Genesis: An Extensible Java

by

Ian Lewis BComp Hons

Submitted in fulfilment of the requirements for the

Degree of Doctor of Philosophy

University of Tasmania

February 2005
This thesis contains no material which has been accepted for a degree or diploma by the University or any other institution, except by way of background information and duly acknowledged in the thesis, and to the best of the candidate’s knowledge and belief no material previously published or written by another person except where due acknowledgement is made in the text of the thesis.

Ian Lewis
This thesis may be made available for loan and limited copying in accordance with the
Copyright Act 1968.

Ian Lewis
Abstract

Extensible programming languages allow users to create fundamentally new syntax and translate this syntax into language primitives. The concept of compile-time meta-programming has been around for decades, but systems that provide such abilities generally disallow the creation of new syntactic forms, or have heavy restrictions on how, or where, this may be done.

Genesis is an extension to Java that supports compile-time meta-programming by allowing users to create their own arbitrary syntax. This is achieved through macros that operate on a mix of both concrete and abstract syntax, and produce abstract syntax. Genesis attempts to provide a minimal design whilst maintaining, and extending, the expressive power of other similar macro systems.

The core Genesis language definition lacks many of the desirable features found in other systems, such as quasi-quote, hygiene, and static expression-type dispatch, but is expressive enough to define these as syntax extensions. User-defined macros produce only well-formed syntactic structures via the use of a predefined set of classes that define a Java abstract syntax.

At the heart of Genesis is a flexible parser that is capable of parsing any context-free grammars — even ambiguous ones. The parser is capable of arbitrary speculation and will consider all possible parses. The parser constructs a graph of possible paths, and is capable of dynamically pruning this graph, or combining nodes, based on precedence or associativity rules. This general parser allows macro programmers to forget about parsing, and concentrate on defining new syntax.

One key goal of this system was to address the programmer’s learning curve by providing as simple a system as possible. This was achieved by the use of the flexible parser, the introduction of only one new construct to standard Java, and extensions to make programming macros more user friendly.

The expressiveness of Genesis is wide ranging; it is capable of providing small scale limited use macros, large scale semantic modifications, through to complete language replacements.
To demonstrate this expressiveness, we implement many of the simple test cases found in other systems, such as a type-safe printf, assertions, and iteration statements. These test cases require an ability to perform static type-checking and to manipulate compile-time values and abstract syntax trees. As additional examples of Genesis’ expressive power we also provide implementations of embedded subsets of SQL and Haskell. As a final proof of power, the Haskell subset can operate as a stand-alone extension independent of any recognisable Java code.
Acknowledgements

Thanks to Vishv for being my most long-suffering supervisor and for steadfastly maintaining to this day that he has no idea what I am doing whatsoever.

Also, to Arthur and Julian for helping get me over the line, battered and bruised though I may be.

Thanks to the School of Computing for providing support when it was really necessary and without which I would never even come close to finishing.

To Julian for showing me no matter how annoyingly pedantic I may have thought myself there is always someone worse, and who will scare me.

To Nicole simply for torturing countless cats (as is my understanding) rather than slaying me alive simply because, as my office neighbour, she was Ned to my Homer.

To Adam for giving us all someone to mutter incessantly under our breaths about at his appearance of progress and for all the tennis ball humour — just hilarious. Oh yeah, and for proof-reading.

Thanks to Mark for all of those closed door discussions, the whiteboard Jeffisms, misogynistic research proposals, and for becoming a living urban legend.

To Byeong for showing me that it really is worth taking the time to greet people, and for being brave enough to come drinking with me — once.

To Craig, for scheduling his holiday for the worst possible time… and for making up for it by scheduling another celebratory one once this ordeal is finally over.

Thanks to Banjo’s, Hungry Jacks, Eastlands, Centrepoint, Northgate, KFC, and their ilk for donating places where, strangely, I initially found myself most productive. Particularly thanks to those who donated their power to make this all possible.

To my darling girlfriend Kim for tolerating all those assurances of “two more weeks”, and just being tolerant in general.

And finalee, tanks to those harmleff alcambahol molecools that didn’t hampa me won bit.
# Table of Contents

**Abstract** .......................................................................................................................... VII
**Acknowledgements** ......................................................................................................... IX
**Table of Contents** .............................................................................................................. XI
**List of Code Examples** .................................................................................................... XVIII
**List of Figures** .................................................................................................................. XXI
**List of Tables** ................................................................................................................... XXII

## 1 Introduction ...................................................................................................................... 1

1.1 Overview .......................................................................................................................... 2

1.1.1 Assessment of Success .......................................................................................... 3

1.1.2 Conventions ......................................................................................................... 4

1.2 Sources of Code Examples ......................................................................................... 6

1.2.1 Java ......................................................................................................................... 6

1.2.1.1 Java1.4 ............................................................................................................... 6

1.2.1.2 Java1.5 ............................................................................................................... 7

1.2.2 MultiJava ............................................................................................................... 7

1.2.3 Maya ....................................................................................................................... 7

1.2.4 Java Syntax Extender ........................................................................................... 8

1.2.5 OpenJava ............................................................................................................... 8

1.2.6 Haskell .................................................................................................................... 8

1.2.7 Template Haskell ................................................................................................... 9

1.2.8 Lisp .......................................................................................................................... 9

1.2.8.1 Common Lisp .................................................................................................. 9

1.2.8.2 Scheme ............................................................................................................ 9

1.2.9 C++ .......................................................................................................................... 10

1.2.10 MS^2 ..................................................................................................................... 10

1.2.11 SQL ....................................................................................................................... 10

1.2.12 Pro*C ................................................................................................................... 10

1.2.13 Icon ....................................................................................................................... 11

1.2.14 Ada ....................................................................................................................... 11

1.3 Thesis Structure ......................................................................................................... 12

## 2 Extensibility ..................................................................................................................... 15

2.1 Overview ....................................................................................................................... 16

2.2 Defining Extensible ..................................................................................................... 17

2.2.1 Other Similar Terminology .................................................................................. 17

2.2.2 Previous Extensibility Definitions ....................................................................... 17

2.2.3 Definition of Extensible ....................................................................................... 18

2.3 Language Extension Mechanisms ............................................................................. 20

2.3.1 Library Systems .................................................................................................... 20

2.3.1.1 Threading ....................................................................................................... 21

2.3.1.2 SQL .................................................................................................................. 22

2.3.1.3 Summary ........................................................................................................ 24

2.3.2 Open Compilers ................................................................................................... 24

2.3.2.1 Glasgow Haskell Compiler ........................................................................... 25

2.3.2.2 Summary ........................................................................................................ 25

2.3.3 Text Macros ............................................................................................................ 25

2.3.3.1 Assertions ....................................................................................................... 27

2.3.3.2 Iteration .......................................................................................................... 28

2.3.3.3 Generators ..................................................................................................... 28

2.3.3.4 Message Maps ............................................................................................... 33

2.3.3.5 DEBUG_NEW ................................................................................................. 34

2.3.3.6 Summary ........................................................................................................ 35

2.3.4 Two-tier Languages ............................................................................................ 35

2.3.4.1 C++ Template Meta-programming ................................................................. 36

2.3.5 Integrated Language Features ............................................................................. 39
# Table of Contents

2.4 IS EXTENSIBILITY NECESSARY? ................................................................. 40
  2.4.1 What is Necessary? ............................................................................ 42
  2.4.2 Language Modifications .................................................................. 43
    2.4.2.1 Language Standards Revision / Development ...................... 44
    2.4.2.2 Research Languages ............................................................ 44
    2.4.2.3 Embedded Languages .......................................................... 45
    2.4.2.4 Java 1.5 ............................................................................. 46
  2.4.3 How Much Power is Too Much Power? ........................................... 47
  2.4.4 Multi-paradigm languages ............................................................... 48
  2.4.5 Previous Interest ............................................................................ 49

3 ASSESSMENT OF EXTENSIBILITY ................................................................. 51
  3.1 OVERVIEW ......................................................................................... 52
  3.2 DESIRABLE LANGUAGE PROPERTIES ............................................. 53
    3.2.1 Hygiene and Referential Transparency ..................................... 54
  3.3 CRITERIA FOR RATING EXTENSIBLE LANGUAGES ....................... 56
    3.3.1 Power ...................................................................................... 56
    3.3.2 Usability ................................................................................. 57
    3.3.3 Error Handling ....................................................................... 58
    3.3.4 Previous Research .................................................................. 58
  3.4 BENCHMARK TEST CASES ................................................................. 60
    3.4.1 Simple constructs ...................................................................... 61
      3.4.1.1 Assertions ......................................................................... 61
      3.4.1.2 Iteration ........................................................................... 63
      3.4.1.3 Typesafe Formatted Output .......................................... 65
    3.4.2 Complex Constructs ................................................................... 67
      3.4.2.1 SQL Subset ....................................................................... 67
      3.4.2.2 Generators ........................................................................ 68
      3.4.2.3 Haskell Subset ................................................................... 69

4 REVIEW OF EXTENSIBLE LANGUAGES ................................................... 75
  4.1 OVERVIEW ......................................................................................... 76
  4.2 LISP / SCHEME .................................................................................. 78
    4.2.1 Power ....................................................................................... 79
      4.2.1.1 S-expressions .................................................................... 79
      4.2.1.2 Macros: defmacro ......................................................... 80
      4.2.1.3 Macros: define-syntax .................................................. 81
      4.2.1.4 Syntax Creation .............................................................. 82
    4.2.2 Usability ..................................................................................... 82
      4.2.2.1 Name Capture: gensym .................................................. 83
      4.2.2.2 Name Capture: define-syntax Revisited ......................... 83
    4.2.3 Error Handling ........................................................................... 84
    4.2.4 Applicability to Benchmarks ..................................................... 85
    4.2.5 Extensibility Criteria Assessment ............................................. 86
  4.3 TEMPLATE HASKELL ........................................................................ 88
    4.3.1 Power ....................................................................................... 88
      4.3.1.1 Reification ....................................................................... 89
    4.3.2 Usability .................................................................................... 89
      4.3.2.1 Splicing .......................................................................... 90
      4.3.2.2 Quasi-quotiation ......................................................... 91
    4.3.3 Error Handling ........................................................................... 91
    4.3.4 Worked Examples ...................................................................... 92
      4.3.4.1 Type-safe Formatted Output ...................................... 92
      4.3.4.2 Selection From an N-tuple .......................................... 93
    4.3.5 Applicability to Benchmarks ..................................................... 94
    4.3.6 Extensibility Criteria Assessment ............................................. 95
  4.4 META SYNTACTIC MACRO SYSTEM ............................................... 97
    4.4.1 Power ....................................................................................... 97
    4.4.2 Usability .................................................................................... 97
    4.4.3 Error Handling ........................................................................... 98
TABLE OF CONTENTS

4.4.4 Worked Examples ................................................................. 98
  4.4.4.1 Dynamic Binding ............................................................ 98
  4.4.4.2 Extended Enumerations ................................................. 100
4.4.5 Applicability to Benchmarks ................................................ 102
4.4.6 Extensibility Criteria Assessment ......................................... 103
4.5 JAKARTA TOOL SUITE .......................................................... 105
4.6 JAVA SYNTAX EXTENDER ....................................................... 106
  4.6.1 Power ............................................................................ 106
  4.6.2 Usability ......................................................................... 107
  4.6.3 Error Handling ............................................................... 108
  4.6.4 A Worked Example: foreach ........................................... 108
    4.6.4.1 Underlying Implementation ....................................... 108
    4.6.4.2 The syntax Macro ..................................................... 109
  4.6.5 Applicability to Benchmarks ............................................. 110
  4.6.6 Extensibility Criteria Assessment ...................................... 111
4.7 OPENJAVA ............................................................................. 113
  4.7.1 Power ............................................................................ 113
  4.7.2 Usability ......................................................................... 114
  4.7.3 Error Handling ............................................................... 114
  4.7.4 Applicability to Benchmarks ............................................. 115
  4.7.5 Extensibility Criteria Assessment ...................................... 115
4.8 MAYA ..................................................................................... 117
  4.8.1 Power ............................................................................ 117
    4.8.1.1 Grammar Productions and Semantic Actions ............... 117
    4.8.1.2 Laziness ................................................................. 117
    4.8.1.3 Overloading Mayans ............................................... 118
  4.8.2 Usability ......................................................................... 118
  4.8.3 Error Handling ............................................................... 119
  4.8.4 Worked Examples ........................................................... 119
    4.8.4.1 Assertions ............................................................. 119
    4.8.4.2 Iteration ............................................................... 120
  4.8.5 Applicability to Benchmarks ............................................. 123
  4.8.6 Extensibility Criteria Assessment ...................................... 124
5 GENESIS LANGUAGE DEFINITION .............................................. 127
  5.1 OVERVIEW .......................................................................... 128
  5.2 DESIGN RATIONALE .......................................................... 129
    5.2.1 Arbitrary Syntax Creation .............................................. 129
    5.2.2 Compile-time Interrogation ............................................ 129
    5.2.3 Base Language ........................................................... 130
    5.2.4 Outward Language Simplicity ....................................... 130
      3.2.4.1 Programmer Support ............................................. 131
      3.2.4.2 Parser Restrictions ............................................... 131
    5.2.5 Inward Language Simplicity .......................................... 131
    5.2.6 Error Reporting .......................................................... 132
  5.3 MACRO DEFINITIONS .......................................................... 133
    5.3.1 Parameters ................................................................... 133
    5.3.2 Precedence ................................................................. 134
      5.3.2.1 Precedence by Grammar Modification .................... 135
    5.3.3 Associativity ............................................................... 136
      5.3.3.1 Associativity by Grammar Modification .................... 136
    5.3.4 Zero Argument Macros ............................................... 136
    5.3.5 Modifiers ...................................................................... 137
    5.3.6 Exceptions .................................................................... 137
    5.3.7 Macro Body ............................................................... 138
    5.3.8 Evaluation Order ........................................................ 138
    5.3.9 Placement and Scope .................................................... 138
    5.3.10 Grammar ................................................................. 139
  5.4 TOKENISING ........................................................................ 140
    5.4.1 Tokenising Approach Overview .................................... 140

• xiii •
TABLE OF CONTENTS

6 REVIEW OF PARSERS ................................................................................................................................. 159

6.1 OVERVIEW ................................................................................................................................................ 160
6.1.1 Extensible Language Parsing .............................................................................................................. 160
6.1.2 Mid-parse Grammar Modification ...................................................................................................... 161
6.2 GRAMMARS ................................................................................................................................................ 163
6.2.1 Grammar Structure .............................................................................................................................. 163
6.2.2 Chomsky Hierarchy .............................................................................................................................. 163
6.2.3 Context-free Grammars ....................................................................................................................... 164
6.2.3.1 Backus-Naur Form ....................................................................................................................... 165
6.2.3.2 Extended BNF ............................................................................................................................ 165
6.2.4 Grammar Properties ........................................................................................................................... 166
6.2.4.1 Ambiguous Grammars ............................................................................................................... 166
6.2.4.2 Left- and Right-recursive Grammars ....................................................................................... 167
6.3 PARSERS .................................................................................................................................................... 168
6.3.1 Derivation ............................................................................................................................................ 168
6.3.2 Parsing Methods ................................................................................................................................. 169
6.3.2.1 Derivation Categorisation .......................................................................................................... 169
6.3.2.2 Lookahead Categorisation ......................................................................................................... 169
6.3.3 Tokenising .......................................................................................................................................... 170
6.3.4 Parse Table Approach ........................................................................................................................ 174
6.3.4.1 Advantages .................................................................................................................................. 174
6.3.4.2 Disadvantages ............................................................................................................................ 177
6.4 TOP-DOWN PARSING ............................................................................................................................... 172
6.4.1 Predictive Parsing ............................................................................................................................... 172
6.4.1.1 Advantages ............................................................................................................................... 173
6.4.1.2 Disadvantages ............................................................................................................................ 173
6.4.2 Parse Table Approach ........................................................................................................................ 174
6.4.2.1 Advantages .................................................................................................................................. 176
6.4.2.2 Disadvantages ............................................................................................................................ 177
6.5 BOTTOM-UP PARSING ............................................................................................................................. 178

• xiv •
TABLE OF CONTENTS

8.6.1.3 Production of Java Source Code ................................................................. 222
8.7 STANDARD LIBRARY EXTENSIONS ................................................................. 223
  8.7.1 Quasi-quotation .............................................................................................. 223
    8.7.1.1 Hygiene ..................................................................................................... 224
    8.7.1.2 Unquoting .................................................................................................. 225
  8.7.2 Macro Definition Shorthands ........................................................................ 226
    8.7.2.1 Automatic Construction Macros .............................................................. 226
    8.7.2.2 Automatic Lists ........................................................................................ 226
    8.7.2.3 Optional Macro Parameters ..................................................................... 227
    8.7.2.4 Statically Type-checked Parameters ......................................................... 229
9 ANALYSIS AND COMPARISON ........................................................................... 231
  9.1 OVERVIEW ........................................................................................................ 232
  9.2 IMPLEMENTATION OF TEST CASES .............................................................. 233
    9.2.1 Assertions ...................................................................................................... 233
      9.2.1.1 Basic Implementation ............................................................................ 233
      9.2.1.2 Quasi-quote Implementation ................................................................. 234
      9.2.1.3 Implementation Issues .......................................................................... 234
    9.2.2 Iteration ......................................................................................................... 234
      9.2.2.1 Basic Implementation ............................................................................ 234
      9.2.2.2 Quasi-quote Implementation ................................................................. 235
      9.2.2.3 Hygienic Implementation ..................................................................... 236
      9.2.2.4 Static-type Matching Implementation ............................................... 236
    9.2.3 Type-safe Formatted Output ........................................................................ 236
      9.2.3.1 Example Expansion ............................................................................ 236
    9.2.4 SQL Subset ................................................................................................... 239
    9.2.5 Generators .................................................................................................... 241
      9.2.5.1 Translation of Formal Parameters ......................................................... 243
      9.2.5.2 Translation of Local Variable Declarations ........................................ 243
      9.2.5.3 Generation of Resumable Code ............................................................ 244
      9.2.5.4 Example Expansions ............................................................................ 244
      9.2.5.5 Explicit Use of GeneratorBase ............................................................. 246
      9.2.5.6 Implementation Issues .......................................................................... 247
    9.2.6 Haskell Subset .............................................................................................. 248
      9.2.6.1 Construction .......................................................................................... 248
      9.2.6.2 Type Abstract Syntax Classes .............................................................. 249
      9.2.6.3 Evaluation ............................................................................................. 251
      9.2.6.4 Type Checking ...................................................................................... 253
      9.2.6.5 Embedded Usage ................................................................................ 253
      9.2.6.6 Standalone Usage ................................................................................ 254
      9.2.6.7 Extended Forms ................................................................................... 255
    9.2.7 Implementation Review ................................................................................ 257
  9.3 QUALITATIVE ASSESSMENT ............................................................................ 261
    9.3.1 Power ........................................................................................................... 261
      9.3.1.1 Benchmark Test Cases ........................................................................ 262
      9.3.1.2 Quasi-quotation Implementation .......................................................... 262
      9.3.1.3 Other Standard Library Macros ............................................................ 262
    9.3.2 Usability ........................................................................................................ 263
    9.3.3 Error Handling ............................................................................................. 264
    9.3.4 Extensibility Criteria Assessment .................................................................. 264
  9.4 MAYA COMPARISON ....................................................................................... 269
    9.4.1 Benchmark Test Cases Comparison ............................................................ 269
      9.4.1.1 Lines of Code Comparison .................................................................. 271
    9.4.2 MultiJava ....................................................................................................... 272
    9.4.3 Extensibility Criteria Comparison Summary ............................................... 273
  9.5 GRAPH EXPANSION PARSING ..................................................................... 277
    9.5.1 Acceptable Grammars .................................................................................. 277
    9.5.2 Efficiency ..................................................................................................... 277
      9.5.2.1 Theoretical Performance ..................................................................... 277
      9.5.2.2 Empirical Results ................................................................................ 278

• xvi •
# Table of Contents

## 10 Conclusion and Future Work

10.1 Conclusion ................................................................................................................................. 284
10.2 Future Work ................................................................................................................................. 286

10.2.1 Flexible Lexical Analysis ....................................................................................................... 286
10.2.2 Delayed Macros ....................................................................................................................... 287
10.2.3 Zero Argument Macros .......................................................................................................... 287
10.2.4 Migration to Java1.5 ............................................................................................................... 288
10.2.5 Parser Efficiency .................................................................................................................. 288
10.2.6 Context-sensitive Graph Expansion Parsing ........................................................................ 289
10.2.7 Integration of Genesis Parsing and Java Compiling ............................................................. 289
10.2.8 Improved Error Tracking ..................................................................................................... 289
10.2.9 Usability Surveys .................................................................................................................. 290
10.2.10 Library Support ................................................................................................................... 290
10.2.11 Improved Embedded Haskell .......................................................................................... 290
10.2.12 Ultimate Aim ...................................................................................................................... 291

## References

## A Genesis Abstract Syntax

A.1 Abstract Syntax Classes .............................................................................................................. 306

## B Genesis and Maya Simple Test Cases

B.1 Assertions ...................................................................................................................................... 314
B.2 Iteration ...................................................................................................................................... 315
B.3 Type-safe Formatted Output ....................................................................................................... 316
## List of Code Examples

Code Example 2.1: Java Threading ................................................................. 21  
Code Example 2.2: Ada Threading (modified from [Bar91§14.4, pp. 291]) ........ 22  
Code Example 2.3: Java Embedded SQL .................................................. 23  
Code Example 2.4: Pro*C Embedded SQL ............................................... 24  
Code Example 2.5: C++ Macro Misuse .................................................... 26  
Code Example 2.6: C++ Macro Parsing Difficulties ............................... 26  
Code Example 2.7: Powerful C++ Macro Construct ................................. 27  
Code Example 2.8: C++ Assertion Macro ............................................... 27  
Code Example 2.9: C++ Iteration Macro ............................................... 28  
Code Example 2.10: Icon Fibonacci Generator ....................................... 29  
Code Example 2.11: Icon Fibonacci Sequence ....................................... 29  
Code Example 2.12: Improved Icon Fibonacci Sequence ....................... 29  
Code Example 2.13: C++ Generator Helper Classes ............................... 30  
Code Example 2.14: Basic C++ Generator Macros .................................. 31  
Code Example 2.15: C++ Fibonacci Generator ....................................... 31  
Code Example 2.16: C++ Fibonacci Generator After Macro Expansion ....... 32  
Code Example 2.17: Extended C++ Generator Macros ........................... 33  
Code Example 2.18: Improved C++ Fibonacci Generator ....................... 33  
Code Example 2.19: C++ Message Map Macro Usage ............................. 34  
Code Example 2.20: C++ Debugging Macro .......................................... 34  
Code Example 2.21: C++ Debugging Macro Usage ................................. 35  
Code Example 2.22: C++ Debugging Macro After Expansion .................. 35  
Code Example 2.23: C++ Template Meta-programming ........................... 36  

Code Example 3.1: Assertions .................................................................. 61  
Code Example 3.2: Java1.5 Assertions ................................................... 62  
Code Example 3.3: Maya Assertions ...................................................... 63  
Code Example 3.4: Iteration .................................................................. 63  
Code Example 3.5: C++ STL Iteration ................................................... 64  
Code Example 3.6: Java1.5 Iteration ...................................................... 64  
Code Example 3.7: Maya Iteration .......................................................... 65  
Code Example 3.8: Typesafe Formatted Output ..................................... 65  
Code Example 3.9: Java1.5 Typesafe Formatted Output ......................... 67  
Code Example 3.10: Embedded SQL Subset ......................................... 68  
Code Example 3.11: Java Generators ..................................................... 69  
Code Example 3.12: Embedded Haskell Subset .................................... 70  
Code Example 3.13: map Function ...................................................... 70  
Code Example 3.14: map Function with Type Signature ....................... 71  
Code Example 3.15: map Function Without Pattern Matching ............... 72  
Code Example 3.16 map Function Using Lambda Functions .................. 72  
Code Example 3.17: List Comprehension Decomposition ....................... 73  

Code Example 4.1: Simple Lisp Program Fragment .................................. 79  
Code Example 4.2: Lisp Quotation Expression for Code Example 4.1 ........ 79  
Code Example 4.3: Lisp Simulated Infix Expressions ............................. 80  
Code Example 4.4: defmacro ............................................................... 80  
Code Example 4.5: define-syntax ......................................................... 81  
Code Example 4.6: define-syntax General Form ................................... 81  
Code Example 4.7: Quasi-quote and Unquote ....................................... 82  
Code Example 4.8: swap Function ....................................................... 82  
Code Example 4.9: Improved swap Function ....................................... 83  
Code Example 4.10: Hygienic swap Function ....................................... 83  
Code Example 4.11: swap Function using syntax-case ......................... 84  
Code Example 4.12: swap Function with Error Handling ..................... 85  
Code Example 4.13: Template Haskell printf Expansion ...................... 90  
Code Example 4.14: Template Haskell Assertions .................................. 91
<table>
<thead>
<tr>
<th>Code Example</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.12</td>
<td>Generator Method Implementation</td>
<td>242</td>
</tr>
<tr>
<td>9.13</td>
<td>Translation of Formal Parameters</td>
<td>243</td>
</tr>
<tr>
<td>9.14</td>
<td>Translation of Local Variable Declarations</td>
<td>243</td>
</tr>
<tr>
<td>9.15</td>
<td>Repeating Generator Expansion</td>
<td>244</td>
</tr>
<tr>
<td>9.16</td>
<td>Fibonacci Generator Expansion</td>
<td>246</td>
</tr>
<tr>
<td>9.17</td>
<td>Sub-sequence Generator Expansion and Use</td>
<td>247</td>
</tr>
<tr>
<td>9.18</td>
<td>FunObject Interface</td>
<td>248</td>
</tr>
<tr>
<td>9.19</td>
<td>Haskell Subset Cons Abstract Syntax Class</td>
<td>249</td>
</tr>
<tr>
<td>9.20</td>
<td>Type Abstract Syntax Classes for the Haskell Subset</td>
<td>250</td>
</tr>
<tr>
<td>9.21</td>
<td>Evaluation of Function Application</td>
<td>252</td>
</tr>
<tr>
<td>9.22</td>
<td>Haskell Subset if Expression</td>
<td>253</td>
</tr>
<tr>
<td>9.23</td>
<td>Embedded Haskell Wrapper Definition</td>
<td>254</td>
</tr>
<tr>
<td>9.24</td>
<td>Standalone Haskell Module</td>
<td>254</td>
</tr>
<tr>
<td>9.25</td>
<td>Standalone Haskell main Method</td>
<td>255</td>
</tr>
<tr>
<td>9.26</td>
<td>Function Declarations Definition</td>
<td>256</td>
</tr>
<tr>
<td>9.27</td>
<td>Operator Currying Definition</td>
<td>256</td>
</tr>
<tr>
<td>9.28</td>
<td>Simple Single Source List Comprehensions</td>
<td>256</td>
</tr>
<tr>
<td>9.29</td>
<td>Multiple Condition Single Source List Comprehensions</td>
<td>257</td>
</tr>
<tr>
<td>10.1</td>
<td>Extended Tokeniser Possibility</td>
<td>286</td>
</tr>
<tr>
<td>10.2</td>
<td>Macro Expansion RequiringDelayed Macros</td>
<td>287</td>
</tr>
<tr>
<td>A.1</td>
<td>Compilation Unit Classes</td>
<td>304</td>
</tr>
<tr>
<td>A.2</td>
<td>Type Classes</td>
<td>305</td>
</tr>
<tr>
<td>A.3</td>
<td>Statement Classes</td>
<td>305</td>
</tr>
<tr>
<td>A.4</td>
<td>Variable Declaration Classes</td>
<td>306</td>
</tr>
<tr>
<td>A.5</td>
<td>Method and Type Declaration Classes</td>
<td>307</td>
</tr>
<tr>
<td>A.6</td>
<td>Expression Classes</td>
<td>308</td>
</tr>
<tr>
<td>A.7</td>
<td>Identifier Classes</td>
<td>309</td>
</tr>
<tr>
<td>A.8</td>
<td>Literal Classes</td>
<td>309</td>
</tr>
<tr>
<td>B.1</td>
<td>Genesis Assertion Definition</td>
<td>314</td>
</tr>
<tr>
<td>B.2</td>
<td>Maya Assertion Definition</td>
<td>314</td>
</tr>
<tr>
<td>B.3</td>
<td>Genesis Iterator Definition</td>
<td>315</td>
</tr>
<tr>
<td>B.4</td>
<td>Maya Iterator Definition</td>
<td>315</td>
</tr>
<tr>
<td>B.5</td>
<td>Genesis Type-safe Formatted Output Definition</td>
<td>316</td>
</tr>
<tr>
<td>B.6</td>
<td>Maya Type-safe Formatted Output Definition</td>
<td>316</td>
</tr>
</tbody>
</table>
List of Figures

