-field</td>
<td>41</td>
</tr>
<tr>
<td>anonymous union</td>
<td>207</td>
</tr>
<tr>
<td>ANSI</td>
<td>265</td>
</tr>
<tr>
<td>A0</td>
<td>265</td>
</tr>
<tr>
<td>AOP</td>
<td>265</td>
</tr>
<tr>
<td>ARM</td>
<td>54, 265</td>
</tr>
<tr>
<td>array composition</td>
<td>123</td>
</tr>
<tr>
<td>array_modifer</td>
<td>65, 91, 280</td>
</tr>
<tr>
<td>array-of</td>
<td>91</td>
</tr>
<tr>
<td>asm-definition</td>
<td>40, 91, 121, 287</td>
</tr>
<tr>
<td>Aspect Oriented Programming</td>
<td>3, 239, 262</td>
</tr>
<tr>
<td>AspectJ</td>
<td>359</td>
</tr>
<tr>
<td>assignment</td>
<td>87</td>
</tr>
<tr>
<td>assignment operator</td>
<td>53</td>
</tr>
<tr>
<td>assignment_expression</td>
<td>65, 91, 280</td>
</tr>
<tr>
<td>assignment-expression</td>
<td>91, 371</td>
</tr>
<tr>
<td>AST</td>
<td>70, 146, 201, 205, 265, 374, 378</td>
</tr>
<tr>
<td>at-literal</td>
<td>38, 281</td>
</tr>
<tr>
<td>auto</td>
<td>59, 62, 74, 104, 135, 140, 288, 291</td>
</tr>
<tr>
<td>B</td>
<td>78</td>
</tr>
<tr>
<td>backslash</td>
<td>257</td>
</tr>
<tr>
<td>back-track</td>
<td>187, 362</td>
</tr>
<tr>
<td>backward substitution</td>
<td>99</td>
</tr>
<tr>
<td>base</td>
<td>66, 92, 280</td>
</tr>
<tr>
<td>base class composition</td>
<td>120</td>
</tr>
<tr>
<td>base meta-class</td>
<td>18, 64, 135, 266</td>
</tr>
<tr>
<td>base_specifier</td>
<td>65, 92, 280</td>
</tr>
<tr>
<td>base_specifier-40, 45, 64, 92, 120, 286, 287</td>
<td></td>
</tr>
<tr>
<td>base-specifier-clause</td>
<td>64, 288</td>
</tr>
<tr>
<td>base-specifier-list</td>
<td>120, 366</td>
</tr>
<tr>
<td>BETA</td>
<td>215</td>
</tr>
<tr>
<td>binary search</td>
<td>377</td>
</tr>
<tr>
<td>bison</td>
<td>145</td>
</tr>
<tr>
<td>bison+</td>
<td>145</td>
</tr>
<tr>
<td>bit-field</td>
<td>373</td>
</tr>
<tr>
<td>BNF</td>
<td>7, 145, 265</td>
</tr>
<tr>
<td>body</td>
<td>56, 126, 288</td>
</tr>
<tr>
<td>boundary</td>
<td>58</td>
</tr>
<tr>
<td>break</td>
<td>95</td>
</tr>
<tr>
<td>built_in</td>
<td>92</td>
</tr>
<tr>
<td>built_in_specifier</td>
<td>65, 66, 280</td>
</tr>
<tr>
<td>built_in_typeSpecifier</td>
<td>92</td>
</tr>
<tr>
<td>concatenation</td>
<td>33, 35, 75, 78, 99</td>
</tr>
<tr>
<td>character</td>
<td>65, 91, 280</td>
</tr>
<tr>
<td>character-level substitution</td>
<td>77, 96, 97, 145</td>
</tr>
<tr>
<td>comment_line_numbers</td>
<td>211</td>
</tr>
<tr>
<td>comments</td>
<td>3</td>
</tr>
<tr>
<td>compilation model</td>
<td>201</td>
</tr>
<tr>
<td>compiler diagnostics</td>
<td>211</td>
</tr>
<tr>
<td>composite declaration</td>
<td>119</td>
</tr>
<tr>
<td>composition</td>
<td>6, 55, 119</td>
</tr>
<tr>
<td>file</td>
<td>382</td>
</tr>
<tr>
<td>utility</td>
<td>204</td>
</tr>
<tr>
<td>compound-declaration</td>
<td>51, 91, 120, 291, 382</td>
</tr>
<tr>
<td>compound-statement</td>
<td>60, 91</td>
</tr>
<tr>
<td>formal parameter</td>
<td>117</td>