Figure 3.1: SQL Subset Grammar .................................................................................................. 68
Figure 3.2: Haskell Subset Grammar ............................................................................................ 73
Figure 3.3: Haskell Embedding Grammar ....................................................................................... 74
Figure 4.1: Lisp S-expression for Code Example 4.1 ........................................................................ 79
Figure 4.2: JSE Call and Statement Macro Grammars ..................................................................... 106
Figure 5.1: Expression Grammar .................................................................................................... 135
Figure 5.2: Method Definition Grammar ......................................................................................... 137
Figure 5.3: Genesis Grammar .......................................................................................................... 139
Figure 5.4: Simple Tokenising .......................................................................................................... 141
Figure 5.5: Multi-character Symbol Grouping Tokenising .............................................................. 141
Figure 5.6: Single-character Symbol Output Tokenising ................................................................. 142
Figure 5.7: Multi-character Symbol Combinations Tokenising ...................................................... 143
Figure 5.8: Tokeniser Transition Diagram ....................................................................................... 145
Figure 5.9: Same Sub-rule Precedence ............................................................................................. 151
Figure 5.10: Different Sub-rule Precedence ..................................................................................... 152
Figure 5.11: Associativity ................................................................................................................ 152
Figure 5.12: Partial Abstract Syntax Type Hierarchy ......................................................................... 156
Figure 5.13: Exception Type Hierarchy ............................................................................................ 157
Figure 6.1: Generalised Grammar Rule ............................................................................................ 163
Figure 6.2: BNF Context-free Grammar ........................................................................................... 165
Figure 6.3: Ambiguous Simple Expression Grammar ....................................................................... 166
Figure 6.4: Unambiguous Simple Expression Grammar ................................................................. 166
Figure 6.5: Non Left-recursive Simple Expression Grammar ........................................................... 167
Figure 6.6: High-level Compiler Model ............................................................................................ 168
Figure 6.7: Derivations ..................................................................................................................... 169
Figure 6.8: Non-ambiguous Unparsable Grammars .......................................................................... 170
Figure 6.9: Deterministic Finite Automaton ..................................................................................... 171
Figure 6.10: Table-driven Predictive Parser Model ......................................................................... 174
Figure 6.11: Shift-reduce Parser Reductions ...................................................................................... 178
Figure 6.12: Shift-reduce Parser Model ............................................................................................. 179
Figure 6.13: Operator-precedence Parser Model .............................................................................. 180
Figure 6.14: LR Parser Model ......................................................................................................... 182
Figure 6.15: Expression Grammar in Chomsky Normal Form .......................................................... 191
Figure 6.16: CYK Expression Parsing Recognition Table ................................................................. 191
Figure 6.17: Chart Parser Representation ......................................................................................... 193
Figure 7.1: Multipass Graph Expansion ........................................................................................... 200
Figure 7.2: Single Pass Graph Expansion .......................................................................................... 202
Figure 7.3: Simple Expression Partial Match Tree ............................................................................ 205
Figure 7.4: Dangling-else Partial Match Tree .................................................................................... 205
Figure 8.1: Genesis Compiler Structure ........................................................................................... 208
Figure 8.2: Precedence Mangling Examples ..................................................................................... 213
Figure 8.3: Mangled Name Grammar ............................................................................................... 214
Figure 8.4: Name Mangling Algorithm ............................................................................................. 214
Figure 8.5: Graph Produced From Tokenising Code Example 8.6 .................................................... 216
Figure 8.6: Graph Produced From Tokenising Code Example 8.7 .................................................... 217
Figure 8.7: Method Declaration Fragment ......................................................................................... 227
Figure 9.1: Differing Time Complexities of Earley’s Algorithm ........................................................ 278
Figure A.1: High-level Abstract Syntax Class Hierarchy ................................................................. 306
Figure A.2: Declaration Abstract Syntax Class Hierarchy ................................................................. 306
Figure A.3: Expression Abstract Syntax Class Hierarchy ................................................................. 308
List of Tables

Table 3.1: Criteria for rating an Extensible Language’s Power ..........................................................56
Table 3.2: Criteria for rating an Extensible Language’s Usability ......................................................57
Table 3.3: Criteria for rating an Extensible Language’s Error Handling ............................................58
Table 3.4: Benchmark Test Cases Summary ........................................................................................60
Table 4.1: Lisp Applicability to Benchmark Test Suite ........................................................................85
Table 4.2: Lisp Extensibility Criteria Assessment .................................................................................86
Table 4.3: Template Haskell Applicability to Benchmark Test Suite .....................................................95
Table 4.4: Template Haskell Extensibility Criteria Assessment ............................................................95
Table 4.5: MS$^2$ Applicability to Benchmark Test Suite .......................................................................102
Table 4.6: MS$^2$ Extensibility Criteria Assessment ...............................................................................103
Table 4.7: JSE Applicability to Benchmark Test Suite ..........................................................................110
Table 4.8: JSE Extensibility Criteria Assessment ................................................................................111
Table 4.9: OpenJava Applicability to Benchmark Test Suite .................................................................115
Table 4.10: OpenJava Extensibility Criteria Assessment .......................................................................115
Table 4.11: Maya Applicability to Benchmark Test Suite .....................................................................123
Table 4.12: Maya Extensibility Criteria Assessment ..............................................................................124
Table 6.1: Chomsky Hierarchy ...............................................................................................................163
Table 6.2: Simple Expression LL Parse Table ......................................................................................176
Table 6.3: Simple Expression Operator-precedence Table ..................................................................182
Table 6.4: Simple Expression LR Parse Table ....................................................................................185
Table 6.5: Simple Expression LALR Parse Table ................................................................................186
Table 7.1: Single Pass Graph Expansion Evaluation ..........................................................................203
Table 7.2: Final Algorithm Graph Expansion Evaluation ....................................................................205
Table 9.1: Rules for Function Type Equality .......................................................................................250
Table 9.2: Genesis Applicability to Benchmark Test Suite .................................................................257
Table 9.3: Genesis Extensibility Criteria Assessment ........................................................................264
Table 9.4: Genesis and Maya Benchmark Test Suite Comparison .....................................................269
Table 9.5: Genesis and Maya Lines of Code Comparison ...................................................................271
Table 9.6: Genesis and Maya Extensibility Criteria Comparison ........................................................273
Table 9.7: Earley Versus GEP Time Complexity ................................................................................279
Table 9.8: Earley Versus GEP Comparison ........................................................................................281
Introduction

1. Introduction
2. Defining extensibility
3. Assessing extensibility
4. Reviewing extensibility
5. Designing the language
6. Reviewing parsing
7. Implementing a parser
8. Language implementation
9. Evaluation
10. Conclusion

CHAPTER 1
1.1 Overview

Extensible programming languages allow users to create fundamentally new syntax and translate this syntax into language primitives. The concept of compile-time meta-programming has been around for decades. It originally appeared in the Lisp [Ste90, Dyb03] community and similar approaches have been attempted in other languages. For example, extensions have been attempted for Haskell [PJ99] and C [KR78] (Template Haskell [SPJ02] and MS2 [WC93] respectively). There is also a limited meta-programming facility in C++ [Str91]. Typically, systems that provide such abilities generally disallow the creation of new syntactic forms, or have heavy restrictions on how, or where, this may be done. Such restrictions severely limit the forms that can be expressed in such languages.

There have been many attempts of providing meta-programming facilities for Java (eg. [BP01, TCKI00, Bak01]) which are reviewed (along with those of Lisp, C, and Haskell) in Chapter 4.

Genesis is an extension to Java that supports compile-time meta-programming and provides further support by allowing users to create their own arbitrary syntax. This is achieved through macros that operate on a mix of both concrete and abstract syntax, and produce abstract syntax. Genesis provides a minimal design whilst maintaining, and extending, the expressive power of other similar macro systems.

The core Genesis language definition lacks many of the desirable features found in other systems, such as quasi-quote, hygiene, and static expression-type dispatch. Unlike other systems however, Genesis is expressive enough to define these as syntax extensions (see section 8.6).

Like most such systems, Genesis’ user-defined macros produce only well-formed syntactic structures via the use of a predefined set of classes (see subsection 5.6.1) that define a Java abstract syntax. Programmers are free to extend and add to this set of classes to create their own abstract syntax and to define macros that specify translations in pure Java syntax.

At the heart of Genesis is a flexible parser (see Chapter 7) that is capable of parsing any context-free grammars — even ambiguous ones. The parser is capable of arbitrary speculation and will consider all possible parses. The parser constructs a graph of
possible paths, and is capable of dynamically pruning this graph, or combining nodes, based on precedence or associativity rules. This general parser allows macro programmers to forget about parsing, and concentrate on defining new syntax. This is viewed as essential to providing a usable system that allows arbitrary syntax creation.

Indeed, one key goal of this system was to address the programmer’s learning curve by providing as simple a system as possible (see section 5.2 for further rationale behind the design). This was achieved by the use of the flexible parser, the introduction of only one new construct to standard Java, and extensions to make programming macros more user friendly.

The expressiveness of Genesis is wide ranging: it is capable of providing small scale limited use macros, large scale semantic modifications, through to complete language replacements. Great care was taken not to limit the scope of applicability of Genesis’ macros.

To demonstrate this expressiveness, we implement many of the simple test cases found in other systems, such as a type-safe `printf`, assertions, and iteration statements (see subsection 3.4.1 for descriptions of these test cases and section 9.2 for their implementations). These test cases require an ability to perform static type-checking and to manipulate compile-time values and abstract syntax trees. Whilst these test cases require sophisticated facilities for implementation, they are still limited in scope.

As further examples of the expressive power of Genesis we also provide implementations of embedded subsets of SQL and Haskell (see subsection 3.4.2 for their description and section 9.2 for their implementations). To provide a direct comparison with Maya (the most comparable of the Java extensions) the possibility of MultiJava implementation is discussed in subsection 9.4.2. In addition to these, a generator function (similar to that of Icon [GG90]) implementation is also provided (see subsections 3.4.2.2 and 9.2.5). These extensions require much more sophisticated syntax creation and manipulation facilities than simple statement macros.

As a final proof of power, the Haskell subset can operate as a stand-alone extension independent of any recognisable Java code (see section 9.2.6.6).

### 1.1.1 Assessment of Success

The successfulness of this work will be assessed by three factors:
• Qualitative evaluation against a developed list of criteria for a “good” extensible language (see section 3.3). Other extensible languages will also be rated by these criteria also (see chapter 4).

• Implementation of the benchmark test cases from section 3.4. These test cases are carefully chosen to illustrate the ability to add simple confined constructs, to add outwardly simple but inwardly sophisticated constructs, and to make wholesale changes to the syntax and semantics of the language.

• Direct comparison with implementations of examples from Maya, which, as we shall see in section 4.8, is the most comparable of those languages from previous research. Code length can be used as a real metric of comparison (brevity is one of the criteria identified in section 3.3). Maya and Genesis are also compared across the full range of these criteria.

1.1.2 Conventions

Throughout this work a number of conventions are used for the layout of code examples, grammars, and internet addresses.

Code appearing within text is emphasised by the use of a fixed width font. Code Examples are also in a fixed width font, and use the following conventions:

• Keywords (or equivalent) appear in boldface, eg. `while`.

• Indentation distance is two spaces.

• Gaps in code where effectively any form is appropriate is specified by ellipses, i.e. “…”.

Context-free grammars appear in a fixed width font and use the following conventions:

• Terminals appear in boldface, eg. `while`, `++`, etc.

• Productions have a non-terminal left-hand side, followed by `::=`, and an Extended BNF right-hand side.

• Square brackets surround optional elements.

• A plus symbol indicates one-or-more of the preceding symbol, eg. `identifier+`.

• An asterisk indicates zero-or-more of the preceding symbol, eg. `identifier*`.
Parentheses group symbols and work in conjunction with other EBNF forms, eg. for a list of comma separated identifiers: \texttt{identifier \ (, \ identifier)+}

In-text references are in abbreviated author/date form, eg. [Doe99].

Internet addresses appear as underlined URLs, eg. \url{http://somewhere.com/somefile.html}. 
1.2 Sources of Code Examples

In this work, example code is drawn from a large variety of programming languages, either for introduction of the concepts involved, or for direct comparison purposes. Most of these languages are mainstream, while others, due to their research nature, are not. In order to facilitate the accessibility of these later code examples, all of these languages are briefly introduced. These introductions include (where available) references to their formal definitions (or closest approximation), downloadable versions of these definitions for quick reference, and language homepages.

1.2.1 Java

Java is the implementation language for this work, but “Java” has continued to be redefined, and as a result two distinct flavours of Java are discussed. The version numbers of Java are confusing at best, so it is necessary to define precisely which are meant. In particular we are not concerned with versions of the Java Virtual Machine (JVM), the Java SDK, or any other auxiliary package/facility. In this context the important revisions are those that are made to the language definition and as a result to the compiler.

1.2.1.1 Java1.4

Java1.4 (a.k.a. Java 2 Standard Edition 1.4) had only one minor addition to the language: the introduction of a new statement for handling assertions.

This work describes an extension of Java1.4, and the implementation is in Java1.4.


Homepage: http://java.sun.com/
1.2.1.2 Java1.5

Java1.5 is the first revision to provide support for generics (also referred to as: templates, parametric polymorphism, etc.) and has a host of other new features, eg. an enhanced for loop, automatic boxing/unboxing, typesafe enumerations, variable length argument lists, static imports, and embedded class metadata.


See also [Bra99] and [Bra02] for further information.

**Homepage:** [http://java.sun.com/](http://java.sun.com/)

### 1.2.2 MultiJava

MultiJava is a Java extension that adds open classes and symmetric multiple dispatch (multimethods). Open classes allow the addition of methods to existing classes without modifying the original class, and multimethods provide run-time polymorphic dispatch on all of the arguments of a method, not just the first.


**Homepage:** [http://multijava.sourceforge.net/](http://multijava.sourceforge.net/)

### 1.2.3 Maya

Maya is a Java extension that allows for new syntax creation. It permits both abstract and concrete syntax extension and has a novel approach of lazily interleaving type-checking and parsing.

---

1 Java1.5 is the revised Java language definition included as part of the (increasingly confusingly named) Java 5 platform.


Homepage: http://www.cs.utah.edu/~jbaker/maya/

### 1.2.4 Java Syntax Extender

JSE is a Java extension that provides meta-programming support.


Homepage: http://people.csail.mit.edu/people/jrb/jse/index.htm

### 1.2.5 OpenJava

OpenJava is a Java extension that provides for class-based meta-programming support.

**Language specification:** [Tat99] Michiaki Tatsubori: *An Extension Mechanism for the Java Language*, Master’s Thesis, University of Tsukuba, February 1999. Available from:


Homepage: http://www.csg.is.titech.ac.jp/openjava/

### 1.2.6 Haskell

Haskell is a general purpose purely functional language. Part of the motivation for the development of Pizza [OW97] came from a desire to provide some facilities found in languages like Haskell.


Homepage: http://www.haskell.org/
1.2.7 Template Haskell

Template Haskell is a meta-programming extension of the Haskell language.


Homepage: http://www.haskell.org/th/

1.2.8 Lisp

Lisp (an acronym for list processing) [Mcc60] is one of the first computer programming languages. It first appeared in 1959, which is within a few years of the time of early Fortran [For77]. It was originally an interpreted functional language. It is one of the earliest programming languages still widely used.

1.2.8.1 Common Lisp

The Lisp ANSI standardisation process was started in the early 1980s, and lead to Common Lisp in the early 1990s.


1.2.8.2 Scheme

Scheme is a statically scoped dialect of Lisp that was the first language to provide hygienic macros.


Homepage: http://www.swiss.ai.mit.edu/projects/scheme/
1.2.9 C++

C++ is a multi-paradigm language that was built upon the C language [KR78]. Its major additions to C include improved data abstraction, object-orientation, generic programming, and exception handling.

Please note that while mentioned in the text, the code examples do not specifically use the C language.


1.2.10 MS²

MS² is a meta-programming extension of the C language.


1.2.11 SQL

SQL (Structured Query Language) is a database query language that has almost universal adoption.


**Homepage:** http://sql.org

1.2.12 Pro*C

Pro*C is an Oracle precompiler that adds embedded SQL support to C/C++. All embedded SQL statements are translated to normal C/C++ functions and the resultant program is compiled by a regular compiler.

Corporation, June 1997. Available from:
http://www-rohan.sdsu.edu/doc/oracle/server803/A54661_01/toc.htm

### 1.2.13 Icon

Icon is a high-level language that treats all functions as limited co-routines with a concept of success or failure (called *Generators*). Unlike traditional call-return semantics, Icon Generators may return multiple (perhaps infinite) independent values.


**Homepage:** [http://www.cs.arizona.edu/icon/](http://www.cs.arizona.edu/icon/)

### 1.2.14 Ada

Ada was a result of a ten year design process by the United States Department of Defense in an effort to reduce expenditure by providing a single common programming base. Ada was first ratified as a standard in 1983 by ANSI, and later by ISO in 1987. It was subject to revision in 1995.


**Homepage:** [http://www.adahome.com/](http://www.adahome.com/)
1.3 Thesis Structure

The basic structure of this thesis is as follows:

- a review of previous work is conducted;
- a new language is proposed;
- along the path to implementation, a review of parser techniques is conducted;
- as a result of this review, a new general parser is proposed;
- using this parser, the language is implemented; and
- finally, the success of this new language is examined.

At the beginning of each chapter, a document map is provided with the current chapter highlighted to remind the reader of their current position in this work. A reader well-versed in parsing may choose to defer reading of the pre-implementation parser review and new parser design (chapters 6 and 7) and continue straight to the implementation of the Genesis language (chapter 8).

A chapter-by-chapter breakdown of the structure is as follows:

- In Chapter 2, a definition of the term extensible is given, uses of this term in the field of programming languages are examined, and a comprehensive review behind the reasons in providing an extensible system are provided.
- In Chapter 3, we provide criteria, and a suite of benchmark test cases in order to rate the success of this work and other extensible languages.
- In Chapter 4, an in-depth review of the most applicable languages from Chapter 2 is conducted — few of these languages are truly extensible, as per the definition in Chapter 2.
- In Chapter 5, the definition of the Genesis language developed in this work is provided. This definition consists of the language grammar and semantics.
- In Chapter 6, a review of parser techniques is conducted, along with an examination of their applicability to the language defined in Chapter 4.
- In Chapter 7, a new parsing technique, Graph Expansion Parsing, is introduced and defined. This chapter explains the motivation for this technique, its development, and the optimisations performed on the algorithm.
- In Chapter 8, the full implementation of a Genesis compiler is described, this compiler has the parser from Chapter 6 at its core.
• In Chapter 9, a comparison of the efficiency of the parser to other techniques is given, the implementation of the test examples outlined in Chapter 3 is provided, Genesis is rated against the criteria specified in Chapter 3, and comparisons to other similar work are provided.

• In Chapter 10, the conclusions of this work are drawn, and possibilities for future research are outlined.
CHAPTER

Extensibility

1. introduction
2. defining extensibility
3. assessing extensibility
4. reviewing extensibility
5. designing the language
6. reviewing parsing
7. implementing a parser
8. implementing the language
9. evaluation
10. conclusion
2.1 Overview

Before it is possible to review previous related work, it is necessary to have an exact understanding of what extensibility means in the field of programming languages. In the beginning of this chapter (section 2.2), we define precisely what extensibility means in the context of this work and look at the variety of areas in which the term extensibility is used (section 2.3). Uses of the term extensible differ markedly from source to source.

To review those languages which are considered to satisfy our definition of extensibility, we first discuss why extensibility is a desirable property for a programming language. This is done in section 2.4.
2.2 Defining Extensible

The term “extensible” is widely used in a variety of contexts. It is necessary to precisely define its meaning in this work and to look at some examples of its (mis)use in the field of programming languages.

2.2.1 Other Similar Terminology

Throughout this work the term extensible is used; however other works may refer to an equivalent concept as meta-programming, or merely speak of compile-time evaluation. As we shall see, both encompass the idea of extensibility partially in that they allow a user to write programs that themselves write programs, but extensibility concerns itself with more than just this meta-programming.

In essence, meta-programming differs from extensibility only in that programmers are given the ability to give their meta-programs any syntax they choose.

2.2.2 Previous Extensibility Definitions

In his work arguing for the necessity of extensible languages, Gregory Wilson split the definition of extensible:

“A syntactically extensible language allows programmers to define new forms by specifying what the new syntax looks like, and how it maps back to the language’s primitives.”

“A semantically extensible language allows programmers to define entirely new kinds of operations, or to change the behavior of built-in ones.” [Wil04]

These two definitions seem appropriate within the context of this work, but the following statement introduces doubt as to the exact meaning of these definitions:

“C macros and C++ operator overloading are probably the most familiar examples of each kind of extensibility, although both are severely restricted.” [Wil04]

It is unclear how C/C++ macros can be said to *extend* the syntax as they are so severely restricted in their use as to appear as library calls, or as identifiers (if being used with no arguments). Also, it would seem that they allow for semantic extensibility because it is
possibly to modify the default behaviour of many language constructs (see section 2.3.3 for a more in-depth discussion, and particularly section 2.3.3.5, for an example).

Also, C++ operator overloading satisfies only the second part of the definition of semantic extensibility, as it is possible to overload the built-in operators, but not possible to create new ones. However, operator overloading can satisfy the first part of the definition: Haskell [PJ99] provides such facilities.

In the description of the language Maya, extensibility is explained loosely by:

“Syntax extension can be used to embed a domain-specific language within an existing language” [BH02§1, pp. 1]

This explanation alone implies that extensibility is only useful for projects that fundamentally change the original language. While useful, language embeddings are only one of desirable capabilities of extensible languages. In addition, extensibility can be employed to add minor features perhaps very specifically targeted. Indeed, many of the examples that follow in the Maya work are of far smaller macros.

In the description of the language Template Haskell, meta-programming, a key aspect of extensibility, is defined as follows:

“The purpose of [meta-programming] is to allow programmers to compute some parts of their program rather than write them, and to do so seamlessly and conveniently.” [SPJ02§1, pp. 1]

This definition says that programs can be written that write programs, and also that these programs should be simple to construct and should be written in the same language as ordinary programs. Much of this definition concerns itself with matters of language quality that should not be confused with any definition of the term extensible — it should be clear that a language need not be convenient before it can be considered extensible.

### 2.2.3 Definition of Extensible

The definition of extensible used for the purposes of this work, and strictly within the domain of programming language design can be stated simply as:

The term extensible can be applied to a programming language that provides constructs that allow for the creation of new syntax and semantics.

• 18 •
It follows from this definition that:

An extensible programming language is one that allows for the creation of new syntax and semantics.

Meta-programming facilities are considered a requirement of this definition. Although it may be possible to define a programming language that was extensible but does not allow for meta-programming we will not concern ourselves with such possibilities.

This seemingly simple definition has some heavyweight casualties — as we shall see in section 4.2 even Scheme does not allow the programmer to create new syntax.

Other definitions of extensible tend to allude to particular abilities of any given extensible language, but these can be viewed as merely a measure of the language’s quality. Such properties include, but are not limited to:

- the embedding of another domain-specific language within itself;
- the creation of small syntax additions of limited scope;
- overriding the behaviour of built-in syntax; and
- creation of optimisations.

A comprehensive set of criteria for rating extensible languages appears in section 3.3.
2.3 Language Extension Mechanisms

This section introduces systems that either claim extensibility via various means, or attempt to provide an extension mechanism that would not fall under our definition of extensible from section 2.2.3. These range from simple libraries, through to open compilers, and even languages that are extensible but not via first-class language constructs.

The systems examined here demonstrate what is possible with traditional systems, and illustrates the various work-around approaches taken due to their lack of extensibility. These systems are reviewed here in increasing order of relevance, with the final subsection (2.3.5) merely introducing languages that are examined further in chapter 4.

2.3.1 Library Systems

We regard extensions provided through the use of the normal library mechanism, to not fit the definition of extensible. While libraries are important, and features may be added to the language via them, it is only in a very rigid and constricting form that must still fit the syntax of the original language exactly. As a result many things that may be desirable are still impossible. Other constructs, whilst possible, are not easy to implement, or must be implemented in an undesirable form — an often seen example is that SQL support is generally provided by forcing all queries to be written as strings (see subsection 2.3.1.1).

It is possible to provide new facilities through traditional libraries, but never new syntax or semantics. Library mechanisms can only use the primitives of the base language, be they classes, procedures, functions, or some other form of definition. All such primitives have a rigid syntax which is unalterable.

The major advantage of providing extra functionality through libraries is that the facilities are (ideally) available to all programmers — Java’s threading support shows how seamlessly this can be achieved\(^2\) (see section 2.3.1.1).

\(^2\) Although Java does rely on the use of `synchronized` in order to implement many features.
The examples provided here are typical of the kind of facilities provided through library systems, and in particular we look at how this support can rarely match the power and flexibility of direct language support.

### 2.3.1.1 Threading

A typical example of where facilities are provided through libraries is with many threading mechanisms.

C/Unix systems provide a host of different libraries to enable threading (eg. POSIX threads [Mue93, DM05] and UI threads [Nor96]). C++ systems on most platforms have similar approaches (e.g. Boost Threads [Gur04§9]).

Java’s threading mechanism (see Code Example 2.1), whilst not overly different to those of other languages (especially those of C++), is provided through a class that is part of Java’s Development Kit (JDK) [GJSB00§17.12], and is, as a result of being standard, available to all Java programmers. This universality results in the belief that threading is part of Java, even though it is only a feature of its standard libraries.

```java
class AtTheSameTime extends Thread {
    public void run() {
        // implementation goes here
    }
}
```

**Code Example 2.1: Java Threading**

Facilities such as inter-thread communication are left very much up to the programmer, Java does provide a synchronized keyword so that the programmer is not required to program mutual exclusion locks, but no other direct support is given.

**Improved Approach**

In stark contrast to Java are languages such as Ada [Uni83] which provide direct support for threads through syntactic constructs, they provide functionality generally not provided in library systems - either due to complexity of implementation or sheer impossibility. Synchronisation of tasks is provided through a mechanism known as a *rendezvous*. Multiple rendezvous attempts are handled through an extended version of Ada’s `select` statement [Bar 91§14]. Ada allows programmers more power than the equivalent library based constructs.
Java’s use of the `synchronized` keyword on variables can be expressed in Ada as shown in Code Example 2.2. This example defines a task that has two entry points: `READ` and `WRITE`. This task will automatically execute when the program starts. The body of the task stores insists that the first allowed access to the variable is a write (read attempts will just be queued until at least one write has completed), it then allows either read or write access in any order, but ensures that only one access to the variable can take place at any given time. This is a simple version that implements only a single protected variable of a fixed type, and Ada is expressive enough to remedy both of these flaws, but not quite expressive enough to provide access to the variable without function calls.

Ada provides for much more powerful uses of these constructs, such as task types, and timed and conditional rendezvous. Most of these features are impossible to provide in such a convenient form in other languages that support threading via library mechanisms.

### 2.3.1.2 SQL

SQL [ANS92] is the language of choice for communicating with databases. For this reason, most mainstream languages have SQL libraries available (e.g., Java’s JDBC/SQL [EH01] and Visual Basic’s ODBC [Mic05b]). SQL support in such languages is
typically provided through allowing programmers to specify SQL queries in strings which are then passed to a SQL library for evaluation.

There are two notable aspects of this: the language is providing no real direct support for SQL (support for any language can be provided through parsing embedded strings), and syntax-checking must be performed at run-time. A SQL interpreter is provided within the library to check the validity of any given SQL query, and to execute once validated. Whilst this approach is highly desirable in some circumstances that require true run-time SQL creation, in those that don’t, it forces the programmer into an effective double-compilation situation.

```java
// give all dynamic qualities a value, but normally this kind of SQL would // appear inside a function wrapper
String tableSource = "Movies";
String searchField = "Title";
String searchString = "Star";
String sortField = "Year";

Connection connection = null;
// perform some database connection code and hopefully succeed
try {
    Statement stmt = connection.createStatement();
    ResultSet resultSet = stmt.executeQuery("SELECT * FROM " + tableSource + " WHERE " + searchField + " LIKE /*" + searchString + "*/" + " ORDER BY " + sortField);
    // now pull apart the ResultSet object and do something useful
} catch (SQLException e) {
    // do something useful with error information
}
```

**Code Example 2.3: Java Embedded SQL**

The example in Code Example 2.3 shows typical embedded SQL in Java. The obvious inelegance of this approach compared to the three line SQL query it represents is immediately apparent from this example. Notice that the table to search, the field to search for, what the field is supposed to be “like”, and how the presentation order are all dynamic qualities. However, what is particularly important to note here, is that this is not truly dynamic SQL, as the structure of the SQL query never changes.