</tr>
<tr>
<td>Term</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>concrete</td>
<td>220</td>
</tr>
<tr>
<td>condition</td>
<td>40, 69, 287, 289</td>
</tr>
<tr>
<td>consistency check</td>
<td>212</td>
</tr>
<tr>
<td>const</td>
<td>285</td>
</tr>
<tr>
<td>const_cast</td>
<td>220</td>
</tr>
<tr>
<td>constant_expression</td>
<td>65, 91, 280</td>
</tr>
<tr>
<td>constant-expression</td>
<td>50, 69, 91, 285, 289</td>
</tr>
<tr>
<td>constructor initializer composition</td>
<td>125</td>
</tr>
<tr>
<td>constructor_head</td>
<td>371</td>
</tr>
<tr>
<td>context</td>
<td>2</td>
</tr>
<tr>
<td>context dependency</td>
<td>363</td>
</tr>
<tr>
<td>context-dependency</td>
<td>44</td>
</tr>
<tr>
<td>#include</td>
<td>361</td>
</tr>
<tr>
<td>&lt;</td>
<td>362</td>
</tr>
<tr>
<td>&gt;</td>
<td>362</td>
</tr>
<tr>
<td>meta-type</td>
<td>363</td>
</tr>
<tr>
<td>type-name</td>
<td>362</td>
</tr>
<tr>
<td>context-dependent</td>
<td>361</td>
</tr>
<tr>
<td>context-free</td>
<td>87, 100, 257, 360</td>
</tr>
<tr>
<td>continuation line</td>
<td>78</td>
</tr>
<tr>
<td>continue</td>
<td>95</td>
</tr>
<tr>
<td>control-statement</td>
<td>40, 62, 68, 287</td>
</tr>
<tr>
<td>conversion</td>
<td>40, 62, 68, 287</td>
</tr>
<tr>
<td>meta-type</td>
<td>95</td>
</tr>
<tr>
<td>conversion-declarator</td>
<td>365</td>
</tr>
<tr>
<td>conversion-function-id</td>
<td>44, 282, 365</td>
</tr>
<tr>
<td>conversion-type-id</td>
<td>365</td>
</tr>
<tr>
<td>copy constructor</td>
<td>50, 53</td>
</tr>
<tr>
<td>CORBA</td>
<td>240</td>
</tr>
<tr>
<td>-cp</td>
<td>379</td>
</tr>
<tr>
<td>Cpp replacement</td>
<td>75</td>
</tr>
<tr>
<td>CRC replacement</td>
<td>212, 265</td>
</tr>
<tr>
<td>-cs</td>
<td>379</td>
</tr>
<tr>
<td>-ctd</td>
<td>379</td>
</tr>
<tr>
<td>ctor-initializer</td>
<td>56, 288, 367, 371</td>
</tr>
<tr>
<td>-ctp</td>
<td>379</td>
</tr>
<tr>
<td>-cts</td>
<td>379</td>
</tr>
<tr>
<td>curiously recurring template pattern</td>
<td>230</td>
</tr>
<tr>
<td>cv_qualifier</td>
<td>65, 91, 280</td>
</tr>
<tr>
<td>cv-qualifier</td>
<td>40, 91, 103, 122, 285, 287</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>-D</td>
<td>346</td>
</tr>
<tr>
<td>dangling else ambiguity</td>
<td>69</td>
</tr>
<tr>
<td>dangling pointer</td>
<td>92</td>
</tr>
<tr>
<td>database</td>
<td>212</td>
</tr>
<tr>
<td>date</td>
<td>73, 346</td>
</tr>
<tr>
<td>debugger</td>
<td>211, 360</td>
</tr>
<tr>
<td>declSpecifier</td>
<td>65, 91, 280</td>
</tr>
<tr>
<td>declaration</td>
<td>65, 69, 88, 91, 95, 280, 289, 368</td>
</tr>
<tr>
<td>actual</td>
<td>18, 85</td>
</tr>
<tr>
<td>body</td>
<td>59</td>
</tr>
<tr>
<td>potential</td>
<td>18, 85</td>
</tr>
<tr>
<td>declaration-seq</td>
<td>60, 64, 288</td>
</tr>
<tr>
<td>declaration-statement</td>
<td>40, 91, 287</td>
</tr>
<tr>
<td>declarator</td>
<td>92, 375</td>
</tr>
<tr>
<td>declarator-id</td>
<td>44, 91, 285, 291, 381</td>
</tr>
</tbody>
</table>

**INDEX**

*Meta-compilation for C++*

29-June-2001
enum-specifier 40, 44, 92, 122, 284, 287, 371
error 73, 345
  recovery 141
error diagnosis 258, 378
exception 66, 92, 280
exception specification 261
exception_declaration 92
exception_specification 65, 91, 280
exception-declaration 92
exception-specification 91, 125
exception-specification composition 125
execute 18
execution 108
exit 56, 126, 288
explicit 123, 140, 291
explicit-instantiation 40, 91, 121, 287, 376
explicit-specialization 40, 91, 121, 287
export 123, 291, 357, 381
exported list 94
exported-tree 66, 140, 289, 291
expression 40, 65, 69, 91, 95, 280, 287, 289,
  370, 376, 377
expression tree 89
expression-statement 40, 91, 102, 287, 363,
  368
extended language 138
extension-namespace-definition 45, 284
extern 123, 372
F
-f 210
FFT 265
file 73, 201, 291, 346, 382
dependency 382
dependency graph 208
name 378
suffix 202
system 378, 379
file_dependency_specifier 65, 280
file_placement_specifier 65, 280
file-dependency-declaration 40, 55, 121,
  287, 288, 291, 382
file-entity 291, 381
file-id-list 56, 288
file-line 360
file-name 291, 380, 381, 382
file-placement-declaration 40, 55, 121, 287,
  288, 291, 381
file-space 202, 381
filespace 66, 92, 280
filespace composition 120
filespace_specifier 65, 92, 280
filespace-declaration 40, 55, 62, 283, 287,
  288, 292, 382
filespace-specifier 40, 92, 287, 291, 382
file-specifier 291, 381
find 73, 345
flex 145
flex++ 145
floating-literal 36, 281
Flyweight 222
FOG 6, 33, 257, 262, 265
FOG grammar 313
for 69, 289
force 2
formal_parameter_name_resolution 117
forward declaration 204, 205, 382
friend 123, 210
frozen 204, 291, 383
funarg 259
function 66, 92, 135, 208, 280
  body 127
  entry 127
  exit 127
  post 127
  pre 127
  pure virtual 57
  scope 287
  value 127
  function body 135
  function body composition 125
  function composition 124
  function placement 53
  function_modifer 65, 280
  function_modifer 91
  functionSpecifier 65, 92, 135, 280
  functional cast 103
  functional-cast 369, 377
  function-body 56, 60, 91, 288, 383
  function-definition 40, 46, 52, 56, 92, 285,
    287, 371
  function-returning 92
  function-specifier 47, 49, 103, 283
  function-statement-scope 56
  function-try-block 40, 56, 125, 261, 287, 288
  function-used-block 55, 56, 288, 383
  future research 211
G
garbage collector 89, 226
gcc 50
Generative Programming 3
generic syntax element 141, 363, 373
get_cpp 73, 346
get_env 73, 346
getenv 346
global namespace 209, 382
GNU 145, 265
GoF 265
goto 62, 367
GP 265
grammar 18, 266
congcatenation 34, 35
tokenization 34, 36
guard 202, 291, 380
GUI 265
<table>
<thead>
<tr>
<th>Index</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>handler</td>
<td>65, 91, 280</td>
</tr>
<tr>