**Improved Approach**

Ideally, SQL queries would be syntax checked at compile time, and would actually be expressed directly in the base language – and indeed that is the approach taken in many languages. For such languages, such facilities are part of their sales pitch. In fact,
embedded SQL is common enough for the ISO SQL standard to specify how it should be done [ANS89].

Code Example 2.4 demonstrates embedded ISO SQL in Pro*C. In this example, the colon is necessary to distinguish `a` from a database identifier. Pro*C allows fixed structure queries to be mixed with variables from the surrounding code but does not allow tables or fields to be specified dynamically within such queries. A more complete embedded SQL implementation would suffer from no such restrictions.

```c
int a;
int taxFileNumber = 876543210;
EXEC SQL SELECT Salary INTO :a
     FROM Employee
     WHERE TFN=:taxFileNumber;
printf("The salary is %d\n", a);
```

Code Example 2.4: Pro*C Embedded SQL

### 2.3.1.3 Summary

As we have seen, language extensions via library definition will almost certainly lead to imperfect extensions due to the necessity of compromises forced by the library mechanism. Direct language modification always provides more power; the major argument against adding direct language support is that of complexity.

### 2.3.2 Open Compilers

Any language that claims to be extensible by merely opening the source of the compiler is another example of misuse of the term extensible as it has been defined in this work. Any compiler for which the source is available can be extended, and hence so too the language it compiles. This, however, does not provide extensibility in any truly useful way – any new features force a new compiler to be provided to all that wish to use them. If two modifications are made independently there is no way for these to be reconciled.

When implementing new features in this way, great care must be taken in order to retain the robustness of the compiler. Future revisions only serve to make this task more arduous.
The major advantage of this approach is that with full compiler source, any modification is possible: the language can be altered in any way, from minor changes to entirely new typing systems.

Although not developed with the aid of an open source compiler, the language Pizza made such wholesale changes to Java’s type system, and in part to the development of the Java 1.5 specification.

### 2.3.2.1 Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) [GHC02] is an open-source compiler for which many extensions have been made for research purposes. Examples of such extensions include Concurrent Haskell [MPJT04], Parallel Haskell [TPL01] and Template Haskell.

Template Haskell was considered interesting enough that its implementation has been added into GHC as an extension. Indeed, Template Haskell satisfies many of our properties of an extensible language, and is described further in section 4.3.

### 2.3.2.2 Summary

Open compilers facilitate the modification of a language as it is easier for people to realise extension ideas they may have. In section 2.4.2 we shall further examples of how in the most successful cases, such modification has led to adoption within the evolving original language.

### 2.3.3 Text Macros

This subsection gives us our first look at meta-programming facilities, albeit in a far-from-perfect form. In order to demonstrate their usefulness, a large number of examples are provided, the bulk of which are revisited in later sections, and actually serve as part of the benchmark test suite (see section 3.4).

Text macros (or token macros), were the earliest form of macro system [WC93], their development followed close behind that of assembly language.

> “[Text] Macro systems support a limited form of syntax extension. In most systems, a macro call consists of the macro name followed by zero or more arguments.” [Bak01§1, pp. 1]
Macro systems, such as those found in C++ [Str00§7.8] provide direct manipulation of the source text/tokens. This manipulation is via rigid macro syntax (which may be somewhat different from the language’s normal syntax), and is unable to provide any error checking.

The classic example of macro misuse is demonstrated in Code Example 2.5. Unlike a function call, \(10+10\) is not evaluated before the macro is applied, so the result of \(\text{sqr}(10+10)\) is actually \(10+10\times10+10\) which, due to precedence, evaluates to \(120\). In this case the macro can be rewritten with liberal use of parentheses to ensure correct evaluation for simple arguments. However, nothing can be done to ensure intuitiveness if the argument to \(\text{sqr}\) contains side-effects.

```cpp
#define sqr(x) x*x
...
sqr(10+10)
```

**Code Example 2.5: C++ Macro Misuse**

In more pathological cases, macros can appear to the programmer as normal library extensions, but can break if certain arguments are supplied. The rather contrived example in Code Example 2.6 will fail due to the fact that the template call \(\text{sum}<3,5>()\) is not resolved when the macro is expanded, and as a result the stream of tokens appears to contain two arguments rather than one. This problem can be resolved by bracketing the expression on the calling side. For a user, this situation is intolerable and may be many calls deep within macro code.

```cpp
template<int a, int b> sum() { return a + b; }
#define negate(x) -x
int i = negate(sum<3,5>());
```

**Code Example 2.6: C++ Macro Parsing Difficulties**

Due to their unchecked nature however, macros can provide powerful constructs. In Code Example 2.7, typically the function \(\mathcal{E}\) would be implemented with exceptions, and that is the preferred approach, but this macro does have its advantages: there is no runtime penalty for its graceful exit, and it is able to take any type that defines an output operator as its argument. This kind of code is often used for complicated operations such as compiler state unwinding on detection of syntax errors.
Text macros can provide much more complex and useful macros than these however, and we provide many examples of macro use in this section. What the following examples show is the power that these macros provide, even if it is in an imperfect form. Many of these examples are simplified versions of what could be possible using more advanced techniques such as templates, and template meta-programming (see section 2.3.4.1), for examples of such improvements see the Boost Libraries [AG04].

### 2.3.3.1 Assertions

It is often useful to test a condition and flag an error if the check fails. C++ macros can handle this with ease.

Almost all elements of the macro in Code Example 2.8 are heavily platform dependent, but the intent should be clear. If the condition passed to the `assert` does not evaluate to true, the macro will display the error and cause the program to break. Passed to the error display function is the current file, the line number that failed the assertion, and the text of the expression that caused it.

[C++ Assertion Macro]

C++ has other conditional compilation facilities that allow this macro to effectively vanish from release code. Languages without such facilities often need to provide explicit support for assertions, and it has been a recent addition to Java for this reason. See section 3.4.1.1 for a fuller treatment on the rationale behind assertions.

```c++
#define FAIL(x) { cerr << "error: " << x << endl; 
RELEASE_SYSTEM_RESOURCE(someLocalVariable); 
return false; }

bool f() {
    someLocalVariable = OBTAIN_SYSTEM_RESOURCE();
    if (somethingBadHappend) FAIL("oh no!");
    if (somethingElse) {
        int errorCode = ...;
        FAIL(errorCode);
    }
    ...
    return true;
}

Code Example 2.7: Powerful C++ Macro Construct
```
2.3.3.2 Iteration

An often performed programming task is that of iterating through a list of values. If we wish to iterate through a C++ collection, using the standard iterator classes, the macro in Code Example 2.9 can be used.

The variable \_i is named in such a fashion as to reduce the chances of that name appearing within either of the macro’s arguments or within its body; in general, this problem is called name capture [SPJ02]. Macros that eliminate this problem are called hygienic [SPJ02], see section 3.2.1 for more information.

```
#define FOREACH(element, container) { \
    container::iterator _i = container.begin(); \
    while (_i != container.end()) { \
        container::value_type element = *_i;
        #define ENDFOREACH }

#enddefine
```

```
vector<int> v;
FOREACH(x, v)
    cout << x << endl;
ENDFOREACH
```

Code Example 2.9: C++ Iteration Macro

It would seem that nesting this macro would create problems due to the repeated use of the variable \_i but, surprisingly, it does not. Since we require a uniquely named variable, \texttt{element}, as a parameter, the variable \_i is used for a limited scope, and no name clashes will occur.

2.3.3.3 Generators

Generators are procedures that can produce multiple return values, not as a list, but one at a time, as new values are needed. The programming language Icon uses generators to great effect.

“The greatest difference between Icon and other programming languages is this: in Icon, expressions are generators. Expressions generate sequences of values.”

[Chr96§3.1, pp. 35]

The Icon language is considered interesting enough for there to be a full Java implementation: Jcon [PT99].

The Icon code for defining a procedure that will generate all elements of the Fibonacci sequence is shown in Code Example 2.10. In this example, \texttt{suspend} is similar to
return in other languages, but the state of the procedure `fib` is retained, and the
procedure can be restarted from the point of suspension.

In Code Example 2.10, `every` is analogous to the `FOREACH` macro from the previous
section; by design, this usage would not terminate; it will continue printing the
Fibonacci sequence indefinitely.

```plaintext
procedure fib()
    local x, y
    x = 0
    y = 1
    repeat {
        suspend y
        x = x + y
        suspend x
        y = x + y
    }
end

... 

every x := fib() do write(x)
```

**Code Example 2.10: Icon Fibonacci Generator**

Icon provides other facilities for limiting the number of times we retrieve a value from a
generator. For example if we wished to output all Fibonacci numbers below 100, we
could simple use the expression in Code Example 2.11.

```plaintext
every x := fib() & x < 100 do write(x)
```

**Code Example 2.11: Icon Fibonacci Sequence**

In Icon every procedure has a concept of success or failure, and it is actually possible to
write expressions such as that found in Code Example 2.12. In this example, when `<
fails to produce any more values (i.e. the last Fibonacci number is greater than or equal
to 100), the whole expression terminates.

```plaintext
every write(fib() < 100)
```

**Code Example 2.12: Improved Icon Fibonacci Sequence**

In the following subsections, we demonstrate a text-macro system that can reproduce
the use of Icon’s `suspend` construct. Although a little lengthy, this macro example is,
perhaps, the best example provided here that demonstrates C++ macros ability to _almost_
introduce new syntactic forms, but as previously stated ‘new’ syntax is still constrained
to either macro calls, or identifiers.
**Helper Class Definitions**

Code Example 2.13 introduces a pair of classes: `Iterator` and `Generator`. The `Iterator` class is more similar to a Java iterator than a C++ iterator. The `Generator` class has slightly stronger conditions for use than a traditional iterator in that it requires that `hasNext` and `next` are called alternatively for the duration of the iteration. In fact `hasNext` is responsible for determining whether or not there exists another value in the sequence by performing a calculation and buffering its result.

```c++
template<class T>
class Iterator { public:
    virtual bool hasNext() = 0;
    virtual T next() = 0;
};

template<class T>
class Generator : public Iterator<T> {
private:
    T nextVal;
    int reentry;

protected:
    Generator() : reentry(0) { }
    bool suspend(const T& x, int r) {
        nextVal = x;
        reentry = r;
        return true;
    }
    int position() { return reentry; }

public:
    typedef T valueType;
    virtual bool hasNext() = 0;
    virtual T next() { return nextVal; }
};
```

**Code Example 2.13: C++ Generator Helper Classes**

The internal workings of this class are hidden from end-users, but all overriding classes need access to the `suspend` method in order to take advantage of the cached next element.

**Basic Macro Definitions**

This macro example relies on an odd, generally unknown, and (thankfully) almost unused property of the C++ `switch` statement: case labels are just a special example of arbitrary labels, and as such can appear anywhere within code. As a result of this it is possible to use a `switch` to jump into the middle of a heavily nested structure.
Writing generator style code by overriding the `Generator` class directly would be tedious at best, and would unduly expose the use of `switch` in such an unusual way — a set of macro definitions shields users from this.

The macros in Code Example 2.14 define primitives that can be used to write macros in a similar fashion to the Icon example in Code Example 2.10.

```c
#define GENERATOR switch (position()) { case 0: return suspend(value, __LINE__); case __LINE__: ;
#define SUSPEND(value) return suspend(value, __LINE__);
#define ENDGENERATOR } return false;
```

**Code Example 2.14: Basic C++ Generator Macros**

In Code Example 2.14, the `GENERATOR` and `ENDGENERATOR` macros provide a hidden switch statement that allows for re-entry into the generator code at predefined points. The `SUSPEND` macro controls the definition of these points, and is also responsible for returning the current value from the generator. Notice the use of the special pre-processor value `__LINE__`, this will always be equal to the current line number of the source file. Without this special value each use of `SUSPEND` would be forced to explicitly specify a unique identifier to enable the correct resumption of the generator.

On the calling side, in order to provide a limited form of the Icon `every` statement we simply use a macro very similar to `FOREACH` from section 2.3.3.2, but modified to work on the newly introduced `Iterator`.

**Fibonacci Example**

```c
class fibIterator : public Generator<int> {
    int x;
    int y;

public:
    fibIterator() { }

    bool hasNext() {
        GENERATOR
        x = 0;
        y = 1;
        while (true) {
            SUSPEND(y);
            x = x + y;
            SUSPEND(x);
            y = x + y;
        }
        ENDGENERATOR
    }
};
```

**Code Example 2.15: C++ Fibonacci Generator**
Code Example 2.15 demonstrates how to implement the Fibonacci example using these primitives. In this simple implementation it is still necessary to write a great deal of surrounding setup code. A more complicated set of macro definitions (see the following subsection entitled “Extended Macro Definitions and Example”) would remove this complexity also. What is most important here is that, despite the different syntax, what appears within the GENERATOR / ENDGENERATOR macros is equivalent code to the Icon definition from Code Example 2.10.

Code Example 2.16 demonstrates how the previous code of Code Example 2.16 would appear if all of the macros were expanded to their full form. A couple of things are worth noting here. Firstly, without the ‘versatile’ switch statement this macro would not be possible, and secondly, while it would be possible to write code directly in this fashion and get the benefits of using generators, the code produced is not particularly elegant or maintainable.

```cpp
class fibIterator : public Generator<int> {
  int x;
  int y;

public:
  fibIterator() {
  }

  bool hasNext() {
    switch (position()) {
      case 0:
        x = 0;
        y = 1;
        while (true) {
          return suspend(y, 13);
        case 13:
          x = x + y;
          return suspend(x, 15);
        case 15:
          y = x + y;
          break;
        }
    }
    return false;
  }
};
```

Code Example 2.16: C++ Fibonacci Generator After Macro Expansion

**Extended Macro Definitions and Example**

In Code Example 2.17 the definitions for generator macros are expanded upon, to create less burden on the generator programmer. They hide the definition of both the class extending Generator and the call to hasNext. In Code Example 2.18 these new definitions are used to write in a form even closer to that of the original Icon definition.
Even these improved macros still have limitations: we are working only with generators that have no arguments and further work is required to produce macros that support arguments. In order to reduce programmer burden, it is likely that any such system would require a different macro definition to be defined for each number of arguments.

```c
#define GENERATOR_DECLARATION(type, name) \ 
    Generator<type>* name() { return new name(); }

class name : public Generator<type> { \
    public: \
    name() { } \
    private:
#define GENERATOR \ 
    public: \ 
    bool hasNext() { \ 
        switch (position()) { case 0: \ 
        #define ENDGENERATOR } return false; } }
```

Code Example 2.17: Extended C++ Generator Macros

```c
GENERATOR_DECLARATION(int, fibIterator)
    int x;
    int y;

GENERATOR
    x = 0;
    y = 1;
    while (true) {
        SUSPEND(y);
        x = x + y;
        SUSPEND(x);
        y = x + y;
    }
ENDGENERATOR
```

Code Example 2.18: Improved C++ Fibonacci Generator

### 2.3.3.4 Message Maps

Operating systems generally allow processes to communicate by passing *messages*. In such an operating system, a large portion of programming involves handling these messages. For example, every time an input event (such as a keystroke or mouse input) occurs, a message is sent to the appropriate process, which then must handle this event. This code is typically referred to as a *message map*, i.e. it maps messages to appropriate handling routines. Writing message handling code is a repetitive process.

The Microsoft Visual C++ programming platform for Windows applications makes extensive use of macros to eliminate repetitive tasks [Mic05a].

Code Example 2.19 declares a message handling routine that handles a window paint message with a call to the standard window class member (*OnPaint*), calls a specified member function on detection of a click on a button (identified by *IDC_START*), and
calls a specified member function (SaveAnimationFrame) on detection of any thread finishing messages.

```cpp
BEGIN_MESSAGE_MAP(CMandelDlg, CDialog)
    ////{{AFX_MSG_MAP(CMandelDlg)
    ON_WM_PAINT()
    ON_BN_CLICKED(IDC_START, OnStartDrawing)
    ON_MESSAGE(WM_THREAD_FINISHED, SaveAnimationFrame)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

**Code Example 2.19: C++ Message Map Macro Usage**

This construct is defined in order to remove the repetition inherent in Windows message handling routines. These multiple calls to macros actually define a function that simply returns a datastructure which captures all of the event handlers that the user defines for use in a generalised pre-defined function. All of this is hidden from the user, however, and a message map could be implemented in a variety of other ways.

A message map has rather odd syntax (compared to standard C++ constructs), but is very easy to use. Macros in this case remove repetitive constructs simply and effectively.

**2.3.3.5 DEBUG_NEW**

In order to be able to better track down memory leaks, it is useful during debugging to retain information on all memory allocations and where they take place within code. It would be possible to manually replace normal memory allocation calls with a function that recorded this information, but the C++ macro system provides an elegant solution.

In Code Example 2.20, despite the rather esoteric C++ syntax, the macro simply replaces any instance of the `new` keyword with a macro `DEBUG_NEW` and this in turn expands to a call to an overloaded `new` operator that takes two extra arguments: the current filename and the current line number. The `#ifndef/#endif` pair checks that the compiler is producing a debug executable, and ensures that this macro will only operate in debug mode. When not in debug mode, run-time performance is not hampered by the extra overhead the overloaded `new` operator would require.

```cpp
#ifdef _DEBUG
#define DEBUG_NEW  
#endif

#define new DEBUG_NEW

void* operator new(size_t size, char* filename, int line);  
#endif
```

**Code Example 2.20: C++ Debugging Macro**
For example, consider Code Example 2.21, when performing a debug build. After macro expansion this code will expand to that of Code Example 2.22.

```cpp
int* a = new  int;
SomeClass* x = new SomeClass;
SomeClass* y = new SomeClass(42, "junk");
```

**Code Example 2.21: C++ Debugging Macro Usage**

```cpp
int* a = new("thisfile.cpp", 1) int;
SomeClass* x = new("thisfile.cpp", 2) SomeClass;
SomeClass* y = new("thisfile.cpp", 3) SomeClass(42, "junk");
```

**Code Example 2.22: C++ Debugging Macro After Expansion**

This macro will only operate when the compiler is run in debug mode. Debugging facilities are improved by this macro, and run-time performance is not hampered as the macro is not used when a release build is performed.

### 2.3.3.6 Summary

Text macros are a powerful tool in the right hands, but their use is constrained by the rigid syntax they can provide. As we shall repeatedly see, much use has been made of such imperfect systems, and this is strong evidence that any improvement to meta-programming facilities is of great interest.

It should be stressed that the examples presented here are simplistic compared to some of the techniques used by experienced C++ macro programmers. In particular, real-world `assert` macros are defined in such a fashion as to be usable in places where statements would cause a syntax error.

### 2.3.4 Two-tier Languages

A few languages have attempted to provide more sophisticated meta-programming facilities than text macros by providing a secondary language that enables the user to write extensions that are executed at compile-time.

Such systems may be able to provide extensibility as defined, but do so at the cost of requiring the user to learn this secondary language. This has many disadvantages:

- often this secondary language is either harder to program in, or less expressive, than the main language;
- programmers must clearly understand two languages instead of one, and must be able to reconcile the differences between them; and

• 35 •
in the majority of cases there is no apparent correspondence between the extension code itself, and the code produced by executing the extension. “Asymmetry between the static and dynamic language is bad… It might be desirable to have near-perfect symmetry between the static and dynamic languages, even to the extent of allowing side-effects at compile-time.” [Vel99]

Examples of such two-tier systems are Pliant [TS?] and C++. In the following subsection we briefly examine the approach taken by C++.

2.3.4.1 C++ Template Meta-programming

One of the most startling accidents [Rob01] in programming language design is C++ template meta-programming. Programmers realised that they could utilise the type system of C++ templates to perform calculations — a use never intended by the language designers.

“C++ has an elaborate meta-programming facility known as templates. The basic idea is that static, or compile-time, computation takes place entirely in the type system of C++.” [SPJ02§10.1, pp. 12]

This was first discovered by Unruh [Unr94] in 1994, when he produced a program that would produce compilation errors that contained a prime number calculation.

As a simple example of this template meta-programming functionality, consider the set of template definitions to calculate integer powers at compile-time in Code Example 2.23. Here, the second template definition is a partial specialisation, and it provides the base case for a recursive calculation of a power (functional programmers may notice that this is a common functional programming idiom with horrendous syntax). The C++ template type system is being tricked into performing calculations for us.

```cpp
template<int X, int Y> struct pow {
    static const int result = X * pow<X, Y-1>::result;
};

template<int X> struct pow<X, 0> {
    static const int result = 1;
};