<td>handler-seq</td>
<td>40, 287, 373</td>
</tr>
<tr>
<td>hash code</td>
<td>212</td>
</tr>
<tr>
<td>-hd</td>
<td>379</td>
</tr>
<tr>
<td>hide</td>
<td>118</td>
</tr>
<tr>
<td>-hp</td>
<td>379</td>
</tr>
<tr>
<td>-hs</td>
<td>379</td>
</tr>
<tr>
<td>-hld</td>
<td>379</td>
</tr>
<tr>
<td>-hpt</td>
<td>379</td>
</tr>
<tr>
<td>-hts</td>
<td>379</td>
</tr>
<tr>
<td>id</td>
<td>44, 67, 104, 282, 288, 289</td>
</tr>
<tr>
<td>identifier</td>
<td>35, 36, 37, 65, 91, 102, 140, 280, 281, 282, 291</td>
</tr>
<tr>
<td>idiom</td>
<td>2, 215</td>
</tr>
<tr>
<td>if</td>
<td>69, 289, 364</td>
</tr>
<tr>
<td>implementation</td>
<td>291, 380, 381, 382</td>
</tr>
<tr>
<td>file</td>
<td>202</td>
</tr>
<tr>
<td>implicit int</td>
<td>376</td>
</tr>
<tr>
<td>include</td>
<td>291, 383</td>
</tr>
<tr>
<td>include file</td>
<td>119</td>
</tr>
<tr>
<td>include file guard</td>
<td>202, 382</td>
</tr>
<tr>
<td>include file reference</td>
<td>382</td>
</tr>
<tr>
<td>include-declaration</td>
<td>40, 121, 287, 291, 374, 383</td>
</tr>
<tr>
<td>incompatibility</td>
<td>73</td>
</tr>
<tr>
<td>linkage specification</td>
<td>75</td>
</tr>
<tr>
<td>indexed array initializer</td>
<td>50, 372</td>
</tr>
<tr>
<td>inheritance</td>
<td>358</td>
</tr>
<tr>
<td>hierarchy</td>
<td>57</td>
</tr>
<tr>
<td>utility</td>
<td>204</td>
</tr>
<tr>
<td>inherited</td>
<td>216</td>
</tr>
<tr>
<td>InheritedTypedef</td>
<td>215, 216</td>
</tr>
<tr>
<td>init-declarator</td>
<td>49, 55, 56, 63, 92, 285</td>
</tr>
<tr>
<td>init-declarator-list</td>
<td>49</td>
</tr>
<tr>
<td>initializer</td>
<td>49, 55, 288</td>
</tr>
<tr>
<td>default</td>
<td>50</td>
</tr>
<tr>
<td>initializer_clause</td>
<td>65, 95, 280</td>
</tr>
<tr>
<td>initializer-clause</td>
<td>40, 49, 50, 285, 287</td>
</tr>
<tr>
<td>inline</td>
<td>47, 77, 125, 283</td>
</tr>
<tr>
<td>inline/implementation</td>
<td>48</td>
</tr>
<tr>
<td>inline/interface</td>
<td>48</td>
</tr>
<tr>
<td>integer-literal</td>
<td>36, 281</td>
</tr>
<tr>
<td>integrity</td>
<td>211</td>
</tr>
<tr>
<td>interface</td>
<td>291, 380, 381, 382</td>
</tr>
<tr>
<td>file</td>
<td>202</td>
</tr>
<tr>
<td>interpret</td>
<td>18</td>
</tr>
<tr>
<td>intrinsic meta-type</td>
<td>88</td>
</tr>
<tr>
<td>intrinsic-meta-type-name</td>
<td>65, 280</td>
</tr>
<tr>
<td>invocation scope name resolution</td>
<td>112</td>
</tr>
<tr>
<td>is_leaf</td>
<td>58</td>
</tr>
<tr>
<td>is_pure</td>
<td>58</td>
</tr>
<tr>
<td>is_root</td>
<td>57</td>
</tr>
<tr>
<td>isomorphic</td>
<td>222</td>
</tr>
<tr>
<td>isomorphism</td>
<td>222</td>
</tr>
<tr>
<td>iteration-statement</td>
<td>62, 91</td>
</tr>
<tr>
<td>iterator</td>
<td>65, 93, 280</td>
</tr>
<tr>
<td>J</td>
<td>Java 3, 260, 262</td>
</tr>
<tr>
<td>jump-statement</td>
<td>62, 91</td>
</tr>
<tr>
<td>K</td>
<td>keyword 65, 91, 280</td>
</tr>
<tr>
<td>L</td>
<td>label 62, 367, 373</td>
</tr>
<tr>
<td>labeled-statement</td>
<td>41, 62, 91</td>
</tr>
<tr>
<td>ALR</td>
<td>257, 265</td>
</tr>
<tr>
<td>large programs</td>
<td>5</td>
</tr>
<tr>
<td>lbibug</td>
<td>252</td>
</tr>
<tr>
<td>icc</td>
<td>252</td>
</tr>
<tr>
<td>leaf</td>
<td>261</td>
</tr>
<tr>
<td>lex</td>
<td>2, 145</td>
</tr>
<tr>
<td>lexeme</td>
<td>266</td>
</tr>
<tr>
<td>lexical analysis</td>
<td>266</td>
</tr>
<tr>
<td>lexical hierarchy</td>
<td>106</td>
</tr>
<tr>
<td>lexical redundancy</td>
<td>5</td>
</tr>
<tr>
<td>lexical scope name resolution</td>
<td>111, 115</td>
</tr>
<tr>
<td>line</td>
<td>360</td>
</tr>
<tr>
<td>line number</td>
<td>211</td>
</tr>
<tr>
<td>line-context</td>
<td>360</td>
</tr>
<tr>
<td>line-context-seq</td>
<td>360</td>
</tr>
<tr>
<td>line-literal</td>
<td>360</td>
</tr>
<tr>
<td>linkage</td>
<td>66, 92, 280</td>
</tr>
<tr>
<td>linkageSpecification</td>
<td>65, 92, 280</td>
</tr>
<tr>
<td>linkageSpecication</td>
<td>40, 60, 92, 120, 123, 287, 372</td>
</tr>
<tr>
<td>incompatibility</td>
<td>75</td>
</tr>
<tr>
<td>list</td>
<td>91, 94</td>
</tr>
<tr>
<td>literal</td>
<td>70, 290</td>
</tr>
<tr>
<td>LL</td>
<td>257, 265</td>
</tr>
<tr>
<td>logically-const</td>
<td>219</td>
</tr>
<tr>
<td>lookahead</td>
<td>70, 367</td>
</tr>
<tr>
<td>LR</td>
<td>265</td>
</tr>
<tr>
<td>M</td>
<td>macro 76, 259, 346</td>
</tr>
<tr>
<td>macro processing</td>
<td>96</td>
</tr>
<tr>
<td>make</td>
<td>210</td>
</tr>
<tr>
<td>dependency file</td>
<td>210</td>
</tr>
<tr>
<td>makefile</td>
<td>210</td>
</tr>
<tr>
<td>malloc</td>
<td>92</td>
</tr>
<tr>
<td>marshalling</td>
<td>18, 240</td>
</tr>
<tr>
<td>member variable delegation</td>
<td>358</td>
</tr>
<tr>
<td>member variable initializer</td>
<td>125</td>
</tr>
<tr>
<td>member-declaration</td>
<td>383</td>
</tr>
<tr>
<td>member-declarator</td>
<td>55, 63, 92, 285</td>
</tr>
<tr>
<td>member-specification</td>
<td>60</td>
</tr>
<tr>
<td>mem-initializer</td>
<td>40, 91, 287</td>
</tr>
<tr>
<td>memory leak</td>
<td>92</td>
</tr>
<tr>
<td>meta-</td>
<td>18, 62, 266</td>
</tr>
<tr>
<td>meta_class</td>
<td>66, 92, 280</td>
</tr>
<tr>
<td>meta_class_specifier</td>
<td>65, 92, 280</td>
</tr>
</tbody>
</table>
INDEX

meta-function 66, 92, 135, 280
meta-function-specifier 65, 92, 135, 280
meta-parameter 92
meta-parameter-specifier 92
meta-statement-declaration 69, 289
meta-type 65, 91, 135, 280
meta-variable 66, 92, 280
meta-variable-specifier 65, 92, 280
meta-additive-expression 71, 290
meta-and-expression 71, 290
meta-array-of 93
meta-cast 103
meta-class 18, 63, 64, 129, 135, 266, 288
meta-class-declaration 40, 64, 287, 288
meta-class-key 63, 288, 289
meta-class-specifier 63, 65, 288, 289
meta-compilation 63
meta-compilation-stages 83
meta-compiler 18
meta-compile-time 18
meta-conditional-expression 57, 71, 288, 290
meta-construction 136, 261