const int z = pow<5, 3>::result;
```

Code Example 2.23: C++ Template Meta-programming
This type system is considered to be a functional language in its own right — so C++ effectively has one imperative language for the majority of work, and an embedded functional language (with an awful syntax) that can be used for meta-programming.

“The type system is rich enough that one can construct and manipulate arbitrary data structures (lists, trees, etc) in the type system, and use the computations to control what object-level code is generated.” [SPJ02§10.1, pp. 12]

This kind of code often bears little resemblance to the code it is going to produce, and the mechanism at work here is generally little-understood by the end user. Nonetheless there has been a recent flurry of activity in the C++ meta-programming community, with many projects relying on this facility. The most extensive set of libraries commonly using meta-programming is Boost [AG04]. The facilities it provides include: mathematical calculation, string processing, function-objects, and memory management facilities. One of the most extraordinary examples of template meta-programming is to produce a Lisp subset which is interpreted by the type system [CE98].

“The fact that C++ templates are so widely used [for meta-programming] is very strong evidence of the need for such a thing: the barriers to their use are considerable.” [SPJ02§10.1, pp. 12]

If meta-programming in C++ is so baroque, and yet so often utilised, then it should be clear that its power is considerable. Any system that provides the power of meta-programming but with improved usability should be considered a great improvement [Wil04].

**Mathematical Calculation**

Matrix operations (as well as other linear algebra calculations) are a classic example of C++ template meta-programming. Specialised code that can be as efficient as hand generated code can be produced for matrix operations, and this process is transparent to the user. This enables users to forget about efficiency, and to program with matrices in the usual way, e.g. \( A = B + C \times D \).

This has traditionally been the domain of Fortran, and specifically its BLAS library [Bla01]. A concerted effort was made by many developers to try and match Fortran’s efficiency for these kinds of operations [AG04, RJHC’96].
In explanation of the rationale for the creation of Boost [AG04], the following quote appears:

“It would be nice if every kind of numeric software could be written in C++ without loss of efficiency, but unless something can be found that achieves this without compromising the C++ type system it may be preferable to rely on Fortran, assembler or architecture-specific extensions.” [Str94§6.5.2]

Libraries such as Blitz++ [Vel99, Vel01] and POOMA [RJHC^96] relied heavily on the use of template meta-programming, and claim to match the performance of Fortran. In these libraries, efficiency issues are the concern of the meta-programmer who is free to apply specialisations depending on the size and shape of the matrices. Small statically sized matrices can have their operations fully unrolled and can utilise lazy evaluation, while the largest matrices can utilise other highly optimised techniques.

Of major concern to the optimisation of C++ code is the elimination of temporary values and virtual function calls. To solve the problems of ‘temporaries’, the Boost library defers calculation until enough information is known to produce a suitable optimisation:

“[using] lazy evaluation as known from modern functional programming languages. The principle of this approach is to evaluate a complex expression element wise and to assign it directly to the target.” [WK00]

To remove virtual function calls, the Boost library found a solution:

“… called expression templates. Expression templates contain lazy evaluation and replace dynamic polymorphism with static, i.e. compile time polymorphism.” [WK00]

These systems show that C++ meta-programming is expressive enough to provide the library designer with many powerful optimisation strategies and, as a result, abstraction has been maintained without significant performance penalties. An extensible language could provide these facilities and, when properly designed, would make their implementation more natural.
2.3.5 Integrated Language Features

As discussed in the previous section, it is generally considered undesirable to provide meta-programming facilities in a form that greatly differs from the usual programming idiom [Vel99, SPJ02]. Instead, meta-programming should be provided in the language itself. Template Haskell is one language that uses this approach:

“… the static computation language is the *same* as the dynamic language, so no new programming idiom is required.” [SPJ02§10.1, pp. 12]

Meta-programming in such a language has available to it the full power of the base language, and as such can produce the most powerful constructs. Examples of languages that provide integrated meta-programming facilities are Lisp/Scheme, Template Haskell, Java Syntax Extender, OpenJava, and Maya. We discuss these languages, the differences in expressive power, and their claims to extensibility in Chapter 4.

Before we can adequately cover these languages, we will first look at the need for extensibility, and mechanisms for comparing extensible languages.


2.4 Is Extensibility Necessary?

In [Bar91], the evolution of programming languages is described as having undergone three major advances in abstraction:

- expression abstraction (introduced, with constraints, in languages such as Fortran [For77]);
- control abstraction (introduced in Algol60 [Nau60]); and
- data abstraction, which (due to the age of the book) is “still occurring” (introduced with language such as Simula [BDMN73], Smalltalk [GD83] and ML [Mil84]).

Expression abstraction allowed the programmer to ignore such issues as machine registers. Control abstraction freed the programmer from such house-keeping issues as labelling and gotos. Data abstraction allows the details of the actual physical representation of data to be separated from the operations defined upon the data; from the introduction of records, through to classes, and then parameterised classes.

Each stage saw a reduction in the number of tedious and error-prone tasks that would normally be the responsibility of the programmer. Existing programming languages still require many repetitive and error-prone tasks to be performed. The programmer often has no choice as the language itself is not expressive enough for their needs. In many situations, programmers resort to using different languages for each often-performed task. Repetitive tasks lead to the creation of new languages to alleviate this repetition — just as there were once an abundance of ‘new’ text processing languages (eg. SNOBOL [FGP64], Perl [WCS96], etc.), recent trends have spawned a host of scripting languages (eg. Python [RD04], Ruby [Mat98], etc.), and Web/HTML languages (e.g. PHP [Ach05], etc.).

It could be argued then, that the next stage in abstraction is that of syntax abstraction, so that these repetitive tasks could be handled where they appear.

“The next big improvement in programmer productivity is going to come from making programming languages more extensible.” [Her04]

There are reasons for continued development of programming languages other than repetition. Looking at any reasonably extensive list of programming languages would

• 40 •
tell us that there are at least a few thousand world-wide, and it is likely that many never make it into the public domain. The reason for the creation and characteristics of a new language are varied. For example, a new language might:

- offer new previously unthought of constructs, or even entire evaluation strategies;
- be built upon old languages, or merely based upon them;
- attempt to provide the best features of other languages (eg. multi-paradigm languages such as C++);
- be tailored to a particular machine architecture (eg. as C originally was), or as part of the infrastructure of another program (eg. QuakeC [Mon96] and UnrealScript [Swe98]); and/or
- be created to test purely to see if an idea will work, or with a single specific goal, i.e. ‘research languages’ (eg. the language BrainF*** [Wik95] was designed to be Turing complete but with minimal compiler executable size).

But even all these reasons are just a tiny proportion of all reasons behind language design. We will look at a few more of these motivations in some of the following subsections. Many of these reasons can be supported by providing an extensible language, if such a language were available much language development could actually become libraries of such a language.

“In order to make that happen, we need to make compilers, debuggers, and other tools extensible frameworks.” [Her04]

One strong argument in favour of extensible languages is the abundance of situations in which they can be useful, and in the number of these where they could easily replace traditional methodologies:

“From now on, a main goal in designing a language should be to plan for growth.” [Ste99, pp. 6–7]

The creators of Template Haskell consider that their system could be used for:

- conditional compilation (i.e. replacing a pre-processor);
- program reification, i.e. program introspection (eg. automatic generation of serialisation code);
- algorithmic program creation;
• abstractions impossible in the base language; and
• optimisations (such as algebraic laws and function in-lining).

In particular, allowing users to define optimisations takes the emphasis off the compiler (and the compiler writer), and provides power to those that best understand the possibility for optimisation for a given programming domain. Many optimisations are just not cost-effective to include in a compiler:

“…many interesting optimizations have too narrow an audience to justify their cost… An alternative is to allow programmers to define their own compile-time optimizations.” [Rob01, pp. 1]

Extensible languages could also alleviate many of the reasons for language revision, reduce the frequency/complexity of standardisation, and allow for a faster pace of language development for end users.

Language revision often occurs to correct design deficiencies or in order to provide new functionality in such a form that is impossible to provide through any other means. Generally such new functionality is provided with syntax unmatchable by library/module mechanisms.

Such language revision can lead to revision of associated standards, which can be a slow process. Language revisions are often embraced by programmers before standardisation is complete, but each compiler may provide a different set of non-standard extensions. A system that allows language extension without compiler modification could allow the adoption of new extensions by the programming community without any compiler, language, or standard revision.

As we shall see, many of the arguments against extensibility are the same as those arguments that were (and in some cases continue to be) levelled against other, now common-place, language features. Also, extensibility can be viewed as the next step in a natural trend of language development.

2.4.1 What is Necessary?

The most likely question to be asked is whether or not we need extensibility.

We could also ask ourselves whether or not we need generics / parametric polymorphism and, strictly speaking, the answer is no, they are not necessary — you can write essentially equivalent code without language support for generics, but in a far
less desirable form. Indeed, for a programmer that uses a language that features generics as a matter of course, it is quite painful to switch to one that does not.

Being slightly facetious for a moment, we could also ask if we need classes, procedures, conditional loops, etc. We could all still be programming in assembly, without a pre-processor, or better still, machine-code was good enough for all our purposes.

“We write programs in high-level languages because they make our programs shorter and more concise, easier to maintain, and easier to think about.”

[SPJ02§3, pp. 2]

If the standard facilities of current high-level languages have these benefits, then a well constructed extensible language should only serve to increase them.

“Many low level details (such as data layout and memory allocation) are abstracted over by the compiler, and the programmer no longer concerns himself with these details.” [SPJ02§3, pp. 2–3]

This is but part of a continuing trend for increased abstraction from the conception of computer programming.

2.4.2 Language Modifications

It should be clear from the previous section that programming languages have continued to evolve over time, newer languages provide facilities that were not available in their predecessors. Also when designing a language it is impractical to attempt to cater for every use from the outset, it is simply impossible to anticipate every desire of every future user. However, language designers do attempt to provide enough functionality that everything should still be at least possible, even if not simple. A better solution is to make the language extensible:

“There is a limit to the number of features any compiler writer can put into any one compiler. The solution is to construct the compiler in a manner in which ordinary users can teach it new tricks.” [SPJ02§3, pp. 3]

Language revision has always been part of the process of language design and is a phenomenon more noticeable in mainstream languages, eg. Fortran [For77], C/C++, Ada, Java, HTML [RLHJ99], Perl [WCS96], etc. For those languages that go
mainstream, what tends to happen is as time goes by and more unsupported common programming styles emerge, the language is revised and re-released.

A prime example is Java: the Java programming language was designed with simplicity in mind, but as it became clear that programmers were tiring of explicit casts in their code (both writing them, and maintaining them), a concerted effort was made to provide generics. From the original call for revision [Bra99] in May 1999, it took over five years for the corresponding final release of Java1.5.

Previous Java revisions added the concept of assertions, and bundled with generics in Java1.5 were the addition of automatic boxing/unboxing, type-safe enumerations, variable length argument lists, a revised import mechanism, and class metadata [Bra02]. If Java had been created extensible from the outset, then these features could have been provided as libraries.

2.4.2.1 Language Standards Revision / Development

Language revisions take time, but they are nothing compared to the slow process of standardisation and re-standardisation.

Standards exist to try and constrain language development to a degree, so that (in theory) compiler divergence does not occur. If left unchecked, so many people will want to modify any given language, that a concerted effort must be made to provide a common ground for the base user.

If the language itself allows for these modifications to occur, without the need for a new compiler, then the need for a periodic creation of a revised language standard should all but disappear.

“An extensible language is one which puts this power in everyone’s hands, instead of reserving it for a standards committee.” [Wil04]

2.4.2.2 Research Languages

Research languages are typically small single purpose languages created for experimentation or teaching purposes. Due to their nature they tend to have custom built compilers and little or no tool support. Implementing these languages in an extensible language gives the benefits of pre-existing tools, and hopefully would simplify the process of creating them to begin with.
2.4.2.3 Embedded Languages

As previously covered in section 2.3.1.2, SQL is an example of a language that is frequently embedded in other languages; in this section we describe this as preferable to adding SQL support via strings and the normal library mechanism.

The problem with providing an embedded language is that support is only provided at the language definition level. If we wish to embed HTML, XML, or something else unforeseen then the language must be revised, and again this could require the intervention of a standards committee.

An extensible language could provide embeddings of many other popular languages, and would have the ability to provide new embeddings as the needs of users change.

Design Patterns

Design patterns have been the focus of much research in recent times (e.g. [Haa02], [BFYV96]), much of the hype about them is very similar to that of object-orientation from a few years previously. However,

“Design patterns can be viewed as workarounds for specialized features missing from general-purpose languages. For instance, the visitor pattern implements multiple-dispatch in a single-dispatch language.” [Bak01§1, pp. 1]

Indeed, it should be obvious to anyone who has used design patterns extensively that they end up writing very similar looking code, over and over. In order to remove this repetition, some languages have started supporting some common design patterns explicitly. A recent example is C# [HW01]: it provides explicit support for both the state/observer pattern and delegation [Bak01].

It is clear that this approach has benefits for the programmer. However, this kind of technique is reliant upon the programmer language designer recognising a common need, and then addressing it.

“… unless we are willing to wait for a new language each time a design pattern is identified, such an approach is unsatisfactory. Instead, a language should admit programmer-defined syntax extensions.” [Bak01§1, pp. 1]

Allowing extensibility to take care of design patterns has added benefits: if a programmer does not use a particular design pattern, they are not burdened with having to understand it, and are unlikely to use it in an inappropriate manner. Often, the same
cannot be said for concepts that are directly supported by a language. Indeed, this is one of the great criticisms levelled against C++ in particular (see section 2.4.4 for a discussion of these kinds of issues).

2.4.2.4 Java1.5

Java1.5 was a result of work done on Pizza and Generic Java (GJ) that lead to a call for revision [Bra99]. One of the touted design benefits of GJ was that it used erasure to convert programs with parametric polymorphism, to traditional Java programs, and as such no change was required to the Java Virtual Machine (JVM).

“GJ is backward and forward compatible… with [the] JVM. GJ compiles into JVM code. No change to the JVM is required. The code is verifiable and can be executed on any JDK compliant browser.” [BOSW98a, pp. 2]

Erasure

Implementations that use erasure systematically translate all language extensions into the base language itself. Using erasure Java1.5 programs are (implicitly) converted into Java1.4 programs which are then compiled as normal:

“GJ is translated by erasure: no information about type parameters is maintained at run-time. This means GJ code is pretty much identical to Java code for the same purpose, and equally efficient.” [BOSW98a, pp. 2]

The Java1.4 language was not extended in a way that gave the power of extensibility to a programmer, but Java1.5 was implemented using a technique fundamental to all extensible programming languages.

This property of erasure is what extensible languages do. Even simpler systems like C++ macros are erased from the code during compilation. In fact, generally C++ can be translated to equivalent C code, and this is how the language was originally, and often still is, implemented. The language of choice for implementing new programming languages is typically C, which is often referred to as portable assembly language. If C provided an extensible framework, then these languages could actually be implemented by erasure in C itself.

---

3 Although a minor addition to class file attributes occurred in order to provide more support to bridge methods, this is likely to be for efficiency or merely documentation purposes.

• 46 •
2.4.3 How Much Power is Too Much Power?

One criticism of extensibility is that it is too powerful, programmers would misuse it, and that too many extensions would start to make the language unusable.

This is essentially the same argument levelled against operator overloading. An often cited example [Wik01] of misuse is in the overloading of the C++ arithmetic shift operators (<<, >>) as output primitives — in fact these operators would be recognised by most programmers for their use in output, not in arithmetic. This clearly shows that the one symbol has been given vastly different meanings, even in the standard libraries. This situation could be much worse if programmers can redefine any syntactic structure.

Operator overloading however, is used to produce concise powerful libraries in C++ [AG04, Gur04], Haskell uses operator overloading extensively with great results (eg. for recursive-descent parsing [HM98]), and even the Java language uses overloading on the + operator for strings [GJSB00].

Many reasons can be given as to why string creation is treated specially in Java, but primarily the reason is that programmers often wish to build up strings from multiple sources of different types and concatenate them together — it is an optimisation of a commonly occurring type of code. In reality this is not an uncommon situation, the language designers just have more power at their disposal than the programmer.

Almost any language feature can be misused; many programming style-guides suggest never using operator overloading, break statements, return statements anywhere but the end of a function, etc. Most of those making such suggestions don’t argue for the removal of these facilities altogether, for the simple reason that at times breaking these rules produces the most understandable, or efficient code.

Simply not providing a feature is no solution to the problem of misuse, programmers would shy away from languages or libraries misusing extensibility just as they would from those misusing any feature. Extensibility can provide power in a form that no other language constructs can, and that alone makes it an interesting and important addition to a language.
2.4.4 Multi-paradigm languages

C++ and Ada are two well known multi-paradigm languages. Complaints have been made against both as being too complex [Pla93, Joy96]. Indeed, it is often said that no one person can fully understand all of C++ [Mil00]. Proponents of these systems argue that you only need to know the parts of the language that you need, the other features are there for you to discover if your needs change.

C++ can be used as, at minimum:

- a procedural language in the fashion of C;
- an object-oriented language;
- a meta-programming language;
- a data abstraction language;
- a generic programming language; and
- even a limited form of functional programming language.

It was consciously designed to support multiple paradigms.

“The idea that there is one right way to solve essentially every problem for essentially every user is fundamentally wrong.” [Kal01]

Furthermore, C++ attempted to provide support for its new paradigms while paying close attention to efficiency.

“… a general-purpose programming language must support multiple paradigms and … each paradigm must be supported well and with close-to-optimal runtime and space efficiencies.” [Kal01]

It is also well understood that C++ code can be translated into C code, this is part of the reason that efficiency is so well supported. Again, this is the property of erasure. If one were to create an extensible version of C, it should be possible to create all of the features of C++ within the language.

Arguments continue as to whether or not multi-paradigm languages or small-is-beautiful languages are the most effective way to get a job done [Joy96]. A small-is-beautiful extensible language, however, should be able to nicely side-step this argument completely. Attention can be paid to creating an elegant base language with powerful extensible constructs, and all multi-paradigm support can be provided by the extensions. Such a language would provide the best features of both languages.
2.4.5 Previous Interest

There has been a fair degree of recent interest in the concept of extensibility. Clearly, other researchers are of the opinion that it is at least an interesting language concept.

“Language extensibility has been around for years, but is still largely an academic curiosity. Three things stand in the way of its adoption: programmers’ ignorance, the absence of support in mainstream languages, and the cognitive gap between what programmers write, and what they have to debug.” [Wil04]

Some aspects of this argument mirror those that were arguing for mainstream adoption of parametric polymorphism in less recent times.

Part of the reason for programmers being ignorant of extensibility is not just its lack of common availability, for those programmers that do bump up against extensible languages, there can be great barriers to use. It can be argued that extensibility has to be simplified before the common user will choose to use it.

It is worth reiterating that even without the full power of syntax creation, C++ templates do provide meta-programming, and as a result their has been a lot of excitement about the kind of things that it is possible to use them to do:

“The fact that C++ templates are so widely used [for meta-programming] is very strong evidence of the need for such a thing: the barriers to their use are considerable.” [SPJ02§10.1, pp. 12]

If meta-programming is becoming so popular, being able to provide syntax creation should only add to its appeal.
Assessment of Extensibility

CHAPTER 3

1. Introduction
2. Defining extensibility
3. Assessing extensibility
4. Reviewing extensibility
5. Designing the language
6. Reviewing parsing
7. Implementing a parser
8. Implementing the language
9. Evaluation
10. Conclusion
In order to review multiple languages in an objective way, we must first develop a suitable mechanism to achieve this goal.

In this chapter we first review some desirable properties of extensible languages. These properties are summarised and formalised into a set of criteria for rating such extensible languages (section 3.3).

In addition to this qualitative assessment, we introduce a suite of benchmark test cases that cover the full range of these desirable properties of extensible languages (section 3.4).

In chapter 4, we rate other languages against both this set of criteria and their suitability to this benchmark suite.
3.2 Desirable Language Properties

As previously quoted in the description of Template Haskell, the purpose of extensibility was summarised as follows:

“The purpose of the [extensible programming language] is to allow programmers to compute some parts of their program rather than write them, and to do so seamlessly and conveniently.” [SPJ02§1, pp. 1]

There is a double emphasis here, both on the ability to perform compile-time meta-programming, and also to do so easily. An extensible language should be convenient enough to use so as to allow easy creation of small syntax additions of limited scope, but expressive enough to cater for the embedding of another domain-specific language within itself. It should support overloading of syntax in some form in order to allow both the overriding of the behaviour of built-in syntax and the creation of optimisations.

“Grammar extension macros allow a programmer to make incremental changes to a grammar in order to extend the syntax of the base language.” [BP01§8.4, pp. 12]

A well designed extensible language should also be end-user friendly, great care should be taken to provide constructs that ease the burden on the programmer. An extensible language should not create barriers to use, to this end it should balance simplicity and power — ideally the language would not sacrifice any power whilst maintaining an intuitive programming style.

“Making it easy for users to manipulate their own programs, and easy to interlace their manipulations with the compiler’s manipulations, creates a powerful new tool.” [SPJ02§3, pp. 3]

In particular, macro programmers should not be unduly exposed to the underlying implementation of the language. Many previous systems have limitations to macro definitions imposed by their parser that require end users to have a large degree of background knowledge, and as a result:

“… it is difficult for programmers to understand and solve static grammar ambiguities …” [BP01§8.4, pp. 12]
Weise and Crew [WC93] insist that macros arguments be abstract syntax trees not tokens, and specify three properties of extensible systems:

- syntactic abstraction: the ability to define new syntactic forms;
- non-interference: macro arguments should not suffer from the basic problems of C/C++ text macros (see section 2.3.3); and
- syntactic safety: all macros expansions should result in syntactically valid structures.

Most modern macro systems provide syntactic safety through use of a set of abstract syntax structures. Non-interference has been improved with the introduction of hygienic macro systems (see the following subsection). Most macro systems compromise on syntactic abstraction – despite this being perhaps the most desirable property of extensible languages.

An extensible system should allow macros to be bundled with libraries/classes in much the same way as procedures/methods are. The import mechanism should allow easy use of such macros with convenient syntax and should shield the programmer from conflicts where possible.

### 3.2.1 Hygiene and Referential Transparency

Both hygiene and referential transparency are concerned with the preservation of meaning of variables after macro expansion. The difference between them can be hard to grasp, and has often led in other systems to both terms being combined into the single term hygiene.

“The basic idea is that each named value reference in a macro expansion means the same thing as it meant at the place in the original source code from which it was copied into the macro expansion. This is true whether that place was in the macro definition or in the macro call.” [Sha96a§10]

Hygiene ensures that no variable introduced by a macro expansion will collide with variables from the surrounding context. Generally, hygienic macro systems ensure that any variables declared within a macro expansion are given unique names.

“… the property that variable references copied from a macro call mean the same thing in the expansion is called hygiene.” [Sha96a§10]
Referential transparency ensures that variable references within a macro definition that refer to the surrounding context still refer to the same variables if they are present in the resulting expansion.

“… the property that variable references copied from a macro definition mean the same thing in the expansion is called referential transparency.” [Sha96a]

In essence, hygiene and referential transparency concern themselves with removing inadvertent variable capture; a frequent problem in early macro systems.

“A hygienic macro system guarantees that variables declared in a macro body cannot capture references in a macro argument, while a referentially transparent macro system guarantees that variables local to a macro’s call site cannot capture references in the macro’s body.” [Bak01§2, pp. 6]
3.3 Criteria for Rating Extensible Languages

The following criteria summarise the desirable properties identified in the preceding sections. There are three main criteria on which extensible languages can be rated: power, usability, and error handling. These criteria can be further developed into sub-criteria and this is undertaken in the next section.

3.3.1 Power

These criteria concern themselves with the kinds of extensions that can be supported by a given system. Both the power of the meta-programming constructs and the ability to provide syntax extensions are assessed. In fact, a language that fails to adequately address criterion 1.1 does not even meet our definition of extensible.

A language that meets all of these criteria is providing the programmer with power close to that of a compiler writer — all of the extension types described in section 2.4 are possible.

Table 3.1: Criteria for rating an Extensible Language’s Power

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.1 Syntax Creation</strong></td>
<td>The possibility of defining any new arbitrary syntactic form the user wishes.</td>
</tr>
<tr>
<td><strong>1.2 Syntax Interrogation</strong></td>
<td>The ability to look at the values of literals, the subcomponents of expressions, etc.</td>
</tr>
<tr>
<td><strong>1.3 Syntax Overloading</strong></td>
<td>The capacity to override the behaviour of built-in syntax, or previously defined extensions in certain circumstances.</td>
</tr>
<tr>
<td><strong>1.4 Static Type Interrogation</strong></td>
<td>Interrogation of compile-time static types. This allows static type checking by macros for robustness, and allows the creation of specialisations for specific types (specialisations generally require syntax overloading also).</td>
</tr>
<tr>
<td><strong>1.5 Expressiveness</strong></td>
<td>In order of increased power, an extensible language’s</td>
</tr>
</tbody>
</table>
expressiveness can allow:

- the creation of small syntax additions of limited use;
- the embedding of another domain-specific language within itself; and
- the original system could be completely discarded, and a whole new language could be built instead.

### 3.3.2 Usability

These criteria assess the convenience, safety, and complexity of extensions that can be defined within a language. Extensions, where possible, should be simple to conceptualise, simple to express, and not likely to produce unexpected errors.

Of particular importance in assessing simplicity is the breadth of knowledge that is required of the programmer. It is desirable that programmers don’t need to know parser theory and should be shielded from conflicts that may arise from automatic parser generation.

**Table 3.2: Criteria for rating an Extensible Language’s Usability**

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2.1 Simplicity</strong></td>
<td>The facilities for extensibility should be as simple as possible, in particular, the user should not have to understand parser theory.</td>
</tr>
<tr>
<td><strong>2.2 Brevity</strong></td>
<td>Where possible, macros should appear as similar to the code they expand to. Of particular use here are quasi-quote and unquote (see section 4.2.2).</td>
</tr>
<tr>
<td><strong>2.3 Robustness</strong></td>
<td>Robustness. The programmer should be shielded from writing macros which can unexpectedly conflict with the surrounding context. Of particular use here are referential transparency and hygiene.</td>
</tr>
</tbody>
</table>
3.3.3 Error Handling

These criteria concern themselves with the correctness of macro expansions, and the system’s ability to report errors to the programmer. The language system should allow the macro programmer the ability to explicitly detect errors but should be capable of reporting any other errors in a useful fashion.

Table 3.3: Criteria for rating an Extensible Language’s Error Handling

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Syntactic Correctness</td>
<td>The system should guarantee that all macros produce syntactically correct forms.</td>
</tr>
<tr>
<td>3.2 Error Detection</td>
<td>The system should allow macro programmers to detect errors and report this back to the compiler.</td>
</tr>
<tr>
<td>3.3 Error Reporting</td>
<td>When errors occur during expansion, the user should be given an idea of what the error was in relation to their own code. Of particular importance here are errors that are caught by the compiler, and not the macro itself. Users should not, where possible, have to debug generated code.</td>
</tr>
</tbody>
</table>

3.3.4 Previous Research

Previous research (eg. the languages reviewed in chapter 4) has tended to focus most on criterion 1.2, 2.2, and 2.3, i.e. the ability to look at compile-time program structure, and create robust macros as easily as possible.

A few systems have addressed criterion 1.3, and 1.4 to some success, these systems are concerned with adding power to more traditional systems.

Criterion 2.1 (simplicity) tends to be either not much of an issue for many systems as they do not provide arbitrary syntax, or is largely ignored by those systems that provide limited parsers.

With regard to criterion 1.5 (expressiveness), few systems can embed domain-specific languages, and fewer still can effectively discard the base language.
Some work has concentrated on error handling, but so far this has mostly been limited to criterion 3.1 (syntactic correctness), with some systems addressing criteria 3.2 (error detection).

Surprisingly, the area of least research is criterion 1.1 (syntax creation), most systems that support compile-time meta-programming do so effectively, but often make no attempt to provide arbitrary syntax creation. As we shall see in chapter 4, there are only a few languages that address this issue.

### 3.3.5 The Library System

As discussed in section 3.2, a macro system requires a well designed import mechanism. Whilst none of the criteria in this section directly assess the quality of this mechanism, a poorly designed import mechanism would not meet some of the identified criteria. For example, a library system that overly restricts sharing of macros between modules would be assessed less favourably in terms of expressiveness, and one that provides too few restrictions (such as the C++ macro system) would fail to meet the robustness criteria.
3.4 Benchmark Test Cases

In this section a suite of test cases is suggested as benchmarks for the evaluation of the capabilities of any extensible languages. The cases range from simple statement macros to whole-scale embedded languages and semantic changes. The ability to implement these constructs can be viewed as a ‘proof-by-implementation’ for an extensible language.

For the simple constructs, explanation of how these facilities are provided in C++, Java, and Maya are provided. These three languages have been chosen for their syntactic and semantic similarity, both with each other, and with the target language of this research, but their techniques for supplying these constructs are fundamentally different: C++ can generally support these constructs via text macros, revision of the Java language is typically required, and Maya provides an extensible approach.

For the complex constructs examples of previous implementation attempts are provided, where possible. Unfortunately, these constructs are not, or cannot be, implemented in most languages, so this comparison cannot be as comprehensive as would be preferred.

The benchmark cases are summarised in Table 3.1 with further subsections providing full details.

Table 3.4: Benchmark Test Cases Summary

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 assert</td>
<td>Run-time condition checks.</td>
</tr>
<tr>
<td>2 foreach</td>
<td>Improved iteration on containers.</td>
</tr>
<tr>
<td>3 printf</td>
<td>Type-safe formatted output.</td>
</tr>
<tr>
<td>4 SQL</td>
<td>Embedding of the SQL SELECT statement.</td>
</tr>
<tr>
<td>5 Generators</td>
<td>Integration of generator functions.</td>
</tr>
<tr>
<td>6 Haskell</td>
<td>Embedding of functional programming declarations and mixing of lazy and imperative evaluation.</td>
</tr>
</tbody>
</table>
3.4.1 Simple constructs

These simple constructs are chosen to demonstrate some necessary features of any extensible language.

Each of these simple constructs are new statements. Much power can be added to a language that would only allow the creation of new statements (for a further set of such constructs, and their use, see [Gra93]); as we will see in section 3.4.2 there are many more possibilities for language extension.

3.4.1.1 Assertions

Assert statements are common in many languages in some form. An assertion checks some condition and will cause the program to stop if this condition is not true.

The code fragment in Code Example 3.1 would stop the program if `someVariable` was not a reference to a defined object.

```c
assert (someVariable != null);
```

Code Example 3.1: Assertions

In the debugger-less situation, it is desirable that a failed assertion will produce some sort of meaningful error message — ideally including the source file and line number that caused the exception.

Assertions can be used as an informal design-by-contract facility [Sun02], allowing the programmer to check pre-conditions, post-conditions, and class invariants. The reasons for using assertions mirror those arguments for supplying program proofs, but in a more pragmatic fashion.

**Necessary Language Features**

This construct merely requires us to be able to produce code that will cause an exception to be raised at run-time. This is heavily dependent on the traditional language providing such facilities — but this is the case for all macros, it is not possible to provide facilities that are inherently impossible to provide in the base language.

**C/C++ ASSERT**

C/C++ provides this facility via pre-processor text macros [Str00§24.3.7.2], generally in a form that evaporates when not debugging so that release code suffers no assertion overhead. Review section 2.3.3.1 for further explanation.
Java assert
Direct language support was provided for assertions in Java 1.4 onwards [Sun02]. Interestingly, this is a situation where Java is *almost* expressive enough to handle assertions via the normal library mechanism. It would appear that the choice to add direct language support is for both uniformity and efficiency. Programmers typically provided their own ad-hoc implementation of assertions, either via a library or with direct embedding in code. Often the direct embedding approach was favoured because it allowed for greater efficiency as no library based solution could be implemented without run-time cost when assertions are “switched off”. As a result,

“[library-based implementations of] assertions have never become a part of the culture among engineers using the Java programming language. Adding assertion support to the platform stands a good chance of rectifying this situation.” [Sun02]

It could easily be argued that a vital reason that library solutions would not be uniformly accepted is if their efficiency performance was questionable.

“The library approach was considered. It was, however, deemed essential that the runtime cost of assertions be negligible if they are disabled. In order to achieve this with a library, the programmer is forced to hard-code each assertion as an if statement.” [Sun02]

The Java language needed to be changed to implement assertions in an efficient manner, this is in stark contrast to the ease of providing assertions via the C/C++ pre-processor.

Another, albeit minor, problem with this implementation is the introduction of `assert` as a new keyword, which can break old code.

As shown in Code Example 3.2, the Java 1.5 implementation of assertions provides a convenient syntax that a library designer has no possibility of imitating.

```java
assert x > y;
assert (x >= 0) && (x <= 100) : x   // pass x back as helper info
```

**Code Example 3.2: Java 1.5 Assertions**

Maya assert
As shown in Code Example 3.3, Maya’s `assert` statement mimics a function call in syntax.
Maya provides an extra level of checking to either the C++ or Java assertion mechanisms by ensuring that no assertion expression may contain side-effects [Bak01] — although it is unclear at what level of detail these checks are made. It is unlikely that side-effects can be discovered within functions that make up the expression. (It is more probable that it is only a shallow side-effects check, masquerading as a deep check, and this would likely cause more problems than it would detect.)

### 3.4.1.2 Iteration

A `foreach` statement provides similar functionality to many uses of a traditional `for` statement, namely iterating over a list of values, but does so without making visible the means of iteration. This is considered to be a worthwhile abstraction in its own right, but mostly it simply frees the user from writing repetitive iteration code everywhere. Typical Java iteration code and the `foreach` form suggested here for the example of printing out a list of strings, is shown in Code Example 3.4.

```java
for (Iterator i = list.iterator(); i.hasNext(); ) {
    String s = (String) i.next();
    System.out.println(s);
}
```

(a) Typical Java Iteration

```java
foreach (String s) in list { System.out.println(s); }
```

(b) `foreach` Iteration

Code Example 3.4: Iteration

The first argument specifies the types of objects expected to be found within the container (the second argument), as well a name binding for each of these that can be used freely within the following statement (the third argument), which may or may not be a block.

This form is much more compact, more readable, and less error-prone than the hand-written code. As an abstraction, it may be that different containers may produce different expansions for efficiency reasons. For example, an expansion for the Java class `Vector` could expose its internal array implementation.
**Necessary Language Features**

The primary feature illustrated here is that of true syntax creation, this `foreach` does not attempt to mimic a function call, or the traditional for statement, it provides an almost directly readable statement form. What would be traditionally reserved words, `foreach` and `in`, do not just appear at the beginning of the statement, but as the first and third arguments.

Other features required to provide this construct are that a use of `foreach` requires a variable declaration that is decoupled from the block which requires its use. A well designed version of this construct also requires that we be able to check the type of the container argument to ensure that it is indeed `Iterable`, this requires us to inspect the properties of previously defined variables.

**C++ for_each**

The C++ Standard Template Library (STL) [SL94] contains the function `for_each` [Str00, pp. 523] to provide similar support for its rather different concept of iterators — this is demonstrated in Code Example 3.5.

```cpp
for_each(list.begin(), list.end(), some_function);
```

**Code Example 3.5: C++ STL Iteration**

The major flaw in the C++ approach is that it does not operate on an arbitrary statement. C++ is not expressive enough to allow this, but instead requires a function pointer — this in-turn removes the need to specify an iteration variable. This requirement deters programmers from using `for_each` for simple loops, as the extra effort involved in writing a helper function outweighs the benefits gained by this abstraction.

A macro definition is possible (as seen in section 2.3.3.1) but would require the user to explicitly provide the name of the hidden variable to guarantee that no name clashes occur with external variables.

**Java1.5 for**

As illustrated in Code Example 3.6, Java1.5 provides a similar construct in an amended form of the `for` statement [Bra02].

```java
for(String s : list) { System.out.println(s); }
```

**Code Example 3.6: Java1.5 Iteration**
This form is functionally equivalent to the `foreach` form specified in Code Example 3.4, it specifies the type of a variable to be used within the body of the statement, and it specifies the list to iterate over. Java 1.5 also provides an overloading that allows the use of arrays in place of general collections.

**Maya foreach**
Maya provides a similar construct [Bak01, BH02], but chooses to mimic method call syntax as shown in Code Example 3.7.