meta-constructor 68, 289
meta-control-declaration 40, 55, 62, 68, 121, 287, 288
meta-control-statement 68
meta-data 18
meta-declaration 18
meta-destruction 136, 261
meta-destructor 68, 289
meta-equality-expression 71, 290
meta-exclusive-or-expression 71, 290
meta-expression 121
meta-expression-statement 40, 55, 62, 68, 135, 287, 288
meta-function 18, 38, 62, 65, 67, 135, 289
meta-function-definition 40, 55, 92, 287, 288
meta-function-returning 94
meta-id 63, 104, 288
meta-inclusive-or-expression 71, 290
meta-inheritance 18, 64, 135
meta-level 133
meta-logical-and-expression 71, 290
meta-logical-or-expression 71, 290
meta-mai 261
meta-main 136
meta-multiplicative-expression 71, 290
meta-name 63, 288
meta-name-space 39, 103
meta-nested-constructor-id 68, 289
meta-nested-destructor-id 68, 289
meta-nested-id 63, 104, 288
meta-object 134
meta-object-value 95
meta-parameter 67, 92, 289
meta-parameter-list 67, 289
meta-pointer-to 94
meta-postfix-expression 70, 290
meta-primary-expression 70, 290
meta-programming 18, 135
meta-record-of 95
meta-relational-expression 71, 290
meta-scoped-constructor-id 68, 289
meta-scoped-destructor-id 68, 289
meta-scoped-id 39, 63, 66, 67, 70, 104, 286, 288, 289, 290
meta-shift-expression 71, 290
meta-statement 18, 289
method 95
misparse 193, 362
modifier 65, 91, 280
modularity constraint 207
Monitor 247
monitor 246
MOP 265
MPC++ 260
multi-pass grammar 19, 37, 128, 357, 360, 363, 368, 369
mutable 123, 220
Mutate 219
name 65, 91, 280
capture 76
conversion 132
mangling 119
resolution 55, 103
named-name-space-definition 45, 284
Namespace 346
name-space 103, 266
namespace 45, 46, 63, 66, 92, 202, 205, 266, 280, 284, 288, 291, 382
namespace_alias 66, 92, 280
namespace_alias_definition 65, 92, 280

Page 390 29-June-2001
namespace_definition 65, 92, 280
namespace-alias 361
namespace-alias-definition 40, 92, 121, 287
namespace-body 45, 60, 123, 284
namespace-declaration 40, 287
namespace-definition 40, 45, 46, 92, 120, 284, 287
namespace-name 360
-nc 210, 212
-ne 210
nested substitution 39
nested-id 44, 282
nested-special-function-id 44, 282
new 365, 369
new-declarator 365
new-expression 365, 369
new-initializer 365, 369
new-placement 365, 369, 376
new-type-id 365
NFA 265
nick-name 54
-no_line_numbers 211
NoAssign 217
NoCopy 217
noguard 291, 380
noimplementation 218, 291, 381
non-reserved word 33
non-reserved-word 37, 281
non-terminal 7, 87, 266
not implemented 218
null pointer 92
number 65, 91, 280
number-literal 35, 36, 91, 280, 281
numeric conversion 36
O
-o 210
object 66, 92, 280
 scope 287
object composition 122
object model 3, 129, 130
Object Orientation 3
object statement composition 121
objectSpecifier 65, 92, 280
object_statement 65, 280
object-statement 55, 60, 288, 383
object-statements-clause 55, 106, 126, 135, 287
object-statement-scope 55
object-statement-seq 55, 288
ObjViisp 129, 130
ODR 265
One Definition Rule 3, 6, 45, 119, 201, 211, 262, 263
OO 265
OOP 265
OpenC++ 260, 359
operator 42, 103, 210, 365
new 92
operator delete 365, 377
operator new 365, 377
operator() 224
operator-function-id 44, 282
original-namespace-definition 45, 284
original-namespace-name 361
OsTraits 236
other-identifier 37, 281
OuterNamespace 346
OuterScope 347
overload 54
overloading
 names 359
 reserved words 357
 syntax-macro 140
P
parameter 66, 92, 280
parameter name 54
parameter name composition 125
parameter type composition 125
parameter_specifier 65, 92, 280
parameter-declaration 40, 92, 102, 287, 364, 369, 370
parameter-declaration-clause 370
parenthesis_clause 372
partial syntax 127
partitioning 205
path 291, 379, 380
pattern 2, 221
pattern name 359
persistent 138
phase 1 35, 193
phase 2 193
phase 3 36, 193
phase 4 194
phase 5 195
phase 6 33, 34, 35, 100, 195
phase 7 33, 34, 36, 100, 195, 360
pointer_modifier 65, 91, 280
pointer-to 92
polymorphic tokens 93
polymorphism 87
syntactic 100
pool 203, 291, 383
post 56, 126, 288
prefix-expression 370
postfix-expression 370
postfix-tree-expression 39, 286
potential declaration 18, 85, 88, 119, 376
potential meta-type 88, 92
conversion to actual 95
potential-meta-type-name 65, 280
pre 56, 126, 288
predicate 57
predictable code 2
prefix 291, 379, 380
preprocessing token 33
pretty printing 210
primary template 202, 207
primary-expression 8, 38, 44, 102, 282, 371
primary-tree-expression 39, 286
private 46, 208
production 7
production rule 7, 266
protected 46, 208
Prototype 220
ptr-operator 91, 103, 365
public 46, 208
punctuation 36, 65, 91, 140, 280, 281, 291
pure virtual 46
pure virtual function 57
pure-specifier 376
pure-virtual 46
purify 344
R
record-of 92
recovery 141
recurring problem 2
reduction rule 266
redundancy
algorithmic 5
Cp 18
lexical 5
reference 92
reference to meta-object 89
reference_modifier 65, 91, 280
ReferenceCount 226
reflection 4, 266
register 123
regular expression 258
reify 4
repeated editing 2
reserved 65, 91, 280
reserved word 59, 66, 139
reserved-word 36, 40, 91, 140, 281, 287, 291
restriction
syntax macros global 140
tree-literal as identifier 102
retokenization 37
return 95
re-using-declaration 51, 358, 374
root
class 266
scope 57, 266
Roskind 148, 364
RTTI 18, 237, 265
rule 7, 266
Run-Time Type Information 4
S
scaffolding 217
ScalarMember 225
Scope 347
scope 66, 92, 280
scopeSpecifier 65, 92, 280
scoped_modifier 65, 280
scoped-id 44, 282
scoped-special-function-id 44, 282
segment 56, 127, 288
selection-statement 62, 91
semantic analysis 266
semantic-level substitution 99
semantics 83
sentence 266
separator 379
set 357
shared prefix 371
SHIFT THERE 364
signature
re-use 52
simple-declaration 49, 52, 56, 92, 102, 357, 363
simple-type-parameter 40, 287