```java
list.foreach(String s) { System.out.println(s); }
```

*Code Example 3.7: Maya Iteration*

One of Maya’s `foreach` forms is functionally equivalent to the form proposed here as it operates on an iterable object, a formal parameter, and a code block. Further to the treatment of arrays in the Java 1.5 implementation, Maya provides a specialised version for arrays, vectors, and iterators.

Maya uses overloading to address fundamentally different goals: the array overloading provides support for a *different* type, the iterator overloading provides support for iterators that may not come from a class derived from `Collection`, and the vector overloading provides an *optimised* version for a type that would otherwise be handled by the most generalised version.

### 3.4.1.3 Typesafe Formatted Output

Many languages provide a function that yields formatted output. The first argument to this function is a string (in typical use, a literal string) that specifies what normal text to output, and, using special placeholders, where to output variable data. These variables are specified as an arbitrary number of arguments following the string.

```c
printf("It took %f seconds to perform %d runs.", millis, runs);
```

*Code Example 3.8: Typesafe Formatted Output*

In Code Example 3.8, upon execution `printf` will convert the variable `millis` to a real number, and the variable `runs` to an integer at the specified positions within the format string.

Typical implementations of these formatted output functions allow placeholders for floating-point numbers, integers, characters, and strings, and provides special formatting
operations on these basic types. For example it is possible to force floating-point output to a specified number of decimal places.

A well-defined version performs checks to see if the correct number of arguments are supplied for the number of placeholders given and that the type of each of these arguments matches the corresponding placeholder. In the best implementations these checks are performed at compile-time. This is an example used in many other systems to demonstrate their power, see [SPJ02, Bak01].

**Necessary Language Features**

This construct requires us to be able to take a compile-time literal string and inspect it, and to be able to verify that the types specified within match the types of an arbitrary number of specified actual parameters. Also, we must be able to create new string literals and fully create a new series of instructions to output the created string.

**C/C++ printf**

C/C++ provides a standard library feature `printf` [Pla92§12], that matches the syntax from the previous section. However, the C/C++ version has many flaws. For example, no checking is performed on the types or the arguments supplied, or even whether the correct number is supplied — the latter is particularly poor as too few arguments to `printf` can cause program crashes. Indeed, it is not possible within C/C++ itself to provide `printf` as defined here with these problems remedied. C/C++ is simply not powerful enough to inspect the structure of a literal string at compile-time.

As a result, some compilers provide direct support for the `printf` function and inspect the string literal and the number and types of the arguments (e.g. [Sta’04]). This again illustrates the tendency for compiler-writers to explicitly provide support for language deficiencies.

**Java1.5 printf**

In the Java1.5 specification, direct support has been added for arbitrary numbers of parameters (in the form of “Varargs”, [Bra02]); due to this, a function very similar to

---

4 It may be possible to define a series of macros that produce a printf-like result, or use run time type information (RTTI) to check the types of the arguments, but the fundamental structure of the function would have to be altered.
the C++ printf is now provided, with an even more extensive selection of placeholders.

Unlike the C++ printf, the Java version illustrated in Code Example 3.9 will catch all errors in argument number and type, but will only do so at run-time. This is clearly undesirable for every call of printf that uses a literal string. For such calls, it is possible at compile-time to check every aspect of the call, and give a guarantee of correctness.

```
System.out.printf("%f %c %s", 45.3, 'q', new SomeObject());
```

**Code Example 3.9: Java1.5 Typesafe Formatted Output**

It is worth noting that Java1.4 does not provide facilities for passing an arbitrary number of arguments to a function. Arbitrary length argument lists were added to Java1.5 specifically to provide adequate support for functions such as printf.

**Maya printf**

The Maya implementation of printf [Bak01§5.1.4] provides a subset of the functionality of the Java1.5 version, but it forces the format string to be literal, and performs all checks at compile-time.

### 3.4.2 Complex Constructs

The constructs provided here are more sophisticated extensions than the simple statements from section 3.4.1, they provide facilities that range from expression extensions, to embedded languages and new flows of control.

#### 3.4.2.1 SQL Subset

As demonstrated in section 2.3.1.2, SQL is often added to a language via the normal library system by using embedded SQL strings. A better solution is to provide this functionality via syntax extensions, such as those described in section 2.4.2.3.

Whereas previously discussed syntax extensions required modification of a base language, or creation of a new language (e.g. Pro*C), an extensible language can provide these through its internal syntax extension mechanism.

Rather than provide SQL in the exact same fashion as specified in its standard [ANS89], it is possible to provide some of the SQL statements as an extension to the normal
expression mechanism, so that it is possible to write examples such as Code Example 3.10.

```java
Vector v = SELECT name
    FROM Employee
    WHERE Salary < 30000;
System.out.println("salary is: "+ SELECT salary
    FROM Employee
    WHERE TFN = :x);
```

Code Example 3.10: Embedded SQL Subset

This limited version of the SQL `SELECT` statement is a very small subset of SQL. It has been chosen to demonstrate the feasibility of embedding SQL in such a fashion – not to provide a usable SQL implementation.

**SQL Subset Grammar**

The SQL subset grammar is shown in Figure 3.1.

```
subset ::= SELECT names FROM tables [WHERE condition]
names ::= name | names , , , name | *
tables ::= table | tables , , , table
name ::= identifier | name , , identifier
table ::= identifier
condition ::= condition AND condition | condition OR condition
        | expr | ( ( ( condition )) )
expr ::= simple < simple | simple > simple | simple = simple
simple ::= name | :: java_expression
```

Figure 3.1: SQL Subset Grammar

Within the `java_expression` part it is possible to place any arbitrary Java expression (even another `SELECT` statement) but this expression cannot use terms local to the current SQL statement, only the surrounding context.

**Necessary Language Features**

This extension requires only an addition to the expression grammar. It requires the ability to specify grammars for lists, it also requires the overloading of part of Java expression grammar to create its own limited expression concept.

Also it requires the use of * and : in different contexts than how they are used in Java.

**3.4.2.2 Generators**

Generators, as discussed in section 2.3.3.3, are an interesting extension as they require a different evaluation strategy.
Generators can be provided as an extension, in a much more limited form than in Icon, by tagging each generator with a new modifier `generator`, with just the addition of the `suspend` statement, and the extension of the `foreach` construct to iterate through the generated values.

For example, this extension applied to Java, would allow code similar to that of Code Example 3.11.

```java
generator int fib() {
    int x = 0;
    int y = 1;
    while (true) {
        suspend y;
        x = x + y;
        suspend x;
        y = x + y;
    }
    ...
    forall x in fib() { System.out.println(x); }
}
```

**Code Example 3.11: Java Generators**

**Necessary Language Features**

In languages without a switch statement similar to that of C++, translating a generator’s body into restartable code is far from trivial, and requires sophisticated code manipulation techniques. This translation must take place on the entire function, translation of the `suspend` statements cannot take place until their context is known. The `forall` statement must be specialised to work with whatever the translated representation of the generator is.

### 3.4.2.3 Haskell Subset

There are many approaches to providing imperative forms within a pure functional language. A somewhat unexplored technique is to provide an embedded functional language within a mainstream imperative language. This approach has merits (eg. C++ is used in different ways to support multiple paradigms) — done well it allows for the best of two worlds, both languages can be used for their strengths.

Of primary interest would be the ability to switch between imperative evaluation and lazy evaluation at the user’s behest. Whilst much progress has been made in precisely this area by the functional programming community, current solutions still provide significant initial barriers to use [PJ02].
The general idea is to allow for functional declarations within Java code, and to provide a limited form of calling Haskell functions within Java expressions as demonstrated in Code Example 3.12.

```
fun {
    // insert haskell subset declarations here
}
...
Vector v = fun(take 5 fib)
```

**Code Example 3.12: Embedded Haskell Subset**

In the following subsections we describe a limited Haskell subset that nonetheless is quite expressive. This subset is designed around simplicity over functionality. Features that would potentially add a large implementation cost are avoided wherever possible. The aim is to demonstrate the feasibility of such an embedding, not to provide a full implementation of Haskell.

**Type System**

For simplicity, the subset handles only three types, \texttt{int}, \texttt{* -> *}, and \texttt{[*]} where \texttt{*} can be any type. Whilst this is significantly restricted from full Haskell, it is more expressive than it appears at first glance. The many occurrences of \texttt{*} mean that we can build up quite a complicated set of types. For example, using this type system, with an otherwise full Haskell implementation we could still define the function \texttt{map} in the standard way as shown in Code Example 3.13.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

**Code Example 3.13: map Function**

This definition in fact, is no different to a definition found in standard Haskell, despite the fact that it is technically defined over a much simpler range of types.

This is perhaps cheating a little, as we shall see in the following subsections, since the basic primitives of this subset do not provide for declarations with functions or for pattern matching, this is quite simple syntactic sugar than can be added later, it is simply unnecessary in this reduced definition.
Type Inference
Type inference, while a nice feature of Haskell, and by no means unimplementable within language extensions, is not part of the subset for simplicity. Their remains no barrier for it to be added at a later date.

All declarations are required to explicitly specify their type. As a result the previous definition of \( \text{map} \) must become as shown in Code Example 3.14.

<table>
<thead>
<tr>
<th>Code Example 3.14: ( \text{map} ) Function with Type Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{map} :: ((\text{A} \rightarrow \text{B}) \rightarrow [\text{A}]) \rightarrow [\text{B}] )</td>
</tr>
<tr>
<td>( \text{map} \ f \ [] = [] )</td>
</tr>
<tr>
<td>( \text{map} \ f \ (x:xs) = f \ x : \text{map} \ f \ xs )</td>
</tr>
</tbody>
</table>

The type signature is still checked against the type of all arguments, so strong-typing is not lost.

Simple Operations
The subset contains parameter-less declarations, lambda functions, if-then-else expressions, and let expressions.

Arithmetic Operators
Operations provided on our single basic type are merely the four basic arithmetic operations: +, −, *, and /. With the division operations discarding its remainder.

List Operations
The subset provides only : and [] for list creation, everything else is syntactic sugar and can be added later. Also provided are the list inspector functions head and tail, which respectively provide the first element of a list and the remainder.

Pattern Matching
Pattern matching is not in the subset, but again is merely syntactic sugar, albeit an incredibly concise one. For example, the \( \text{map} \) example can be rewritten (in an automatic way similar to that in [PJ99]) as in Code Example 3.15.

Notice that it is clear that this code could be simplified (and the final error case removed), but this would actually be considered an optimisation step, and not part of the automatic conversion from a pattern-matching form.


### Declarative

In this subset, declarations take the form of an identifier being held equivalent to a value, the value of course can be arbitrarily complex. This does however disallow parameters, so our somewhat worn map example now becomes as in Code Example 3.16.

```haskell
map :: ((A -> B) -> [A]) -> [B]
map f xs' =
  if (xs' == []) then
    []
  else
    if (xs' != []) then let x = head xs' in let xs = tail xs' in
    f x : map f xs
  else
    error -- no case satisfies
```

**Code Example 3.15: map Function Without Pattern Matching**

All functions must be defined before they can be used, this is restriction added only for simplicity, it does not indicate a failure of power in the macro language.

### Standard Functions

No standard functions are part of the subset definition, it is part of the proof-by-implementation to define functions such as `map`, `take`, `foldr`, etc. in these specified primitives.

### Other “Missing” Features

Perhaps the most seemingly restrictive of the missing features is the lack of a Boolean type. Use of the conditional `if` expression functions as normal, but use of Boolean expressions is restricted to this situation alone. However all typical Boolean usages can be simulated using integers if required; although this clearly is not a desirable permanent solution it is appropriate for this proof-of-concept implementation.

The following are relatively straightforward additions via expansion to the provided standard constructs:
• list constructions;
• tuples; and
• list comprehensions.

A little more extravagant are:
• type classes; and
• monads, and do notation.

See [PJ87] for an explanation of how to express the majority of Haskell in more primitive Haskell constructs. An example of this is the simple list comprehension in Code Example 3.17 (similar to [PJ99§3.11]).

```haskell
squares xs = [x * x | x <- xs]
squares xs = map (\x -> x * x) xs
```

Code Example 3.17: List Comprehension Decomposition

The general case for list comprehensions is a little more (but not overly) involved.

**Haskell Subset Grammar**

The Haskell subset grammar is specified in Figure 3.2.

```
subset = (declaration)+
declarion ::= signature definition
signature ::= identifier :: :: :: :: type
definition ::= identifier == == fun_expr
type ::= int | [ type ] | ( type -> type ) | A | ... | Z
fun_expr ::= expr | list | fun
fun ::= expr expr | fun expr
list ::= [] | expr : fun_expr
expr ::= if bExpr then fun_expr else fun_expr
      \ identifier -> fun_expr
      let identifier = fun_expr in fun_expr
      fun_expr operator fun_expr
      ( fun_expr )
      identifier
      literal
      [ ]
operator ::= + | - | * | /
bExpr ::= expr bOp expr
      | bExpr lOp bExpr
bOp ::= > | < | == | /=
lOp ::= && | ||
```

Figure 3.2: Haskell Subset Grammar

The Java embedding is specified in Figure 3.3.
Lazy Evaluation
Once the a particular group of definitions has been recognised as belonging to this subset, an implementation is free to produce compiled code, or to use interpretation in order to simulate lazy evaluation. A key reason to implement a Haskell subset is to demonstrate mixed imperative and lazy evaluation.

Necessary Language Features
Of all the extensions in these test cases this extension requires the most power to implement. The subset grammar is complex and overlaps quite heavily with other Java code, for example, any parser has a fair bit of work to do, in order to differentiate between a Haskell expression and a Java expression.

The most powerful implementation of this subset would also allow for files that contained nothing but code written in the subset. This would demonstrate the capacity of an extensible language to completely rewrite the base language.
Review of Extensible Languages

1. Introduction
2. Defining extensibility
3. Assessing extensibility
4. Extensibility review
5. Designing the language
6. Reviewing parsing
7. Implementing a parser
8. Implementing the language
9. Evaluation
10. Conclusion
4.1 Overview

Now that a basis for reviewing the successfulness of a given extensible language has been established, in this section we examine six successful meta-programming languages: Lisp and its variants, Template Haskell, MS\(^2\), JTS, JSE, OpenJava, and Maya.

- Lisp, and also Scheme, (section 4.2) macros are so successful and so widely used by that programming community that all research into creating new extensible languages is essentially trying to reproduce this success. Lisp/Scheme has the honour of being both the first implementer of syntax macros and also the most successful.

- Template Haskell (section 4.3) has shown that it is possible to nicely define Lisp style macros within a pure functional language, it gathers a large variety of previous work and reproduces it in one coherent system.

- MS\(^2\), (section 4.4) is an extension of the C programming language and provides strong facilities for producing concise macros. The major drawback of this system is its overly symbolic syntax which can be blamed on its C heritage.

- We review multiple Java-based systems: the Jakarta Tool Suite (JTS), the Java Syntax Extender (JSE), OpenJava, and Maya. These systems vary wildly in their suitability for comparison. Of these, Maya (section 4.8) provides the most powerful macro system. It improves upon previous Java extensibility research by removing restrictions on the placement of macros without sacrificing power. As the best implementation in the target language of this work, Maya is a natural choice for qualitative comparison (see section 9.4).

Most of these three systems share certain common aspects:

- they provide quasi-quote, unquote, automatic hygiene, and referential transparency;

- each provide an abstract syntax written in the target language itself, where the easy to use quasi-quotation mechanism fails, it is always possible to build arbitrary programs through this mechanism; and

- they are all implemented in well-established languages.
Where they differ is more interesting:

- they each provide different limitations on where macros can appear within code;
- each allows a different level of interaction between macros; and
- some provide no surrounding context information, whereas others provide the entire environment.

Also interesting is that most of these languages do not adequately meet our requirements for extensibility, namely the creation of arbitrary syntax. Even Lisp macros must conform to Lisp’s own rigid prefix-notation syntax restraints.

This chapter examines only the most relevant systems, many others are not examined (see Camlp4 [Rau03], MacroML [GST01], EPP [Ich99], JPP [Sha96b], <bigwig> [BS02], MPC++ [IHSM+96], ELIDE [BCVM02], and [CMA94]), but all of these have been previously reviewed in [GH03], [BP01], [BH02], [Bak01], and [BLS98]).

These systems have been chosen both to provide examples of meta-programming in various base languages, and also to look at previous approaches at providing meta-programming in Java. All of these systems are examined against the criteria for extensibility from section 3.3.
4.2 Lisp / Scheme

The Lisp (an acronym for list processing) programming language\(^5\) [Mcc60, Ste90] introduced the concept of meta-programming since its inception. Modern Lisp macro systems (especially Scheme [Dyb03]) provide facilities that bear little resemblance to their origins, although they do still support the original style of programming. This brief review looks at the development of the most commonly used Lisp macro facilities and does not attempt to provide a comprehensive examination of all available Lisp variants and their facilities. To this end we examine only Common Lisp (hereto referred to merely as Lisp) and Scheme.

Macros are identified in Scheme as a solution to a variety of problems:

> "Syntactic extensions, or macros, are used to simplify and regularize repeated patterns in a program, to introduce syntactic forms with new evaluation rules, and to perform transformations that help make programs more efficient." [Dyb03§8]

These uses are not new, as Lisp macros have existed for decades [SPJ02]. There would be few Lisp programmers who are unaware of the power that macros provide, and many apparent language extensions are implemented via macros. Macros are taken so seriously in fact, that when undertaking a new project, the standard philosophy of the Lisp programmer is to first modify the language to suit the needs of the project:

> "… modern Scheme systems support elaborate towers of language extensions based entirely on macros." [SPJ02§10.2, pp. 12]

Whilst the techniques behind Lisp macros have evolved and been improved with time, they still rely on the fact that every Lisp program is also a Lisp S-expression [Mcc60]. Other macro systems that provide an elaborate set of datatypes to provide an abstract syntax for meta-programming are mimicking this ability.

Within the Lisp community there is no general consensus as to which dialect provides the better macro system. There are far too many dialects to give them all full treatment

\(^5\) Despite its origin as a single programming language, in more recent usage, the term Lisp has perhaps come to describe a family of related languages.
in this section. Common Lisp is chosen to demonstrate early development of Lisp (even though it is a mature language itself) and Scheme is chosen to demonstrate the development of more powerful features.

Lisp was definitely the pioneering language for meta-programming and each of the systems in the remainder of this chapter are trying to reproduce the power and flexibility of Lisp macro constructs.

4.2.1 Power

At the heart of Lisp’s power is the S-expression. Its use eventually lead to the development of macros: meta-programs that could write other programs.

4.2.1.1 S-expressions

Every Lisp program is also a Lisp S-expression. An S-expression is a simple tree datatype. For example, consider the Lisp program fragment in Code Example 4.1. First note that all Lisp functions are written in prefix notation. This program text is converted into an equivalent S-expression that represents it in a tree form as shown in Figure 4.1.

\[
(+ 1 2 (\text{somefunc} 42 (- x y)) 3)
\]

Code Example 4.1: Simple Lisp Program Fragment

![Tree representation of S-expression]

Figure 4.1: Lisp S-expression for Code Example 4.1

If we wished to write a function to produce this program we could write the function in Code Example 4.2, where the quote expression ensures that we do not calculate any part of the expression. This quote is just a shorthand, Lisp allows us to construct an S-expression manually using its list constructor primitives.
Lisp provides run-time meta-programming support via the `eval` function, which will evaluate any S-expression at run-time.

Lisp functions can take an arbitrary number of arguments and each argument can be an arbitrary tree. The function is free to interpret its arguments in any way. Through this mechanism it is possible to provide new syntax. For example, consider the code fragment in Code Example 4.3.

```
(infix_exp 5 * 9 - 12 / 4)
```

**Code Example 4.3: Lisp Simulated Infix Expressions**

We could define `infix_exp` to take an arbitrary number of parameters and then parse this ourselves to provide infix notation with precedences. This function is far from perfect and we could foil it by using parentheses as anything within them would be evaluated as a lisp expression before the function call. More advanced macro definitions can improve on this situation.

### 4.2.1.2 Macros: `defmacro`

Consider the two definitions in Code Example 4.4 that take two arguments and only return the first.

```
(defun just-first-function (x y) x)
(defmacro just-first-macro (x y) x)
```

**Code Example 4.4: `defmacro`**

One major difference between these two definitions is what happens upon calling. The function version will evaluate all of its arguments, whereas the macro version will pass them through as S-expressions.

A macro defined in this way should return a replacement S-expression that takes the place of the original macro call.

Both brevity and implementation are improved by the use of `defmacro`: the programmer is alleviated from explicitly specifying which arguments should be S-expressions and explicit use of `eval`; while the removal of `eval` allows for the possibility of compile-time evaluation (although macros could still be implemented as a run-time facility).
4.2.1.3 Macros: define-syntax

Scheme provides a define-syntax macro that is similar to defmacro. Coupled with the use of a syntax-rules macro for pattern-matching with pattern translation and the result is a powerful, yet simple to use, macro system.

Whilst the code in Code Example 4.5 is more verbose than the previous definition of this macro in Code Example 4.4, this is only due to its simplistic nature. Understanding of hygiene (covered in the following subsections) is necessary to comprehend why define-syntax is an improvement upon defmacro.

```
(define-syntax just-first
  (syntax-rules ()
    ((just-first x y) x)))
```

Code Example 4.5: define-syntax

The macro syntax-rules is the most high-level facility provided, and also the least powerful. It provides the user with an environment free from the possibility of unexpected errors but at the cost of expressiveness. There are other facilities for macro definition, some with only limited applicability, others with increased power.

“The language of patterns and templates recognized by syntax-rules… is actually a special case of Scheme macros.” [HM04§5.4, pp. 21]

The general form of the define-syntax macro is shown in Code Example 4.6.

```
(define-syntax some-name
  (lambda (stx)
    ...))
```

Code Example 4.6: define-syntax General Form

The argument to the lambda function is similar to a Lisp S-expression but with added information about variables and scope. Full macros are written through the use of another macro called syntax-case which allows code generation in a more general form than syntax-rules:

“The syntax-case facility allows the construction of macros with pattern matching, as with syntax-rules… but with arbitrary expressions in place of templates for the result expressions.” [HM04§5.4, pp. 21]

This macro allows for pattern matching like syntax-rules, but with arbitrary expressions instead of pattern templates for the resultant code. As a result, it is possible
to allow macros to control their expansion based on properties of their arguments, in much the same way as \texttt{defmacro} style code. Again, the major improvement is hidden from us until we examine usability and error handling.

### 4.2.1.4 Syntax Creation

As previously discussed, Lisp programs are S-expressions. Lisp macros must be defined in a prefix form, but can define new syntax within their arguments — syntax that does not fit the S-expression style would need to be manually parsed.

So, Lisp is not capable of true syntax creation. For example, it is not possible to create a macro that would remove the need for large amounts of closing parentheses that affect most Lisp programs. In fact complex macros tend to add to this problem rather than alleviate it. A system capable of true syntax creation could specify many operations that would not require parentheses at all.

### 4.2.2 Usability

Lisp programmers found they were producing cumbersome code when meta-programming, so the \texttt{quasi-quote} and \texttt{unquote} operators were developed [Ste90].

Quasi-quote provides a similar function to the quote operator introduced in section 4.2.1.1, but allows for the user to escape the quote with the comma operator (generally called \textit{unquoting}).

Consider the code fragment in Code Example 4.7. This macro will create an S-expression that adds its argument to one. The necessity of unquoting may not be immediately apparent; if the unquoting was not present however, the quasi-quotation would include the token $x$, rather than the value bound to the variable $x$.

Code Example 4.8 demonstrates that the classic variable swap example suffers from the same problems in Lisp as it does in C++ macros.

\begin{verbatim}
(defmacro defmacro defmacro defmacro 1+ (x) (`(+ 1 ,x)))
\end{verbatim}

\textbf{Code Example 4.7: Quasi-quote and Unquote}

\begin{verbatim}
(defmacro defmacro defmacro defmacro swap (x y)
  (let ((temp ,x))
    (setf ,x ,y)
    (setf ,y temp)))))
\end{verbatim}

\textbf{Code Example 4.8: swap Function}
At first glance this macro definition seems straightforward, but it will fail if an attempt is made to use it with a variable named `temp`.

It was within the Lisp domain that name capture problems first appeared as a major drawback [Ste90], rather than a minor annoyance, and over the years Lisp systems produced a variety of solutions. These will now be expanded.

### 4.2.2.1 Name Capture: gensym

Name capture was initially handled by requiring the programmer to manually create unique names through the `gensym` function [Dyb03]. Whilst this is a successful solution, it requires the programmer to bear the burden, and program readability inevitably suffers.

Code Example 4.9 corrects the `swap` macro from the previous subsection.

```lisp
defmacro swap (x y)
  (let ((temp (gensym)))
    (let ((,temp ,x))
      (setf ,x ,y)
      (setf ,y ,temp)))
```

**Code Example 4.9: Improved swap Function**

This example is only slightly less readable than before using `gensym`, but the amount of unquoting has increased and this becomes more of a problem as code complexity increases.

### 4.2.2.2 Name Capture: define-syntax Revisited

Name capture problems eventually lead to the development of hygiene:

“Early designs suffered badly from the name-capture problem, but this problem was solved by the evolution of “hygienic” macros.” [SPJ02]

Code Example 4.10 contains a `syntax-rules` based implementation of `swap` that demonstrates the use of hygienic macros.

```lisp
(define-syntax swap
  (syntax-rules ()
    ((swap x y)
      (let ((tmp x))
        (set! x y)
        (set! y tmp)))))
```

**Code Example 4.10: Hygienic swap Function**
In this example `set!` is equivalent to `setf` from earlier examples — Scheme has a different naming convention from standard Lisp (side-effect operators have an exclamation mark). This code also does not use quasi-quotation and requires none of the unquoting that is required in the standard Lisp version.

More importantly, the introduction of the variable `tmp` is guaranteed not to cause name conflicts with the surrounding context. Scheme tracks variable declarations within macros and automatically renames any created variables.

Use of the more powerful `syntax-case` macro requires us to use syntax quasi-quotation, but again quoted expressions create S-expression with added contextual information. A similar syntax is provided for these syntax-quotations and the rules of hygiene still apply however, as demonstrated in Code Example 4.11.

```scheme
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(x y) #'(let ((tmp x))
              (set! x y)
              (set! y tmp))]))
)
```

Code Example 4.11: `swap` Function using `syntax-case`

Again, with even this simple example the code seems overly verbose, but the pattern-matching of macro arguments is a powerful feature for more complicated macros.

### 4.2.3 Error Handling

Error handling may be handled differently amongst Lisp dialects, but Scheme at least has quite extensive facilities. Here we see the major purpose of retaining contextual information alongside S-expressions:

“[it] is essential in allowing… language tools to trace errors and binding relationships back to the original source location in the user’s code where a macro is invoked.” [HM04§5.4, pp. 21]

The extra power of the `syntax-case` macro allows us to make decisions based on the compile-time structure of arguments, and provide explicit error control.

In Code Example 4.12, the purpose of the macro `raise-syntax-error` is clear, and its arguments contain the macro that caused the error, a helpful message and the
syntax object that contains the contextual information. This lexical information is used by the error macro to highlight the original source of the error [Dyb03].