simple-type-specifier 44, 284
single stepping 211
sizeof 71, 290, 369, 376
slash-include 291, 383
slash-utility 291, 383
Smalltalk 3, 130, 220, 260, 262
SmartPointerMember 225
SO 265
SOM 130
SOP 265
source file
protection 210
source-level debugging 360
special-function-id 44, 282
specifier 65, 88, 92, 280, 376
Standard Template Library 222, 357
statement 65, 68, 88, 91, 95, 280, 363
statement-seq 60
Static 108, 117
static 47, 66, 67, 123, 208, 283, 289, 357
StaticFlyweight 221
std
ambiguous 73, 345
date 73, 346
defined 73, 345
diagnostic 73, 80, 345
error 73, 80, 345
file 73, 346
filter 360
find 73, 345
get_cpp 73, 113, 195, 346
get_env 73, 346
map 360
meta-namespace 73, 345
no_alias 80
parse 99, 145, 346
parse_tokens 346
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>usage dependency graph</td>
<td>205</td>
</tr>
<tr>
<td>user-defined</td>
<td></td>
</tr>
<tr>
<td>meta-type</td>
<td>65</td>
</tr>
<tr>
<td>type</td>
<td>63, 64</td>
</tr>
<tr>
<td>using</td>
<td></td>
</tr>
<tr>
<td>8, 44, 51, 56, 65, 66, 74, 92, 123, 280, 284, 288, 289, 291, 357, 374, 382, 383</td>
<td></td>
</tr>
<tr>
<td>template</td>
<td>54</td>
</tr>
<tr>
<td>using_declaration</td>
<td>65, 92, 280</td>
</tr>
<tr>
<td>using_directive</td>
<td>65, 280</td>
</tr>
<tr>
<td>using-declaration</td>
<td>8, 40, 44, 51, 92, 284, 287</td>
</tr>
<tr>
<td>using-directive</td>
<td>40, 91, 121, 287</td>
</tr>
<tr>
<td>utility</td>
<td>74, 203, 204, 291, 380, 383</td>
</tr>
<tr>
<td>inheritance</td>
<td>204</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>value composition</td>
<td>123</td>
</tr>
<tr>
<td>value-parameter_specifier</td>
<td>65, 280</td>
</tr>
<tr>
<td>variable</td>
<td>66, 92, 135, 208, 280</td>
</tr>
<tr>
<td>variable composition</td>
<td>123</td>
</tr>
<tr>
<td>variableSpecifier</td>
<td>65, 92, 280</td>
</tr>
<tr>
<td>variable-statement-scope</td>
<td>55</td>
</tr>
<tr>
<td>view</td>
<td>358</td>
</tr>
<tr>
<td>virtual</td>
<td>49, 125, 283</td>
</tr>
<tr>
<td>constructor</td>
<td>220</td>
</tr>
<tr>
<td>meta-inheritance</td>
<td>135</td>
</tr>
<tr>
<td>Visitor</td>
<td>233</td>
</tr>
<tr>
<td>VLIW</td>
<td>265</td>
</tr>
<tr>
<td>volatile</td>
<td>285</td>
</tr>
<tr>
<td>W</td>
<td></td>
</tr>
<tr>
<td>warning</td>
<td>73, 345</td>
</tr>
<tr>
<td>weaving</td>
<td>3, 242</td>
</tr>
<tr>
<td>while</td>
<td>69, 289</td>
</tr>
<tr>
<td>white-space</td>
<td>33, 38</td>
</tr>
<tr>
<td>whitespace</td>
<td>35</td>
</tr>
<tr>
<td>WholePart</td>
<td>230</td>
</tr>
<tr>
<td>wide string</td>
<td>33, 35</td>
</tr>
<tr>
<td>wide-character</td>
<td>35</td>
</tr>
<tr>
<td>wide-string</td>
<td>35</td>
</tr>
<tr>
<td>worklist</td>
<td>208</td>
</tr>
<tr>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>yacc</td>
<td>2, 145, 146, 197, 252, 257, 265, 344, 360</td>
</tr>
</tbody>
</table>