```
define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(x y)
        (if (and (identifier? #'x) (identifier? #'y))
          #'(let ((tmp x))
              (set! x y)
              (set! y tmp))
          (raise-syntactic-error 'swap "needs identifiers" stx))))))))
```

**Code Example 4.12:** swap Function with Error Handling

### 4.2.4 Applicability to Benchmarks

Many of the examples in the benchmark test suite are hindered by the inability of Lisp to truly create new syntax, but for the sake of the simple macros we shall ignore requirements of extraneous bracketing forms in macro usage.

#### Table 4.1: Lisp Applicability to Benchmark Test Suite

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 assert</td>
<td>Simple definition.</td>
</tr>
<tr>
<td>2 foreach</td>
<td>Simple definition. Proposed syntax can be matched exactly.</td>
</tr>
<tr>
<td>3 printf</td>
<td>Simple definition. Can even specialise on the static-type of the string argument, generating code when it is static and calling a run-time function otherwise.</td>
</tr>
<tr>
<td>4 SQL</td>
<td>Possible to provide SQL support, but matching SQL syntax exactly would require manual parsing of an essentially flat S-expression.</td>
</tr>
<tr>
<td>5 Generators</td>
<td>Generators can be provided.</td>
</tr>
<tr>
<td>6 Haskell</td>
<td>While strictly speaking embedded Haskell would be possible, it would again rely heavily on manual parsing.</td>
</tr>
</tbody>
</table>
### 4.2.5 Extensibility Criteria Assessment

Table 4.2: Lisp Extensibility Criteria Assessment

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Syntax Creation</td>
<td>Arbitrary syntax creation is not provided for. All macros are defined with standard Lisp prefix notation — in general they are indistinguishable from normal Lisp functions. S-expressions still drive all macro expansion which causes macros to have strict syntactic limitations. The simplest example of this limitation is the inability to define macros that alleviate (or lessen) the Lisp closing parentheses problem.</td>
</tr>
<tr>
<td>1.2 Syntax Interrogation</td>
<td>Interrogation is provided through S-expressions. This is an extremely powerful and flexible system.</td>
</tr>
<tr>
<td>1.3 Syntax Overloading</td>
<td>Partially supported. Although <code>syntax-rules</code> does allow for keywords in the latter part of a definition to drive different expansions, overloading cannot be provided to replace previous or built-in definitions.</td>
</tr>
<tr>
<td>1.4 Static Type Interrogation</td>
<td>Supported.</td>
</tr>
<tr>
<td>1.5 Expressiveness</td>
<td>Within the confines of Lisp syntax, Lisp macros can be used for almost any purpose. However, The programmer cannot embed another language within Lisp with correct syntax.</td>
</tr>
<tr>
<td>2.1 Simplicity</td>
<td>Lisp macros are close to Lisp code, which is simple in itself.</td>
</tr>
<tr>
<td>2.2 Brevity</td>
<td>Lisp provides quasi-quotation and unquoting.</td>
</tr>
<tr>
<td>2.3 Robustness</td>
<td>Depending on implementation, users may be burdened with explicit removal of name clashes.</td>
</tr>
</tbody>
</table>
3.1 **Syntactic Correctness**

Macros are guaranteed to produce S-expressions, which, while syntactically correct, may not be meaningful Lisp code.

3.2 **Error Detection**

Programmers can specify explicit checks and raise errors based on the results.

3.3 **Error Reporting**

Scheme, at least, provides sophisticated targeting of error messages with facilities provided to specify the exact original source of errors.
4.3 Template Haskell

Extending Haskell to support extensibility is in some ways easier than it is in Lisp/Scheme, due to well-known benefits of pure functional languages:

“Scheme admits side effects, which complicates everything… Haskell is free of these complications.” [SPJ02§10.2, pp. 13]

Template Haskell [SPJ02] provides the ability to execute Haskell functions at compile-time, called splicing, by use of the operator $ and the function splice (explained further in section 4.3.2.1).

Meta-programs are constructed in various ways, either through algebraic datatypes via a monad, a set of abstractions built on top of this monad, or using quasi-quotation. These three techniques each provide a simplicity/expressiveness trade-off; most programs will be written in quasi-quotation, but for those that cannot be expressed in this fashion, the programmer will need to fall back to the other techniques.

Limited facilities are provided for the inspection of the compiler’s internal structures via reification (explained further in section 4.3.1.1).

4.3.1 Power

Template Haskell provides a set of Haskell algebraic datatypes that represent an abstract syntax. Programmers are free to write normal Haskell functions that manipulate these datatypes directly, either constructing new syntax, or deconstructing existing syntax using Haskell’s powerful pattern matching techniques.

Calling functions at compile time (splicing) requires explicit annotation (covered more in section 4.3.2), and can only appear where an expression or a declaration group would be expected.

“A meta-program can produce a group of declarations, including data type, class, or instance declarations, as well as an expression.” [SPJ02§2, pp. 2]

In many languages this would be unnecessarily restrictive, but in Haskell this covers the majority of possible uses of meta-programming.
4.3.1.1 Reification

Haskell provides compile-time information through its *reification* mechanism:

“Reification involves making the internal representation of [compiler objects] available as a data structure to compile-time computations.” [SPJ02§4, pp. 4]

This mechanism provides facilities for querying the structure of declarations (both datatypes and type classes), the type of declarations, the fixity of operators, and the line number of the statement within the source file.

It is an open question as to which situations reification can be applied. The design of Template Haskell does not specify if it is possible to query the type of variables within expressions, definitions within where clauses etc.:

“It is not yet clear how much reification can or should be allowed. For example, it might be useful to restrict the use of `reifyDecl` to type constructors, classes, or variables (e.g. functions) declared at the top level in the current module, or perhaps to just type constructors declared in data declarations in imported modules.” [SPJ02§8.1, pp. 9]

4.3.2 Usability

In addition to the basic algebraic datatypes defined for syntax creation, Template Haskell provides a quotation monad, that encapsulates meta-programming features such as unique name generation, error reporting and the program reification discussed in the previous subsection. A library of functions is provided within this monadic framework as an easy-to-use interface for the programmer.

Within the monadic library, a quasi-quote mechanism is provided. Inside quasi-quote expressions, Template Haskell performs static scoping and type-checking. It is only when using quasi-quotation that hygiene and referential transparency are assured.

The programmer is free to mix these meta-programming styles, choosing whichever is the most appropriate for each component of a particular computation. See section 4.3.4.2 for an example that mixes both quasi-quotation and monadic library functions.
4.3.2.1 Splicing

In Template Haskell the programmer has a little more work to do when actually using macros than in most systems; each macro call must be explicitly annotated to let the compiler know that the programmer wishes to execute said code at compile-time:

> “C++ template and Scheme macros have a lighter-weight syntax for calling a macro than we do; indeed the programmer may not need to be aware that a macro is involved at all.” [SPJ02§12, pp. 14]

Macro expansion must be prefixed by the splice operator $ or the splice function. For example, in order to use Template Haskell’s version of printf, code similar to that of Code Example 4.13(a) must be written. Here the printf function takes only a literal string as its argument, and produces a function that requires two arguments, (loosely) the first being a number and the second a string. So the result of macro expansion would be something akin to Code Example 4.13(b).

\[
\text{\$}\left(\text{printf } \%d \%s\right) 42 \text{ "foo"}
\]

(a) Explicit Use of the Splice Operator for printf Usage

\[
(\lambda x \rightarrow \lambda y \rightarrow \text{show} \ x \ +\ + \ " \ + \ + \ y) \ 42 \ \text{"foo"}
\]

(b) printf Macro Expansion

Code Example 4.13: Template Haskell printf Expansion

The type checking is actually delayed until the macro is expanded, and utilises the normal Haskell type-checking mechanism (see section 4.3.3 for more details).

The operator $ can appear anywhere that an expression is expected, and the result of applying it must always produce an expression. Since almost everything in Haskell is an expression, this operator is surprisingly versatile.

The function splice allows the programmer to write meta-programs that produce a group of declarations. This allows meta-programs to create type classes, data declaration, and functions.

At times, explicitly alerting the programmer that a macro call is involved can be viewed as advantageous, but in general this requires the programmer to understand more than they should need to. The Template Haskell design requires this because firstly:
“… functions that execute at compile-time are written in the same language as functions that execute at run-time, namely Haskell.” [SPJ02§3, pp. 3]

Secondly, absolutely no distinction is made between compile-time functions and run-time functions, indeed, the programmer can use template functions at run-time if they wish.

4.3.2.2 Quasi-quotation


```haskell
assert :: Expr         -- Bool -> a -> a
assert = [\ b r -> if if if b then then then then r else else else else error error error error ("Assert fail at "+$reifyLocn) ]
```

Code Example 4.14: Template Haskell Assertions

Quasi-quotations cannot appear within other quasi-quotations. For example, the form in Code Example 4.15 is illegal.

```
[ | f [ | 3 | ] | ]
```

Code Example 4.15: Illegal Template Haskell Quasi-quotation

However it is possible to use splice within a quasi-quote, and within that splice quasi-quotuation may be used again. Inside quasi-quotations the splice operator functions similar to unquote. See sections 4.3.4.1 and 4.3.4.2 for examples of interleaving of quasi-quotation and splicing.

4.3.3 Error Handling

The algebraic datatypes, the syntax creation monad, the monadic library, and the quasi-quotation mechanism all produce syntactically correct programs. In addition to this, it is possible for the programmer to explicitly detect some errors, such as inappropriate use of a meta-program. This support is provided via the monad: a meta-program can fail, allowing the compiler to catch and report such failures along with their location.

Further to this explicit error checking, Template Haskell interleaves execution of compile-time functions and type-checking. This ensures early detection of type errors and provides the user with good feedback on their location.
As previously mentioned, it is unclear as to the level of reification provided for static type checking, and hence it is unclear to the extent of explicit checking the programmer can provide.

### 4.3.4 Worked Examples

The following two worked examples demonstrate:

- a full definition of a macro and its support functions within a module, and usage of this module; and
- a macro that mixes both quasi-quotation and monadic library functions.

#### 4.3.4.1 Type-safe Formatted Output

Code Example 4.16 is drawn from both [SPJ02§2] and [GHC02§7.6] and provides a `printf` function similar to that described in section 3.4.1.3 but with marginally different syntax. It takes a single argument, that of the literal string and creates a function that requires arguments as specified by the placeholders in this string. For example, as shown in Code Example 4.13, the string “%d %s” produces a formatting function that requires a number then a string as its two arguments.

```haskell
{- Printf.hs -}
module Printf where

-- Import some Template Haskell syntax
import Language.Haskell.THSyntax

-- Describe a format string.
data Format = D | S | L String

-- Parse a (simple) format string.
parse :: String -> [Format]
-- implementation unwieldy and hence not provided

-- Generate Haskell code from parsed representation of a format string.
gen :: [Format] -> Expr -> Expr
gen [] x = x
gen (D : xs) x = [| \n-> $(gen xs [ | $x++show n |]) |]
gen (S : xs) x = [| \s-> $(gen xs [ | $x++s |]) |]
gen (L s : xs) x = gen xs [| $x ++ $(lift s) |]

-- Generate the Haskell code for the splice from an input format string.
printf :: String -> Expr
printf s  = gen (parse s) [| "" |]
```

**Code Example 4.16: Template Haskell `printf` Definition**

Code Example 4.16 demonstrates a Template Haskell implementation of `printf`. The function `parse` takes the format string and breaks it into a list of format specifiers. The function `gen` is responsible for the construction of the output function; `printf` merely
uses this with a suitable initial value. This latter function builds an expression from a list of format specifiers.

Here we see for the first time interleaving of quasi-quotatation and splicing. Splicing operates as one would intuitively expect in this situation, and gen behaves as a would a standard Haskell function [SPJ02].

On the last line of the definition of gen, the function lift turns its string argument into an expression that would evaluate into the original string — lift is actually provided via a type class and hence works on a variety of types. This is an example of the kind of function provided by the monadic library (see section 4.3.4.2 for an example using more of these functions).

Of particular interest here is the fact that the definitions of meta-programs differ only in that they use quasi-quote and splicing. In effect, these are shorthands, and this file could contain purely standard Haskell code. Even the type Expr is a standard monadic wrapper for the algebraic type Exp.

The cost of this simplicity is that expansions must be explicitly spliced in order to ensure compile-time execution. Code Example 4.17 shows how to import and use the printf function.

```haskell
{-# Main.hs #-}
module Main where

-- Import our template "printf"
import Printf ( printf )

-- The splice operator $ takes the Haskell source code generated at
-- compile-time by "printf" and splices it into the argument of "putStrLn".
main = putStrLn ( $(printf "Hello %d green %s") 42 "people")

Code Example 4.17: Template Haskell printf Usage

The qualified import statement restricts the import to just the printf function. Notice the explicit use of the splice operator to expand the printf form at compile-time. The expansion will produce a function that takes two arguments (a number and a string) and will produce a string.

### 4.3.4.2 Selection From an N-tuple

Code Example 4.18 demonstrates the use of both quasi-quotatation and monadic library functions.
**CHAPTER 4: REVIEW OF EXTENSIBLE LANGUAGES**

**Template Haskell**

Code Example 4.18: Template Haskell N-tuple Selection

This example allows the selection of an indexed member of an n-tuple — something impossible to do in ordinary Haskell. For example, the expression `(sel 2 3)` translates into a lambda function as shown in Code Example 4.19.

```
(sel 2 3)  
→ \x -> case x of (a1, a2, a3) -> a2
```

**Code Example 4.19: Template N-tuple Selection Expansion**

The variable `as` contains a list of strings to be used as variables. The variable `rhs` contains the selected variable to extract from the tuple. The variable `pat` constructs a pattern which is a tuple containing all of the strings from `as`. The variable `alt` specifies a list of alternatives (always containing a single alternative) for a case expression. The monadic library functions `caseE`, `simpleM`, `var`, `ptup`, and `pvar` all aid in simplifying this process. These functions create a case expression, a simple pattern matching expression, a simple variable expression, a tuple pattern, and a simple variable pattern.

Quasi-quotation in Template Haskell is not powerful enough to directly produce arbitrary tuples, and as such the programmer must rely on the previous layer. As a result, monadic library code is much more verbose and requires more effort in understanding than quasi-quotation code.

### 4.3.5 Applicability to Benchmarks

Template Haskell does not suit our benchmark suite particularly well as many of the examples simply aren’t appropriate for a functional language. However, it is still clear that the many constructs cannot be supported, unless we compromise syntactically from the benchmark suite test case definitions.
Table 4.3: Template Haskell Applicability to Benchmark Test Suite

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 assert</td>
<td>Simple definition with quasi-quote. Reification may provide facilities for command-line debug-build switching.</td>
</tr>
<tr>
<td>2 foreach</td>
<td>Not appropriate; standard functions provide iteration functionality.</td>
</tr>
<tr>
<td>3 printf</td>
<td>Simple definition with quasi-quote.</td>
</tr>
<tr>
<td>4 SQL</td>
<td>Possible to provide decent SQL support, but not possible to match syntax exactly.</td>
</tr>
<tr>
<td>5 Generators</td>
<td>Not appropriate in a functional language; lazy evaluation provides better facilities.</td>
</tr>
<tr>
<td>6 Haskell</td>
<td>Not appropriate.</td>
</tr>
</tbody>
</table>

### 4.3.6 Extensibility Criteria Assessment

Table 4.4: Template Haskell Extensibility Criteria Assessment

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Syntax Creation</td>
<td>Arbitrary syntax creation is not provided for. All compile-time functions are simply normal Haskell functions run at compile-time. Splice is only allowed anywhere an expression or a group of declarations occurs.</td>
</tr>
<tr>
<td>1.2 Syntax Interrogation</td>
<td>Interrogation of syntax is provided through an abstract syntax tree provided as a set of Haskell datatypes. Haskell provides excellent support for working with user defined datatypes, and this flows through to this abstract syntax.</td>
</tr>
<tr>
<td>1.3 Syntax Overloading</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>
## 1.4 Static Type Interrogation
Provided in a limited form via reification, but its exact scope is unclear.

## 1.5 Expressiveness
Template Haskell provides no facilities for new syntax creation. Its usefulness is limited to small scale additions to the language.

## 2.1 Simplicity
Meta-programs are programmed in Haskell, which itself tends to lead to simple programs. Each of the three layers of syntax creation are provided in standard Haskell idioms.

## 2.2 Brevity
Quasi-quotation and unquote are provided. The calling of compile-time functions is burdened with the necessity of explicit use of splice.

## 2.3 Robustness
Automatic hygiene and referential transparency are provided when using quasi-quotation. The function `gensym` is provided for explicit fresh name creation when working with the syntax creation monad.

## 3.1 Syntactic Correctness
Constructs created in any of the three syntax creation layers are guaranteed to be syntactically correct.

## 3.2 Error Detection
Errors in usage can be caught by the programmer, and reported to the system via Haskell’s standard `error` construct. Interleaving of parsing and type-checking provides strong support for type checking, although it is unclear whether the programmer can aid this process through reification.

## 3.3 Error Reporting
Due to the interleaving of parsing and type-checking Template Haskell detects errors as soon as possible, and consequentially has a good chance of providing useful error reporting.
4.4 Meta Syntactic Macro System

The Meta Syntactic Macro System (MS²) [WC93] extends the C programming language to provide meta-programming support.

4.4.1 Power

In this system macros are only allowed to appear as declarations, statements, or expressions. Each macro must begin with a name, but following this, the syntax is quite general. For example, it is possible to define a macro for enumerations that has the syntax of Code Example 4.20(a).

```c
new_enum color { red, green, blue };
```

(a) Enumeration Usage

```c
syntax decl new_enum { | $$id::name { $$++, id::ids }; | } {
    return `[ enum $name $ids; ];
}
```

(b) Enumeration Definition

Code Example 4.20: MS² Enumerations

An abstract syntax is provided for compile-time program manipulation. The abstract syntax has a limited set of forms and the user is not able to introduce new ones. No special facilities are provided for the interrogation of these trees.

4.4.2 Usability

This system provides multiple quasi-quote forms as well as an unquote operator. A general quasi-quote form is provided where the user must specify the type of the result as well as three shorthand forms for expressions, statements, and declarations. Each of these forms begins with a backquote but has a different bracket form.

Within these quasi-quotes the user is capable of specifying constructs without adhering to exacting concrete syntax. Consider the definition of `new_enum` in Code Example 4.20(b). Inside the quasi-quotation, the concrete syntax of the enumeration can be ignored; the programmer can merely specify the required parts, i.e. a name, and a list of identifiers. Other systems would typically require the user to work with the abstract syntax objects directly to provide this kind of functionality.
This system provides other programmer shorthands. In the above example the format list of identifiers is specified by a pattern that ensures that there is at least one identifier and that multiple identifiers are comma-separated. These pattern shorthands provide for:

- lists of zero or more, or one or more arguments;
- optional elements, either with or without a leading token; and
- tuples.

It was foreseen that much processing would be performed on these lists of abstract syntax trees, so support was directly introduced for anonymous functions so that functions such as map could be better supported. Anonymous functions reduce the programmer’s burden when using higher-order functions and as a result greatly facilitate the use this powerful, concise coding practice [WC93]. This in turn leads to meta-programs that more closely resemble the code they produce (for an example, see section 4.4.4.2).

These shorthands allow for concise definitions, their drawback being an obfuscating syntax that creates a significant barrier to understanding this system. The major source of this increased confusion is new bracketing forms; this system introduces six of these.

### 4.4.3 Error Handling

The system produces syntactically correct forms via its quasi-quotations but beyond that provides no support for error detection in expanded code:

> “The ease of debugging macros depends upon the quality of the debugger provided by the C programming environment being used.” [WC93§3, pp. 7]

### 4.4.4 Worked Examples

The following two examples demonstrate a simple definition of a statement macro, and a more complex definition which produces a list of declarations. Both of these examples demonstrate the brevity of definitions produced with this system.

#### 4.4.4.1 Dynamic Binding

Code Example 4.21(a) demonstrates the use of dynamic_bind to allow the temporary redefinition of the value of a variable. The idea here is that printLength
is a global variable and it is temporarily changed to the value 10, and reset after the function call within the block.

```plaintext
dynamic_bind {int printlength = 10} {
    print_class_structure(gym_class);
}
```

(a) Dynamic Binding Usage

```plaintext
syntax stmt dynamic_bind {|
    { $$type_spec::type $$id::name = $$exp::init }
    $$stmt::body |
} |
@id newname = gensym();
return `{
    $type $newname = $name;
    $name = $init;
    $body;
    $name = $newname;
}
}
```

(b) Dynamic Binding Definition

**Code Example 4.21: MS² Dynamic Binding**

The code for providing this macro is in Code Example 4.21(b). A macro is defined by use of the `syntax` keyword. The macro mimics a C function declaration in that it expects a type, then a name, followed by a list of parameters, although these parameters are contained within a `{ | . . . | }` pair rather than parentheses. The parameter list can contain an arbitrary number of terminal symbols as well as a series of formal arguments. Each formal argument is specified by the symbol `$$`, followed by the type, followed by `::`, followed by the variable name. There are shortcuts provided for lists of values (see the next subsection for more information).

In the declaration of the variable `newname`, we see that when Abstract Syntax Tree (AST) types are used within normal code they must be prefixed by `@`. Also, no provision for hygiene is made, and we must acquire unique names directly through use of the `gensym` function.

Following the return statement is statement quasi-quote `\{ . . . \}` and within this unquoting is specified by `$`.

Despite its obscure syntax, and the necessity of explicit name-capture avoidance, this definition is still as succinct as an equivalent definition would be in previously examined languages.


4.4.4.2 Extended Enumerations

Code Example 4.22 demonstrates an extension to enumerations that provide for automatic generation of input and output functions.

Code Example 4.22(a) and (b) show an example usage and its automatic expansion. This expansion is straight-forward, the creation of the enumeration requires no transformation, the output function requires a *case* for each enumeration element, and the input function requires an *if* statement for each element.

However, the code defining this macro in Code Example 4.22(c) is less than straight-forward.

On line 1 we define a macro that returns a list of declarations; the usual C-syntax for arrays is maintained. Within the { | ... | } block, we define two macro parameters: the first an identifier called *name*, and the second is a list of comma-separated identifiers called *ids*.

On line 3 we see the shorthand provided for creating lists. This list has three elements, the enumeration declaration (line 5), the print function(lines 8-19), and the read function (lines 22-34). Each of these declarations is defined within a declaration quasi-quote. The enumeration is defined without the need for concrete syntax — this is a general property of the macro system. The programmer is freed from knowing that the list of identifiers to an enumeration must be comma-separated and enclosed within braces.

Within the print function we have an interesting feature of this system, enclosed by the (| ... ) block is an anonymous function definition. Its syntax provides for a list of declarations that act as arguments, followed by an expression. Anonymous functions were specifically added to C’s existing function-pointer concept to allow for the easy use of functions such as *map*. In this case, *map* is used to create a list of cases from the list of enumeration elements.

The definition of the read function follows a similar format to that of the print function — again utilising *map* and an anonymous function to provide a brief definition.
myenum fruit { apple, banana, kiwi; }

(a) Printable Enumeration Usage

enum fruit { apple, banana, kiwi; }
void print_fruit(int arg) {
  switch (arg) {
    case apple: printf("%s", "apple");
    case banana: printf("%s", "banana");
    case kiwi: printf("%s", "kiwi");
  }
}
int read_fruit() {
  char s[100];
  getline(s, 100);
  if (!strcmp(s, "apple")) return(apple);
  if (!strcmp(s, "banana")) return(banana);
  if (!strcmp(s, "kiwi")) return(kiwi);
  return -1;
}

(b) Printable Enumeration Expansion

(c) Printable Enumeration Definition

Code Example 4.22: Printable Enumerations
4.4.5 Applicability to Benchmarks

This system does not provide for static type-checking and has limitations on macro definitions, and as a result its performance on the test suite is mixed.

Table 4.5: MS² Applicability to Benchmark Test Suite

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 assert</td>
<td>Simple definition provided a suitable standard library function exists to interrupt program execution.</td>
</tr>
<tr>
<td>2 foreach</td>
<td>Simple definition provided a set of C functions exist to mimic iterators. Syntax can be matched exactly, as terminals may appear anywhere in a macro pattern. Type checking on the type of the expression could not be provided by the macro itself.</td>
</tr>
<tr>
<td>3 printf</td>
<td>This system is not capable of providing the literal string argument for printf.</td>
</tr>
<tr>
<td>4 SQL</td>
<td>The syntax of the SELECT statement could almost be matched exactly, but would fail at the WHERE clause. It could only provide an arbitrary C expression, and provide no support for the mixing of SQL names and C names.</td>
</tr>
<tr>
<td>5 Generators</td>
<td>Not supported.</td>
</tr>
<tr>
<td>6 Haskell</td>
<td>Not supported due to the requirement that each macro begin with a name.</td>
</tr>
</tbody>
</table>
## Extensibility Criteria Assessment

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Syntax Creation</td>
<td>All macros must begin with a name, but following this, the programmer appears to be free to define new syntax, although some restrictions may apply, e.g. it appears that parentheses are not allowed within macro headers. Macros may only use a limited set of abstract syntax types in their argument lists. Macros can only appear as declarations, statements, or expressions.</td>
</tr>
<tr>
<td>1.2 Syntax Interrogation</td>
<td>Syntax interrogation provided via an abstract syntax with additions to normal C-syntax to provide for easy construction.</td>
</tr>
<tr>
<td>1.3 Syntax Overloading</td>
<td>Unspecified.</td>
</tr>
<tr>
<td>1.4 Static Type Interrogation</td>
<td>Not provided.</td>
</tr>
<tr>
<td>1.5 Expressiveness</td>
<td>It is only possible to use this system for small macros with a limited scope, due to strict rules on macro use and the requirement that macros begin with names.</td>
</tr>
<tr>
<td>2.1 Simplicity</td>
<td>This system introduces much for the programmer to understand with many symbolic additions to a syntax that already has too many symbols. Once these have been learnt, the system is relatively easy to use. However, it is unclear whether parsing conflicts can arise, and if so, how much burden they place on the macro programmer.</td>
</tr>
<tr>
<td>2.2 Brevity</td>
<td>A host of programmer shortcuts are provided; of particular use are the easing of restrictions in syntax within quasi-quotations and the use of patterns within</td>
</tr>
</tbody>
</table>
macro headers.

<table>
<thead>
<tr>
<th>2.3 Robustness</th>
<th>A <code>gensym</code> function is provided, but no attempt is made to provide hygiene.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Syntactic Correctness</td>
<td>All constructs are created within quasi-quotations and are guaranteed to be syntactically correct.</td>
</tr>
<tr>
<td>3.2 Error Detection</td>
<td>Not supported.</td>
</tr>
<tr>
<td>3.3 Error Reporting</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>
4.5 Jakarta Tool Suite

The Jakarta Tool Suite (JTS) [BLS98] is a set of pre-compiler tools for extending programming languages. Its meta-programming facilities exist only within a Java extension called Jak, which is in turn written in another tool called Bali.

Bali is in essence a parser-generator, such tools are common-place and a full description of this is outside the scope of this work.

Jak provides a quasi-quotation system, but for a limited set of constructs and the resultant type must be specified. The same applies for its unquoting mechanism. It is unclear whether quotations can be nested. Powerful tree traversal and editing functions are provided for code manipulation. An attempt at providing hygiene is made, but does not produce an automatic system; programmers are left to specify which variables are to have their names mangled to avoid collisions.

Jak provides code generation facilities, but it is unclear as to how they are to be utilised, it is left unspecified whether or not compile-time evaluation can take place. It is assumed that to define new syntax, or make language extensions one must define them in Bali and embed Jak code to perform any transformations.

Utilising a parser generator for extensions makes this system similar to that of an open compiler system (see section 2.3.2). As a result any extension should be possible, but each extension stands alone — we are in essence creating an entirely new language every time we make a minor change.

The Jakarta tool suite has several features common with the systems discussed in this chapter, but falls too far outside the domain of meta-programming and extensibility to be adequately compared to such systems. We make no attempt to test the applicability of this system to our benchmark suite, and do not rate this system against our extensibility criteria.
4.6 Java Syntax Extender

The Java Syntax Extender (JSE) [BP01] is a macro facility. Its implementation is as a pre-processor taking .jse files and producing .java files.

4.6.1 Power

JSE recognises that some systems allow for the creation of arbitrary syntax, but sacrifices this for simplicity:

“JSE is less ambitious in that it provides a convenient and powerful mechanism for extending the syntax in limited ways. In particular, it provides only a limited number of shapes and requires that macros must always commence with a name.” [BP01§8.4, pp. 40]

To this end, JSE provides limited shapes in the form of call macros and statement macros. A call macro mimics a simple Java method call and is available to appear where a statement or an expression would. A statement macro can only occur where normal statements would and have a more complicated form: optional modifiers and then a series of clauses.

\[
\begin{align*}
\text{call_macro} &::= \text{name}(...) \\
\text{statement_macro} &::= \text{modifiers \ [clause]+} \\
\text{clause} &::= \text{name \ ... \ terminator} \\
\text{terminator} &::= ; | } \\
\end{align*}
\]

Figure 4.2: JSE Call and Statement Macro Grammars

The loose grammar of Figure 4.2 specifies the structure of both call and statement macros. As we shall see, where the ellipses appear in this grammar is where JSE provides a deal of flexibility — this is discussed in the next subsection. The restriction on each macro ending with a semicolon or a brace is born of simplicity and a restriction that fits within Java well:

“Shapes serve to allow easy location of the end of a macro before handing it to the macro expander; shapes are a way to find “the closing bracket”.” [BP01§3.1, pp. 35]

For program manipulation, rather than a full abstract syntax, JSE provides a skeleton syntax tree (SST).
“In general, a SST has fewer categories than a typical AST and instead represents the basic shapes and distinctions necessary for macro processing.”

[BP01§2, pp. 32]

The downside to this approach is that no guarantees can be made as to the syntactic correctness of the programs produced.

Macros are expanded in an outside-in fashion which limits interoperability between macros. These expansions can contain macro calls; macro expansion continues until all macros are removed.

### 4.6.2 Usability

Each macro is defined as a class that implements an interface `SyntaxExpander`. Such definitions are unwieldy and require the programmer to perform many housekeeping tasks (see section 4.6.4).

Thankfully, a shorthand exists that is written within JSE itself, the `syntax` macro alleviates all of these housekeeping tasks and allows the programmer to concentrate on the macro definition (see section 4.6.4.2).

A macro is defined to take only one argument: a SST fragment. Within the actual body of the macro a `syntaxSwitch` construct is used to pattern match this fragment. This approach was modelled on Lisp and Dylan [Sha96a].

Patterns may contain an arbitrary number of terminals and non-terminals, and are also used to bind matches to variable names. JSE provides a rather odd shorthand for pattern names: if the user fails to name a parameter, the system automatically generates a default. For example, failing to name a type will lead to it being called `type`.

Patterns use a set of pre-defined constraints that allow for non-terminals: patterns accept names, types, expressions, statements (either a single semicolon terminated statement, or a block), bodies (an enforced block), and switch statement bodies.

Users are permitted to define new constraints, but it is unclear as to how free this process is. Also, there appears to be no direct way to introduce constraints within code, they must be included on Java’s `CLASSPATH` to be used.
JSE provides support for both automatic hygiene and referential transparency. A quasi-quotiation form is provided along with unquote. Support is provided for nesting of quotations.

### 4.6.3 Error Handling

The use of skeleton syntax trees prevents JSE from providing a guarantee that expansions will be syntactically correct. The user is provided with no mechanism for manually detecting and reporting errors although failure to match any case of the `syntaxSwitch` will cause a `SyntaxMatchFailure` exception to be thrown, it is supposed that the user could hijack this mechanism.

JSE does however make some attempt to provide users with debugging support. Facilities are provided to allow smart editors to perform macro expansion on program strings so that users can witness the result of macro expansion — one expansion at a time if they wish. When compiler errors occur, JSE provides the original source of the error. This provides programmers with feedback on the code they wrote themselves. It is claimed that this simple maintaining of the source location of macro calls gives “reasonable results” [BP01].

### 4.6.4 A Worked Example: `foreach`

In this section we provide two implementations of the `foreach` macro. The first demonstrates the use of this system is its most low-level form, and the second is an abbreviated version using the `syntax` macro.

#### 4.6.4.1 Underlying Implementation

The following code is an implementation of a `foreach` macro that exposes the underlying implementation to the programmer:

A macro is implemented by implementing `SyntaxExpander` by providing the two methods `getClauseNames` and `expand`. It is left unspecified, but it is assumed that the `syntaxSwitch` statement fills in the details of the `clauseNames` structure.

The expand method takes as its only argument a fragment of a skeleton syntax tree. Within this method the user is expected to use the `syntaxSwitch` statement to correctly match the macro form. A pattern for this statement is enclosed within a `{...}
... } block and may contain an arbitrary number of terminals and non-terminals.

Non-terminals follow a ? symbol and are specified by a name, and a type separated by a colon. If the name is omitted the system uses a default. It is possible to match an arbitrary SST by use of the * symbol.

```java
public class foreachSyntaxExpander implements SyntaxExpander {
    private static String[] clauseNames = {};
    public String[] getClauseNames() { return clauseNames; }
    public Expansion expand(Fragment fragments) throws SyntaxMatchFailure {
        syntaxSwitch syntaxSwitch syntaxSwitch syntaxSwitch (fragments) {
            case #{ foreach (?:type ?elt:name in ?:expression) ?:statement }:
                return #{
                    Iterator i = ?expression.iterator();
                    while (i.hasNext()) {
                        ?elt = (?type) i.next();
                        ?statement
                    }
                };
        }
    }
}
```

**Code Example 4.23: JSE Iteration Definition**

The #{ ... } block is overloaded to provide the syntax for the quasi-quotiation, and ? is used to mean unquote. These forms can be heavily nested if required.

It is unclear where the system obtains the name of this macro, it could either be from the class name, or more likely it is from the first terminal in the first case of the `syntaxSwitch`.

### 4.6.4.2 The syntax Macro

The syntax macro allows the programmer to dispense with housekeeping tasks and provides a more concise understandable definition. Code Example 4.24 is the `foreach` macro expressed in this improved form.

```java
public syntax foreach {
    case #{ foreach (?:type ?elt:name in ?:expression) ?:statement }:
        return #{
            Iterator i = ?expression.iterator();
            while (i.hasNext()) {
                ?elt = (?type) i.next();
                ?statement
            }
        };
}
```

**Code Example 4.24: JSE Improved Iteration Definition**
Notice here, that this is just the inner part of the definition from the previous subsection, but we are no longer exposed to any implementation details. Even the class definition is now hidden from us.

### 4.6.5 Applicability to Benchmarks

Unlike many other systems, the use of skeleton syntax trees allows JSE to provide access to the code within its clauses at a pretty basic level. To this end it would be possible to define macros that consist of little more than a wrapper of a SST structure representing whatever the programmer wished — programmers are free to perform extra parsing on this representation. As a result there are few limitations to what would be possible within a macro, although this approach should always be viewed as a workaround.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 assert</td>
<td>Simple definition.</td>
</tr>
<tr>
<td>2 foreach</td>
<td>Simple definition, but not possible to manually enforce the expression is indeed iterable.</td>
</tr>
<tr>
<td>3 printf</td>
<td>Not possible. It should be feasible to create a user-defined constraint to allow the use of literal strings, but JSE has no capacity to check static types.</td>
</tr>
<tr>
<td>4 SQL</td>
<td>JSE may be capable of providing SQL exactly, but much work would be required to match SQL expressions — the SST would have to be manually parsed.</td>
</tr>
<tr>
<td>5 Generators</td>
<td>Possible to define generators, but suspend statements would have to be manually detected.</td>
</tr>
<tr>
<td>6 Haskell</td>
<td>Not possible without full manual parsing of the subset.</td>
</tr>
</tbody>
</table>
### 4.6.6 Extensibility Criteria Assessment

Table 4.8: JSE Extensibility Criteria Assessment

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.1 Syntax Creation</strong></td>
<td>Syntax creation is only available in a limited form, macros must always commence with a name, but may appear anywhere within a file.</td>
</tr>
<tr>
<td><strong>1.2 Syntax Interrogation</strong></td>
<td>Syntax interrogation is via skeleton syntax trees (simpler and more general than abstract syntax trees). Inner macro calls can be interrogated before they are expanded.</td>
</tr>
<tr>
<td><strong>1.3 Syntax Overloading</strong></td>
<td>Not provided.</td>
</tr>
<tr>
<td><strong>1.4 Static Type Interrogation</strong></td>
<td>Not provided.</td>
</tr>
</tbody>
</table>
| **1.5 Expressiveness**           | JSE is capable of providing small syntax additions only. Its limited form of syntax addition would require compromises to provide language embeddings.  
                                    | Both the shorthand for defining new syntax and the pattern-matching construct for ease of use are defined in the language itself.              |
| **2.1 Simplicity**               | JSE’s design focussed heavily on making things user friendly, and this has been partially achieved. Skeleton syntax trees remove the programmer’s need to understand an entire abstract syntax.  
<pre><code>                                | However, the syntax for the definition of new macros is still a little clunky, and this is due to the limited syntax creation abilities of JSE. |
</code></pre>
<p>| <strong>2.2 Brevity</strong>                  | The use of quasi-quote, unquote, and combinations of these provide ease of use. The defined extensions for syntax creation allow the programmer to be relatively free of housekeeping tasks. |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2.3 Robustness</strong></td>
<td>Although unimplemented at the time of writing, JSE provides a description of support for automatic hygiene and referential transparency.</td>
</tr>
<tr>
<td><strong>3.1 Syntactic Correctness</strong></td>
<td>Due to the use of skeleton syntax trees, no guarantees are given as to the correctness of expansions.</td>
</tr>
<tr>
<td><strong>3.2 Error Detection</strong></td>
<td>Meta-programs can throw an exception to indicate a syntax error, but it is unclear if this system can be extended to support other errors.</td>
</tr>
<tr>
<td><strong>3.3 Error Reporting</strong></td>
<td>JSE provides good support for errors with facilities provided for editor support, and rather than the position in an expansion, the original source of an error is provided.</td>
</tr>
</tbody>
</table>
4.7 OpenJava

OpenJava [Tat99] is a Java extension that provides a compile-time meta-object system. It allows meta-classes to be associated with class definitions, and these drive the translation of the defined class. Typical translation is similar to automatic application of the visitor pattern [BH02§2].

OpenJava has limited applicability and its main emphasis is on semantic extensions rather than syntactic ones [BP01]. OpenJava’s relatively primitive ability to provide syntax extension allows an easy implementation without the necessity of handling complex parsing problems.

4.7.1 Power

OpenJava does not permit arbitrary syntax extension. In fact, its ability to add new syntax is very limited:

“Syntactic extension is limited to only a few certain places in class definitions (e.g., class adjectives) and their uses (e.g., after class names in callers).”
[BP01§8.5.4, pp. 41]

Consider the code in Code Example 4.25 to automate the generation of visitor methods. Following the instantiates keyword is the extension being used, and both the visits and on suffixes are defined within this extension. OpenJava also admits the addition of new modifiers. Suffixes allow a following list; it is unclear what elements this list may consist of, but examples show the use of types and literals at least [TCKI00].

```
public interface GUIVisitor instantiates VisitorPattern visits GUIElement {
    void visit() on Container, Panel, Label;
}
```

Code Example 4.25: Open Java Visitor Methods Usage

OpenJava has little syntactic freedom, but it was primarily designed for semantic extensions rather than syntactic ones [BP01].

OpenJava provides interrogation and creation of syntax through a set of abstract syntax classes and a compile-time reflection facility. OpenJava metaclasses must be explicitly
declared by use of implements, and no extensions are possible for primitive types or arrays.

### 4.7.2 Usability

OpenJava requires programmers to write a class that handles translation. Code Example 4.26 contains a code fragment for defining the VisitorPattern example from Code Example 4.25.

```
public class VisitorPattern extends OJClass {
    static void init() {
        registerDeclarationSuffix( "visits", .. );
        registerDeclarationSuffix( "accepts", .. );
        registerMethodSuffix( "on", .. );
    }

    void translate() throws MOPException {
        ... // explicit use of syntax construction methods in here
    }
}
```

**Code Example 4.26: Open Java Visitor Methods Definition**

This approach is straightforward to understand, but somewhat tedious. No facilities are provided to ease the programmer's burden; OpenJava has no quasi-quotations or name conflict protection.

“... OpenJava seems to ignore the technology, hygiene and referential transparency, that makes macros work.” [Bak01§6.4.5, pp. 88]

### 4.7.3 Error Handling

OpenJava does not provide syntactic safety:

“OpenJava lacks some features that make compile-time metaprograms robust:
Its macros can generate illegal pieces of syntax, because they allow metaprograms to convert arbitrary strings to syntax.” [BH02§2, pp. 3]

Translation methods can throw an exception, but it is not described how this exception is used by the compiler or whether or not this system can be explicitly used to provide useful error reporting from macros. Indeed, the handling of translation errors is not described at all.
4.7.4 Applicability to Benchmarks

OpenJava’s extension mechanisms are essentially class-based which makes its applicability to the benchmark test suite limited. As a result of this class-based approach, any attempts to provide simple statement-level extensions would require misuse of the underlying system via a tree traversal to discover statement extensions.

Each of these benchmarks would require the enclosing class to explicitly declare the use of these internally.

Table 4.9: OpenJava Applicability to Benchmark Test Suite

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 assert</td>
<td>Possible via checking to see if each statement was a call to a method called <code>assert</code>, and performing translation if so.</td>
</tr>
<tr>
<td>2 foreach</td>
<td>Not possible. OpenJava cannot provide the statement block.</td>
</tr>
<tr>
<td>3 printf</td>
<td>Possible with syntax changes as it is not possible to create arbitrary length argument list, but this could be handled as an OpenJava prefix specifier.</td>
</tr>
<tr>
<td>4 SQL</td>
<td>Not possible.</td>
</tr>
<tr>
<td>5 Generators</td>
<td>Possible once using same technique as the assert benchmark.</td>
</tr>
<tr>
<td>6 Haskell</td>
<td>Not possible.</td>
</tr>
</tbody>
</table>

4.7.5 Extensibility Criteria Assessment

Table 4.10: OpenJava Extensibility Criteria Assessment

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Syntax Creation</td>
<td>OpenJava provides very limited forms of additions to the syntax in the form of modifiers and suffixes, and these in turn are limited to specific places.</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>1.2 Syntax Interrogation</td>
<td>Provided via a compile-time reflection mechanism and a set of abstract syntax classes.</td>
</tr>
<tr>
<td>1.3 Syntax Overloading</td>
<td>Not provided.</td>
</tr>
<tr>
<td>1.4 Static Type Interrogation</td>
<td>Definitely provided for methods of the defined class, but unclear if it is provided for the surrounding context or for variables.</td>
</tr>
<tr>
<td>1.5 Expressiveness</td>
<td>OpenJava extensions are limited to those of class scope — although it is possible to misuse this mechanism to provide other smaller extensions.</td>
</tr>
<tr>
<td>2.1 Simplicity</td>
<td>Due to strict limitations on additions that can be made by the user there should be no parsing conflicts. Defining extensions is a relatively straightforward process, but users are exposed to the underlying system.</td>
</tr>
<tr>
<td>2.2 Brevity</td>
<td>No support is provided for making macros concise.</td>
</tr>
<tr>
<td>2.3 Robustness</td>
<td>No support is provided for robustness.</td>
</tr>
<tr>
<td>3.1 Syntactic Correctness</td>
<td>OpenJava allows creation of syntax from arbitrary strings and as such does not provide syntactic safety.</td>
</tr>
<tr>
<td>3.2 Error Detection</td>
<td>Translation methods may throw an exception but it is unclear how this is utilised and as to whether the user can explicitly use this mechanism for manually detecting errors in a meaningful way.</td>
</tr>
<tr>
<td>3.3 Error Reporting</td>
<td>Unclear if any support is provided for errors.</td>
</tr>
</tbody>
</table>
4.8 Maya

The Maya programming language [Bak01] is a Java extension that has demonstrated that it can be used for sizeable extensions. Its meta-program definitions attempt to mimic Java declarations closely.

4.8.1 Power

Maya allows both extension and reinterpretation of its syntax. The Maya programmer has the ability to write both new grammar productions and semantic actions.

Semantic actions are expanded in an outside-in fashion, and programmers must explicitly specify which arguments to a meta-program are to have their parsing delayed (see section 4.8.1.2).

Maya provides an abstract syntax for meta-programming, and provides for automatic extension through its ability to define new grammar productions.

4.8.1.1 Grammar Productions and Semantic Actions

Grammar productions are treated as generic functions, whereas semantic actions (called “Mayans”) on such productions are multimethods (i.e. methods that are polymorphic on more than one of their arguments).

Mayans can be dispatched on tokens, syntax trees, or the static type of an expression.

A Mayan can add any production to the grammar, but the resulting language must be recognisable by Maya’s LALR(1) parser (see section 6.5.3.1 for a description of LALR parsing). This is a sizeable restriction, and forces the programmer to be aware of what forms can be supported by LALR(1). As stated in section 3.2 it is desirable that the programmer need not know how parsers function to be able to use an extensible language.

4.8.1.2 Laziness

Maya employs laziness in both type checking and parsing. Lazy type checking allows Mayans to dispatch based on the static types of arguments and conversely to create variable bindings that can be used by other arguments. Both of these properties are useful in defining the foreach macro from section 3.4.1.2: the general form of
foreach should only work on arguments that implement Iterator (or for the Maya example also on Enumeration) but the body of the expansion relies on the definition of a loop variable.

Laziness requires the programmer to explicitly specify how much expansion is required for macro expansion to occur. Formal arguments of a macro may be annotated as lazy; these arguments will not be parsed until after the macro expands.

This system facilitates Maya providing static type-checking, but restricts the interrogation of lazy arguments. This in turn is prohibitive to close interaction between Mayans which limits Maya’s expressiveness.

“Maya infers the node types and layouts corresponding to interior productions based on the built-in Mayans. As a result, user-defined syntax may only appear at the root of the tree.” [Bak01§3.3.1, pp. 26–27]

### 4.8.1.3 Overloading Mayans

Maya supports macro overloading. This allows the definition of the same macro for wildly differing types, and, more importantly, specifically optimised code can be produced for each of these types.

This overloading facility provides a limited capacity for Maya to override built-in Java syntax.

### 4.8.2 Usability

The syntax for declaring Mayans mimics Java method call syntax closely. However, pattern matching forms are also provided which simplify many operations but add an initial barrier to understanding. These pattern matching forms provide a limited form of the functionality that could be provided by tightly nesting macros — this functionality is precluded by Maya’s use of outside-in expansion and lazy parsing.

Also, Maya provides a shorthand for the matching of static types which allows for concise powerful definitions (see section 4.8.4 for an example).

The programmer is required to understand when to use abstract and concrete syntax. These definitions require precise understanding of Maya’s implementation as:
“To define an abstract Mayan… one must understand Maya’s grammar and conflict-resolution techniques.” [Bak01§3.2, pp. 24]

In addition to manual construction of syntax via the abstract syntax classes, Maya provides both quasi-quote and unquote. Its quasi-quote syntax requires the user to specify the type being produced. Maya provides both hygiene and referential transparency within its quasi-quote construct.

Maya facilitates concise forms, but requires great programmer understanding, and:

“...It remains to be seen whether Maya is simple enough to be usable.”
[Bak01§7.3, pp. 91]

The use of Mayans is lexically scoped, the user must explicitly specify the scope of the Mayan with the use statement. This use statement can be at the top-level which provides file-scope.

### 4.8.3 Error Handling

Maya’s abstract syntax and quasi-quote construct provide a guarantee of syntactic correctness of expansions.

The system will automatically detect type errors when no specialisation can be selected to match the types of a macro invocation. In addition to this, the macro programmer can manually detect further errors and throw an exception to be handled by the parser. These exceptions can be produced by abstract syntax classes themselves in order to provide strong support for useful error messages.

### 4.8.4 Worked Examples

The following two examples demonstrate the implementation of assertions and iteration constructs.

#### 4.8.4.1 Assertions

Code Example 4.27 demonstrates a Mayan definition in its most low-level form. Each group of Mayans must appear within a class that implements the interface MetaProgram, and as a result must implement the method run that modifies the parser environment. Each of these Mayan definitions must be explicitly run on the environment, and the resultant environment must be returned. As we will see in the next
example, this is considered too repetitive for general use, and a Mayan is provided to remove this burden from the programmer [Bak01].

```java
package maya.util;
import maya.grammar.*;

use Syntax;

abstract Statement syntax (assert(Expression););

public class Assert implements MetaProgram {
  public Environment run(Environment env) {
    Statement syntax syntax syntax syntax A(assert(Expression e);) {
      return new Statement {
        if (!$e) throw new Error("Assertion failed");
      };
    }
    return new A().run(env);
  }
}
```

**Code Example 4.27: Maya Assertions**

This macro requires both an abstract Mayan declaration, and a concrete Mayan.

The concrete Mayan declaration utilises Maya’s quasi-quote to perform the translation. The quasi-quotation has a form similar to a Java object creation except the abstract syntax class does not have a list of parameters — if it did this syntax would be initially indistinguishable from an anonymous class declaration. Unquote in Maya is specified by the $ symbol.

### 4.8.4.2 Iteration

The implementation of iteration constructs provides a demonstration of Maya’s facilities for the creation of new abstract syntax, overloading of Mayans, lazy parsing, quasi-quotation, unquoting, and compile-time static-type checking.

The Maya implementation of a foreach structure mimics a function call and can be applied to many types as shown in Code Example 4.28(a).

The LinkedList version produces the default expansion which is a simple iterator loop, and the Vector version produces an optimised expansion which accesses the Vector as an array as shown in Code Example 4.28(b).
LISTED LIST list;
list.foreach(String st) {
  System.err.println(st + " = " + h.get(st));
}

maya.util.Vector v;
v.elements().foreach(String st) {
  System.err.println(st);
}

(a) Iteration Usage

for (Iterator enumVar$ = h.keys(); enumVar$.hasNext(); ) {
  String st = (String) enumVar$.next();
  System.err.println(st + " = " + h.get(st));
}

maya.util.Vector v;
{
  Vector v$ = v;
  int len$ = v$.size();
  Object[] arr$ = v$.getElementData();
  for (int i$ = 0; i$ < len$; i$++) {
    String st = (String) arr$[i$];
    System.err.println(st);
  }
}

(b) Iteration Expansion

Code Example 4.28: Maya Iteration Usage and Expansion

In order to demonstrate some important points, Code Example 4.29 contains a subsection of the Mayan definitions required to support iteration.

Firstly, we observe the decoupling of grammar productions from their associated semantic actions: on line 8 there is a generic function definition that defines a class of grammar productions that take a method name, a formal argument in parentheses, and a lazily parsed set of statements surrounded by braces. Following this is a number of Mayan definitions (on lines 10, 25, and 35).

These Mayan definitions all have the same basic form, that matches the abstract definition. The formal argument and block require no further explanation, but the method name part does. Each Mayan specifies that it will expand upon encountering an expression, followed by a ,, and the “method” name foreach. At first glance this does not seem to satisfy the requirements for a method name, however a Maya method name is a specialisation of an expression.

It should be noted here that the programmer is required to have a fairly detailed understanding of Maya’s abstract syntax in order to be able to write macros, this is the price paid for Maya’s expressiveness.
The first Mayan (lines 10–22) is designed to work on expressions that have a compile-time static type that implements the `Iterator` interface. For this to be possible the surrounding context needs to be available to the compiler when the macro is matched. This aspect of Maya’s syntax is on line 11, in the term `Expression:Iterator`.

Code Example 4.29: Partial Maya Iteration Definition
On line 13, this Mayan pulls type information from the formal parameter in order to be able to create a cast expression of the correct type.

On lines 15–21, is the first example use of Maya’s quasi-quotation; here it is used to create a new `Statement`. Within this quasi-quotation, there are multiple uses of the unquote operator $, that allow values from the surrounding macro to be used, all of these are type checked when the macro itself is compiled. Also, the variable `enumVar` is guaranteed not to conflict with the surrounding context on expansion due to hygiene.

On lines 25–29, the specialisation for collections is shown; it simply retrieves the collection’s iterator and uses the Mayan for iterators.

On lines 31–46, a partial definition of the array specialisation is shown. Maya is unable to check the static type of the array directly and the programmer is forced to include specific checks that the parameter is indeed an array (lines 40 and 41). On line 43, the type of the formal argument is checked against the type of the array, this was not possible to do for the iterator Mayan as Java (before 1.5) did not provide type information for collections.

The final part of this implementation (lines 49–53), demonstrate the housekeeping that the programmer is still required to perform in Maya. In fact, this is a Mayan written to reduce Maya’s housekeeping requirements.

### 4.8.5 Applicability to Benchmarks

The use of lazy parsing in Maya provides powerful facilities that lend themselves to producing concise code for many of the benchmark cases. However, this same system severely limits its expressiveness, and as a result, Maya implements part of the test suite concisely, but is incapable of implementing the remainder.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 assert</td>
<td>Simple definition.</td>
</tr>
<tr>
<td>2 foreach</td>
<td>Simple definition.</td>
</tr>
<tr>
<td>3 printf</td>
<td>Simple definition.</td>
</tr>
<tr>
<td>4 SQL</td>
<td>Not possible.</td>
</tr>
</tbody>
</table>

Table 4.11: Maya Applicability to Benchmark Test Suite
5 Generators

Not obviously possible. With change of `suspend` syntax to `return` Maya may be capable of providing this extension.

6 Haskell

Not possible.

### 4.8.6 Extensibility Criteria Assessment

Table 4.12: Maya Extensibility Criteria Assessment

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Syntax Creation</td>
<td>Arbitary placement of new syntax allowed, but outward-in evaluation does not allow for Mayans to depend on other Mayans.</td>
</tr>
<tr>
<td>1.2 Syntax Interrogation</td>
<td>Abstract syntax classes provided for syntax interrogation, although lazily parsed sections can not be examined. Pattern matching in formal arguments of Mayans simplifies some code.</td>
</tr>
<tr>
<td>1.3 Syntax Overloading</td>
<td>It is possible to override the default behaviour of languages forms.</td>
</tr>
<tr>
<td>1.4 Static Type Interrogation</td>
<td>Maya provide concise syntax for ensuring a parameter to a Mayan is of a specified type and provides functionality in abstract syntax for direct interrogation. Mayans make extensive use of specialisation.</td>
</tr>
<tr>
<td>1.5 Expressiveness</td>
<td>Maya is capable of providing small syntax additions only. A MultiJava implementation is provided, but this is possible because its syntax differs only slightly from Java.</td>
</tr>
<tr>
<td>2.1 Simplicity</td>
<td>Successful creation of Mayans relies on understanding of both the provided abstract syntax and the conflict resolution techniques of the parser. If a Mayan declaration causes a conflict, users need to understand LALR(1) grammars.</td>
</tr>
</tbody>
</table>
## 2.2 Brevity
Maya provides quasi-quotations and unquoting facilities, and provides a host of programmer shortcuts for Mayan definitions.

Whilst brevity is provided for Mayan definitions, each group of Mayans needs to be declared within a class that extends `MetaProgram`, and each exported Mayan must be explicitly added to the environment by the programmer.

Before use, each Mayan must be first imported and then its scope must be declared via the `use` statement.

## 2.3 Robustness
Maya provides automatic hygiene and referential transparency.

## 3.1 Syntactic Correctness
Mayans will always produce valid abstract syntax trees.

## 3.2 Error Detection
Syntactic errors in Mayan declarations are detected at compile-time, and the expansion is type checked. Mayans can explicitly check for errors and return useful information to the programmer.

## 3.3 Error Reporting
Explicitly detected errors provide useful error messages, whereas errors undetected by the macro do not.
5.1 Overview

The Genesis language design is inherently very simple, it consists only of a slight modification to the Java grammar, and specification of a new compilation strategy. This compilation strategy involves the definition of a flexible tokeniser coupled with a modification of the standard Java import mechanism.

Firstly, we explain the Genesis design rationale (section 5.2) following the review from the previous chapter.

We introduce the basic form of the macro definition (section 5.4) and explain the subtleties of macro definitions.

A high-level description of the tokeniser design is provided (section 5.5) with an emphasis on its flexibility.

The process of macro expansion is described (section 5.6) which entails a description of the modified import mechanism and a detailed explanation of the order of evaluation of macros.

Finally, the languages standard environment (i.e. facilities that should be available in any implementation) is described (section 5.6); this includes a description of the abstract syntax classes, standard exception classes, type-checking facilities, and a macro reflection mechanism.
5.2 Design Rationale

The Genesis design follows directly from the examination of previous languages in Chapter 4. The focus is on the often conflicting goals of power and simplicity.

5.2.1 Arbitrary Syntax Creation

The major failing of the languages reviewed in Chapter 4 was in arbitrary syntax creation. Each language had substantial restrictions on macro use:

- Lisp allows for arbitrary placement of macros, but not for new syntax creation, all macros are in prefix form.
- Template Haskell permits macros at the declaration and expression level only, requires explicit caller-side identification of macros, and macros must conform to its normal function call syntax.
- MS$^2$ limits macro placement to declaration, statements, and expressions and requires all macros to begin with a name.
- JSE has no restriction on placement within a source file but requires macros to commence with a name.
- OpenJava has strong restrictions on both the placement and syntax of any extensions.
- Maya allows for both arbitrary placement of macros and arbitrary syntax creation. Unfortunately, Maya has restrictions on the interoperability of macros: only lazy arguments can contain further macro definitions.

The primary design goal of this work was to provide the greatest flexibility in arbitrary syntax creation. Wherever possible when this conflicted with other goals, syntax creation flexibility was the winner.

This aim of flexibility resulted not only in the arbitrary macro definition design (see section 5.3) but also in the design of the tokeniser (see section 5.4).

5.2.2 Compile-time Interrogation

The majority of reviewed languages provided good support for the interrogation of syntax, but were greatly varied in their level of support for the overloading of syntax and compile-time type interrogation. Many of the most interesting extensions (e.g.
Maya’s implementation of \texttt{forall} — see 4.8.4.2) rely on both of these facilities. In particular, these kinds of specialisation allow for a host of user-defined optimisations.

Flexibility of macro definition already covers allowing overloading of macros, and it is considered essential that Genesis provide powerful facilities for interrogation of all aspects of compile-time information.

### 5.2.3 Base Language

A secondary design goal was not to create any artificial barriers to adoption. As previously discussed, programmer adoption of meta-programming has been limited. One of the factors behind this is the lack of facilities in most mainstream languages. Even Lisp has never seen much use outside a teaching or artificial intelligence setting. C++ does provide meta-programming, but in a form too inconvenient for most programmers to stomach.

It is for these reasons that a mainstream language (Java) was chosen as the vehicle for this research. Java itself is simpler than many other mainstream languages and this simplifies things from a language extension perspective — eg. the lack of persistent local stack variables. This choice makes design harder than say extending a functional language (eg. surrounding context is more important in Java), but still easier than trying to support extensibility in more complex languages.

### 5.2.4 Outward Language Simplicity

A secondary design goal was to provide the flexibility of arbitrary syntax creation in the most simple form possible — that is from the perspective of both the macro programmers and the macro users. Again, the major reason behind this was to allow for easy adoption.

To this end, it is desirable that macro definition syntax be both as simple as possible and as similar to Java method definition syntax as possible.

Also, programmers should not need to understand the difference between abstract and concrete syntax to be able to define or use macros. No artificial barriers should be created to differentiate between the concrete and abstract parts of a macro definition.
5.2.4.1 Programmer Support
Of the systems reviewed in Chapter 4 almost all provided a quasi-quotation facility and some form of hygiene. Also, most provided guarantees that all code translation would result in syntactically correct code (with the exception of JSE). These facilities allow the programmer to produce concise code and give some measure of confidence in its correctness. It is viewed as essential that Genesis also provides such facilities.

5.2.4.2 Parser Restrictions
Each of the reviewed languages that attempted to provide some form of arbitrary syntax creation (i.e. only JSE and Maya) are restricted by their choice of parser:

- JSE provides early structure detection by its use of skeletal syntax trees. These force the parse into a set of restricted shapes.
- Maya provides for a LALR(1) parser (see section 6.5.3.1) and requires the programmer to understand why parser conflicts may arise and how to repair them.

Parsers are not discussed in this chapter as Genesis was designed irrespective of parser issues. The focus was on providing the most flexible language as possible and to worry about how to parse it later. See Chapter 6 for a review of parser theory, and Chapter 7 for issues relating to the parsing of Genesis.

5.2.5 Inward Language Simplicity
A secondary design goal is to keep the language definition as minimal as possible. Lisp has shown the capabilities of a macro system by defining much of its standard language as macros. The Haskell definition contains a minimal core language and merely states how the more complicated constructs translate into this core language. Even C++ was initially described by a translation to C (i.e. C was the ‘simple’ core language). The benefits of this type of design are three-fold:

- a smaller core language design is easier to reason about, and there is less chance of unexpected interactions between language features;
- if the rest of the language can be defined within your core language, then it suggests the core language has a degree of expressive power; and
- smaller languages require less implementation burden.
Of these, the second point is the most important in regard to extensible languages. If the majority of the language cannot be expressed via macros, then the macro system is not powerful enough. If the language designers are running into restrictions of the language, then so will its users.

### 5.2.6 Error Reporting

The reviewed languages varied greatly in their support for the reporting of errors. Support varied from ignoring errors completely (eg. MS²) to trying to track errors back to their source (eg. JSE). Also some languages allow the user to provide their own error checking and to explicitly flag errors (eg. Maya).

Much error handling is typically left as an implementation issue (notable exceptions to this are Ada and Java). With an extensible language this is not possible as part of what is being defined is compile-time operation.

Error detection both by the compiler and explicitly by the user is necessary for many sophisticated extensions — eg. type-system modifications are not possible without being able to perform type-checking with the possibility of flagging new errors. It is necessary that Genesis provides strong support for explicit errors.

While poor error tracking can create adoption barriers for end-users, it is not essential for the creation of a successful macro system. It is highly desirable however. Genesis provides only limited error tracking support (see subsection 8.5.4).
5.3 Macro Definitions

Macro definitions attempt to take the place of arbitrary context-free grammar rules. A macro definition closely resembles a Java method definition and consists of a return type followed by a list of arguments.

Being able to represent context-free grammar rules in this fashion and the exact syntax of a macro is best demonstrated by an example. In Code Example 5.1 we present a small (incomplete) fragment of a set of BNF rules to define a Java statement and the equivalent concept as Genesis macros. Neither the full set of Java statements nor the macro implementations are shown.

```
statement ::= if(expr) statement else statement
           | while(expr) statement
           | ...
```

(a) Partial Statement Grammar

```
macro Statement (if, (, Expression expr, ),
                 Statement left, else, Statement right) { ... }
macro Statement (while, (, Expression expr, ),
                 Statement statement) { ... }
```

(b) Macro Definitions

Code Example 5.1: Grammar to Macro Translation

Macro definitions are easily distinguishable from Java methods by the keyword `macro`. This is mostly for clarity; it would be possible with the current form of Genesis macro definitions to do away with this entirely. Unlike Java methods, Genesis macro definitions are not named by an identifier, and their formal parameter lists admit varying forms and hence require a little further explanation.

5.3.1 Parameters

While still comma separated, formal macro parameters consist of three components: tokens (as they are defined in section 5.4), normal formal parameters, and literal strings. The latter are only provided for clarity, as it is possible to define a macro as shown in Code Example 5.2(a).

```
macro Expression (Expression left, , , Expression right) { ... }
```

(a) Raw Tokens
5.3.2 Precedence

A macro definition may also specify a precedence. If two macro definitions successfully match the same series of tokens, a precedence can be used to remove the ambiguity. Precedences may range from zero to one inclusive, and are specified (at least in definition) to an arbitrary degree of precision. Macros that do not specify a precedence are given the default precedence of 0.5.

In the example of the comma operator from the previous section it would be necessary to be able to discern whether commas specify use of said operator or actual parameter lists. For example, the preferred option would be to recognise $f(x, y)$ as a call to a function with two actual parameters $x$ and $y$, not a call to a function with one actual parameter $x, y$.

Code Example 5.3: Precedence Syntax

```
macro Expression (Expression left, ",", Expression right) precedence 0.4 { ... }
```

So we would like comma operator to receive a lower precedence, this is done as shown in Code Example 5.3. Note that the choice of 0.4 is arbitrary, but using real numbers for
the precedence means we have, theoretically, an infinite number of precedences in between any two precedences, so arbitrary choices are of little impact.

The use of explicit precedences is a marked departure from the use of relative precedences found in many other systems. Relative precedences are specified by relating the precedence of functions (often only operators) to one another. This approach is unsuitable for the Genesis language as a group of imported macros are not necessarily aware of any other previous macros against which relative precedences could be defined.

5.3.2.1 Precedence by Grammar Modification

For a given grammar, precedence can be simulated by transforming the original definitions into multiple other definitions. This is almost always done for expressions. Consider the simple expression grammar in Figure 5.1(a). This grammar is ambiguous since for expressions such as $1*2+3$ there are multiple interpretations.

```
expr ::= expr op expr | (expr) | number
op ::= + | - | * | /
```

(a) Without Precedences

```
expr ::= factor aop factor
factor ::= term mop term
term ::= (expr) | number
aop ::= + | -
mop ::= * | /
```

(b) With Explicit Precedences

**Figure 5.1: Expression Grammar**

Most grammars would introduce new terms and rewrite Figure 5.1(a) as shown in Figure 5.1(b) [ASU86]. Although, some grammars would not include `aop` or `mop`, and would instead expand them in the rest of the grammar. This approach is obviously possible with multiple macro definitions. For a large number of operator precedences however, this quickly becomes hard to follow. For languages such as Java and, to an even greater degree, C/C++ the number of precedences makes their expression grammars very difficult to follow.

One of the desirable properties of an extensible language is that extensions are simple for the programmer; to that end the precedence scheme for macros is more desirable than forcing them to understand grammar transformations.

- 135 -
5.3.3 **Associativity**

If we have two consecutive applications of the comma operator then there are two possible interpretations, e.g. does \(x, y, z\) mean \((x, y), z\) or \(x, (y, z)\)? For the comma operator it simply doesn’t matter, both interpretations will evaluate \(x\), then \(y\), then \(z\), and return the result of \(z\), but for many other operators/functions it does matter, e.g. assignment is right-associative, subtraction is left-associative, etc.

For the vast majority of macro definitions associativity will never be an issue, but nonetheless it is necessary to allow the programmer to choose between these interpretations. If we chose to specify right associativity for the comma operator we would modify our previous definition to that shown in Code Example 5.4.

```plaintext
macro Expression (Expression left, ",", Expression right)
precedence 0.4 rightassociative { ... }
```

**Code Example 5.4: Associativity Syntax**

Right-associativity is represented by a modifier switch with left-associativity being the default as the vast majority of operator definitions are left-associative. For example, the only binary operators in Java that are right-associative are the assignment operators [Sun02].

5.3.3.1 **Associativity by Grammar Modification**

There are similar transformations to those covered in section 5.3.2.1 to handle associativity, however for the same reasons (i.e. simplicity for the programmer) it was deemed that this approach was undesirable.

5.3.4 **Zero Argument Macros**

It is possible in Extended BNF to define optional components in grammars, for example, in Java methods have an optional series of modifiers, and a snippet of the grammar for defining methods is shown in Figure 5.2(a):

```plaintext
method ::= [modifiers] return_type name (( (( ... )) )) ...
modifiers ::= modifier modifiers | modifier
modifier ::= public public public public | private private private private | static static static static | ...
```

(a) EBNF Definition with Optional Components
Figure 5.2: Method Definition Grammar

Figure 5.2(b) shows the standard approach for representing this in normal BNF: remove the optional part and introduce an ε, which specifies that the rule can be satisfied by nothing.

This use of ε would correspond to a macro definition with no arguments. This is not supported at this time. This does not restrict the power of the resultant system, it merely forces the programmer into expanding out the possibilities. For example an alternative BNF definition not using ε is shown in Figure 5.2(c).

This becomes more of a problem the more optional components a particular definition has, for example a method declaration actually has an optional list of modifiers, an optional list of formal arguments, an optional throws clause, and in the case of abstract methods, an optional body. This situation would require sixteen separate definitions were it to be handled in this form.

Even if this situation was always necessary it is not a significant deficiency, but as we shall see in section 8.7.2, the system is powerful enough that we can define macros to alleviate this forced repetition.

5.3.5 Modifiers

No standard Java modifiers are supported for macros. All macros have effectively public style visibility. There is one new modifier, delayed, that modifies the normal order of evaluation of a macro (see section 5.3.8).

5.3.6 Exceptions

Macro definitions may have a throws clause just like normal Java functions. The language defines an abstract class ParserException that inherits from

• 137 •
Exception, and all exceptions in this throws clause must extend this class. For explanation of exceptions provided in the standard environment, see section 5.6.2.

### 5.3.7 Macro Body

The body of a macro is standard Java code. Provided as standard is a set of classes that provide a full Java abstract syntax. The body of a macro uses these standard classes to create Java programs. This is not a new approach (see [Bak01], [WC93], and [SPJ02]) and when using just this technique the code produced is relatively cumbersome.

“This style of code plagues meta-programming systems.” [WC93]

Many extensible systems try to provide cleaner support for the body of macros with the addition of both quasi-quotation and unquoting. This is always done by providing direct language support.

In Genesis, no direct language support has been provided for these forms, instead the language itself is expressive enough to define these as extensions (see section 8.7.1).

### 5.3.8 Evaluation Order

Macro expansion occurs either in an inside-out or outside-in fashion. All inside-out macros are expanded before the outside-in macros. To provide this choice to the user, macros can use the modifier `delayed`, which tags the macro for outside-in evaluation.

The reasons behind this approach and a more comprehensive description of macro evaluation order is given in section 5.5.2.

### 5.3.9 Placement and Scope

Macros can appear only in the same locations as normal Java methods. They are associated with the enclosing class in a similar fashion to methods. Other classes gain access to macro definitions through an extension of the normal import mechanism (see section 5.5.1). Unlike normal methods, however, all macros are accessible with qualification: i.e. macros do not need to be accessed via objects or in the case of static methods via a class name.

Whilst not currently prevented, it remains to be seen whether there is value in defining macros for inner classes or anonymous classes. It is not clear how these definitions would be accessed with the current system.
5.3.10 Grammar

In addition to the standard Java (Java 1.4 that is\(^6\)) syntax, only the new grammar rules shown in Figure 5.3 are introduced for macro definitions and add the macro declaration rule to the list of class member declarations.

```plaintext
class_member_declaration ::= macro_declaration | ...

macro_declaration ::= [delayed]
    macro return_type identifier (( [macro_argument_list] ))
    [precedence float_literal] [rightassociative]
    [throws_clause]
    method_body

macro_argument_list ::= macro_argument (,, ,, macro_argument)*

macro_argument ::= formal_argument | token | string_literal
```

Figure 5.3: Genesis Grammar

In this grammar the throws_clause is a standard Java throws clause. Four new keywords have been introduced, although, as demonstrated in the following section, Genesis does not have keywords in the traditional sense. However, in most macros the only keyword that would appear is the macro keyword.

New grammar rules are not required for macro calling as each new macro that is loaded (via the import mechanism, see section 5.5.1) will add itself directly to the grammar.

Indeed, it can be considered that the only modification to Java is of a new compiler, even the syntax for defining new macros is loaded as an extension to the compiler. In fact, the very substance of Java1.4 is loaded by default, but it would be possible to load an entirely different group of classes to begin with (see section 8.6.1 for an example of this).

\(^6\) At the time of implementation of Genesis, Java1.4 was chosen over the fledgling Java1.5. Some improvements to Genesis could be achieved by migration to Java1.5 — see section 10.2.4.
5.4 Tokenising

Tokenising is the approach of grouping the raw characters of a source file to allow for easier parsing. For a description of tokenising see subsection 6.3.3.

When breaking up any original source file it is desirable that we place as few restrictions as possible on user defined macros that will ultimately work with this tokenised version. Traditional tokenising methods tend to make early decisions on the exact nature of tokens, eg. literal strings, real versus integer numbers, keywords, etc.

An extensible programming language does not facilitate early decision making as little can be guaranteed about the purpose of any given lexical structure.

Real numbers serve as an appropriate example of the inherent difficulties: if the tokeniser matches `digits.digits` or even simply `.digits` as a real number, then macros that include either of these patterns as part of their definition would be impossible to define without treating that subsection of their definition as a “real” token.

Worse still, suppose we have two macros consecutively, the first one ending in “.”, the second beginning with `digits`. These use of two macros would be rendered impossible by the early matching of real numbers.

Of particular interest here is the case of keywords. It is undesirable to allow users to define their own keywords as keywords traditionally can only appear in exacting locations. For example, we cannot use a keyword as an identifier, if we allowed users to define their own keywords, then code that used those keywords would break upon attempting to incorporate these new macros.

The next few sections describe the development of the tokeniser with the emphasis on making it as flexible as possible.

5.4.1 Tokenising Approach Overview

Each source file is broken into a series of tokens, the only information removed is the position of white space (this itself can be considered a sizeable restriction of flexibility, see section 10.2.1 for a possible technique to improve this).
The basic idea is to strip all source files of comments, and tokenise the entire file treating white space and the change from alphanumeric characters to symbolic characters as separators as shown in Figure 5.4.

```plaintext
if (frogs>toads) ++x; else y -= 10;
```

(a) Source Code

<table>
<thead>
<tr>
<th>if ( frogs &gt; toads )</th>
<th>++ x ;</th>
<th>else</th>
<th>y = = 10 ;</th>
</tr>
</thead>
</table>

(b) Tokenised Version

Figure 5.4: Simple Tokenising

This simple tokenising approach needs some refining to handle symbols and some other special cases.

### 5.4.2 Special Cases

Both string and character literals are treated as exceptions to this basic strategy. These literals must be detected by the tokeniser and subsequently treated as a single token. Early handling of character literals is perhaps not strictly necessary (although spaces could make character literals unreadable), but string literals would suffer badly as a result of the removal of whitespace.

The only restricted symbols are those that are already permitted in Java identifiers, namely the dollar-sign and underscore. When encountered, these symbols are essentially considered to be “alphanumeric” characters.

### 5.4.3 Symbol Handling

It was clear in Figure 5.4 that when multiple symbols characters occurred in sequence that they were part of a single symbol (eg. ++ or --). Consider however the code fragment and its associated tokenised version in Figure 5.5.

```plaintext
x+=(y4++-zebras)*-400.3;
```

(a) Source Code

| x | += | ( y4 | ++ | - | zebras ) | * | - | 400 | .3 |
|---|----|------|--------|---|---|--------|

(b) Tokenised Version

Figure 5.5: Multi-character Symbol Grouping Tokenising

Multiple concurrent symbolic characters have been combined into symbolic tokens; treating these symbols as a single token is not what programmers expect.
5.4.3.1 Traditional Approach

Most systems disallow the creation of new operators and consequently sidestep this problem. Such languages must precisely specify what multiple occurrences of symbolic characters resolve to. For example, in the C/C++/Java program fragment `x+++y`, the tokeniser must consistently return either a post-increment and an addition; or an addition and a pre-increment.

For those systems that allow user-defined operators they typically limit the length of operators and treat some symbols as special. These special forms are typically the bracketing forms (e.g., `(`, `[`, and `{`) and separators (e.g., `,` and `;`). Further to these restrictions, such systems usually require explicit spaces between operators, or at least between user-defined operators.

These approaches are not appropriate to our extensible language design — unlike most languages, Genesis’ concept of a symbol is extended to include bracketing forms and separators/terminators. Any restriction at all to the symbols (and combinations of symbols) allowed in a macro definition could have unforeseen consequences at a later stage.

5.4.3.2 Explicit Spaces

In respect to construction of the tokeniser, the simplest solution would be to require the programmer to place explicit spaces between all symbols that are meant to be separate. Whilst an attractive solution for simplicity reasons, its use is too impractical, especially to those familiar with languages without such restrictions. For example, programmers are unlikely to remember to leave a space between consecutive parentheses.

5.4.3.3 Single-character Symbols

Another approach to support arbitrary symbol creation is to recognise only single character symbols, and to introduce a new grammar rule to provide for multi-character symbols at the parsing stage.

The code fragment from Figure 5.5(a) would simply tokenise as shown in Figure 5.6.

| x | + | = | ( | y | + | + | - | zebras | ) | * | - | 400 | | ; |

Figure 5.6: Single-character Symbol Output Tokenising
The new grammar rule would simply take two consecutive symbols and combine them into one symbol, i.e. \texttt{symbol ::= symbol symbol}.

However, this approach does not distinguish between multi-character symbols and multiple consecutive single character symbols, eg. “--” and “- -”. This is extremely undesirable and worse can lead to ambiguity as the user would never be able to specify that consecutive symbols are not to be treated as one.

This situation may be possible to rectify by allowing whitespace between symbols to pass through the tokeniser, where carefully chosen grammar rules could both remove the whitespace and produce all possibilities of symbols. This was ultimately rejected due to the complexity it would introduce and the parsing method used provides a simple solution for symbols.

**5.4.3.4 Symbol Combinations**

The most flexible approach should allow the user to provide for the removal of ambiguity explicitly by the insertion of whitespace between symbols whilst providing all possible combinations of multi-character symbols. Normal grammar rules would then be used to decide the correct interpretation of the multi-character symbols.

Returning many possible interpretations of character sequences is conceivable for other constructs (such as real numbers) but there is no compelling reason to do so. It is particularly desirable for symbols due to both the problems it solves with the recognition of multi-character symbols and the high frequency of consecutive symbols in typical source files.

The code fragment from Figure 5.5(a) would tokenise as shown in Figure 5.7.

The choice of grammar rules would be responsible for choosing the “correct” symbol combinations, namely: `+=`, `-=`, and `*=`. The result of this is similar to that of the technique using single-character symbols and explicit grammar rules, but without the possibility of introducing ambiguity.
Obviously the number of possible combinations of characters within a multi-character symbol grows exponentially with its length, but it seems that occurrences of long symbol sequences within source files is rare and almost all are large sequences of parentheses as a result of heavily nested expressions.

This approach is significantly different to traditional tokenising mechanisms that always uniquely categorise each sequence. This choice of highly flexible lexical analysis alone may have ultimately led to the necessity of a generalised parser, but as we shall see (section 6.6) a generalised parser has other benefits as well.

### 5.4.4 Tokeniser Algorithm

The transition diagram in Figure 5.8 summarises the function of the tokeniser.

The character groups are defined as follows:

- **Alphanum**: all alphabetic and numeric characters as well as the underscore and dollar sign symbols.
- **Symbol**: all symbolic characters except for the underscore and dollar sign.
- **Whitespace**: tab, space, newline, endline, etc.

Upon reaching end state 4, 6, 10, 13, 15, 17, or 19, the tokeniser performs the appropriate action and falls back to the start state (state 0).

States 1, 9, and 11 require further explanation. Each of these are essentially failure states for whichever rule is currently being followed, and the next state depends on which characters have been previously detected. In each case (except state 1) the preceding characters may be a mix of both alphanumeric characters and symbols, but always starting with a symbol. In each of these cases the tokeniser falls back to symbol handling (state 7) from the beginning of the sequence.
Figure 5.8: Tokeniser Transition Diagram
5.5 Macro Expansion

In order to allow the programmer access to previously defined macros, Genesis provides both an extension to the standard Java import mechanism and the specification of a precise evaluation order for the grammar.

5.5.1 Import Mechanism

Before usage, macros must first be imported via the standard import mechanism. Any macros associated with an imported class have file scope.

Each directly imported class (i.e. those import declarations not ending with ".*") will be examined to see if it has associated macro definitions. If so, each one of these macros will be added to the current grammar. This is the only method for bringing macros into scope.

Each macro definition is associated with a precedence and an associativity. It is possible to add two separate macros to the grammar that differ only by their precedence.

Imported macros have public visibility for the entire file being compiled.

5.5.2 Expansion Strategy

The basic idea is that macros have their bodies executed when their usage is detected, and the macros themselves explicitly build the parse tree. There are some extensions to this basic idea that allow for some of the more powerful macros to be created.

No guarantee is given to exactly which macros will be executed in the entire parsing process, only to that of the order of execution of macros that comprise the final parse tree. In this way we leave the exact parsing technique open to a variety of speculative techniques. The perfect parser (if there is such a thing for this language) would only execute those macros that make up the final parse tree in the expected order and no others, but it remains to be seen if this is possible. Indeed, the implementation provided in Chapter 8 performs many unused macro expansions.

5.5.2.1 Evaluation Order

As described in section 5.3.8, macros are divided into two groups: non-delayed and delayed. the expansion of the former group occurs first in a leftmost-innermost (inside-
out) fashion and the latter group’s expansion follows in a leftmost-outermost (outside-in) fashion.

Outside-in macros effectively have their execution delayed until the surrounding context has been fully determined, and as a result they have access to compile-time static types and the entire structure of the code contained within the current file.

The usage of these macros within code is still detected in an inside-out fashion, and their type information still guides the parse, it is merely their expansion that is delayed.

So if presented with series of nested constructs, the evaluation would expand macros from the inside out, and then any delayed macros from the outside in. The exact reasons behind this strategy are discussed in the following subsections.

### 5.5.2.2 Construction Versus Manipulation

In development it was noticed that macros tended to serve two basic purposes, either they performed manipulations on syntax trees, or they simply constructed these trees. This is generally the case with parsers for complex languages, particularly those that produce C, or some other high-level language, as an intermediary; a large proportion of grammar rules collect information to allow construction of more complex forms further down the parse.

Take for example, the `printf` function as described in section 3.4.1.3. It relies on the preceding context being available in order to be able to type-check its arguments, and it also relies on these arguments being provided (or more accurately: the expression trees that represent its arguments).

For the sake of argument assume that we have to explicitly generate facilities to specify the arbitrary length list of arguments (in reality we can simply reuse the standard classes for actual parameters), then the macro definitions (minus implementation) required are as shown in Code Example 5.5.

```c
macro Statement (printf, (, LiteralString s, ",", Arguments args, )) { ... }
macro Arguments (Expression arg) { ... }
macro Arguments (Arguments args, ",", Expression arg) { ... }
```

Code Example 5.5: `printf` Macro Prototypes

The class `Arguments` and the two macros that produce objects of it are used to build up a list of expressions — i.e. constructing a parse tree. The `printf` macro takes a
number of arguments and performs a complex code-generation procedure — i.e. manipulating a parse tree.

As we saw in Chapter 4, most macro systems provide for manipulation, but either only allow the use of pre-built parse tree elements, or provide an entirely different mechanism for introducing construction.

5.5.2.3 Outermost Versus Innermost Evaluation

Both construction and manipulation have different requirements for the preferred evaluation order.

Take the `printf` example again: if all macros are evaluated outermost first, then the macros to construct this expression list will not have been executed, we will know that there is a compilation path that results in an `Arguments` class being created, but we will have not done it yet. This clearly is not what we intended.

If we evaluated all macros innermost first, then a simple `printf` usage would function as we expect. However, consider a use of the `forall` function (as defined in section 3.4.1.3) in Code Example 5.6(a).

```java
forall (String s) in list { printf("%s", s); }
```

(a) `forall` and `printf`

```java
String[] array = ...;
forall s in array { printf("%s", s); }
```

(b) Specialised `forall` and `printf`

Code Example 5.6: Nested Macro Use

With innermost evaluation we do not know the type of `s` when we expand `printf`. This may not be immediately apparent, because there is obviously a declaration for a variable `s` — but the `forall` has not yet been expanded (indeed, it may not even have been detected), and hence this declaration has not be translated to Java code, and as a result is not examinable by the type system.\(^7\)

Indeed, if we had defined a very specialised version of `forall` for arrays that did not require the user to specify the type of the variable (as arrays maintain their typing

\(^7\) It may be possible to create a system in which macros can be interrogated about the type of their arguments even when they have not been expanded, but this idea is beyond the scope of this work.
information), then the problem is more obvious as is shown in Code Example 5.6(b). The variable s has no declaration available at this time, it is instead created as part of the macro.

### 5.5.2.4 Non-destructive Restriction

As previously discussed, the expansion strategy concerns itself only with the production of a final parse tree. Macros that contribute to this parse tree are guaranteed to be expanded in the order specified.

Apart from the macros that comprise the final parse tree, an implementation of the Genesis language is free to speculatively apply any macro to a matching segment of the input even if the resultant application is finally discarded.

To provide for this, macros should not destructively update their arguments as they may be used in future expansions.

This is not as restrictive as it may seem, as it is typical of meta-programming systems to produce code that doesn’t actually modify its arguments. Usage of quasi-quotation is true to this idea; it builds new programs from old components, it doesn’t modify any arguments it is given.

### 5.5.2.5 Standard Usage

All of the standard macros that create the Java abstract syntax use non-delayed innermost evaluation. Generally, this is what the macro programmer intuitively expects; if programming a macro that operates on expressions, the structure of the expression is expected to be available. Allowing most macros to use innermost expansions allows the creation of sophisticated constructs (see sections 8.6, 9.2, and in particular 9.2.6).

Early innermost evaluation is the default, as it is the most common usage. A macro can be tagged as delayed in order to allow it access to the surrounding context. In the printf example, the two macros that produce the argument list would not need to be delayed, only the printf macro would be, as it requires type-checking of its arguments. If all that was required was an unchecked C-style version, the delayed modifier could be removed.
Standard library extensions to Genesis simplify both the creation of early macros and delayed macros. In particular, we introduce an extension to match compile-time types in section 8.7.2.4, that means that most explicit uses of the `delayed` keyword disappear.

### 5.5.3 Macro Matching

Macros are matched based on the run-time types of parse tree fragments. Unlike Java which has strict rules for eliminating ambiguity, Genesis allows a single macro to match without raising an error even if two macros would be similarly appropriate. For example, consider the analogous Java code fragment in Code Example 5.7.

```java
int f(Object x, String y) ...
int f(String x, Object y) ...
...
String s, t;
f(s, t);
```

**Code Example 5.7: Ambiguous Java Declarations**

This example causes a compile-error due toore the ambiguity. The compiler is unable to decide which method is the most appropriate. Genesis resolves such conflicts, when they occur in macro matching, in favour of the most recently imported/defined macro, even if the macros were declared in the same file. This would appear to be appropriate as it allows macros to override each other. It may prove to be necessary to provide a warning to guard against unexpected macro overridings.

Genesis allows macros with the same arguments to co-exist within the same grammar. A variety of techniques are used to resolve ambiguities that arise from such multiple similar definitions (see subsections 5.3.2, 5.5.4, and 5.5.6).

### 5.5.4 Precedence

Macro precedence is used to resolve conflicts of ambiguity where appropriate. Precedence is only applicable for macros that have exactly the same return type. When resolving conflicts, the macro with the highest precedence contributes to the final parse tree. If a macro throws a non-serious exception (see section 5.5.6 for a description of the types of exceptions that can be thrown) then a lower precedence macro (or set of macros) can take its place.
Precedence conflicts are handled differently depending on whether the matched sequence implies an ordering conflict or a specialisation: i.e. if the rules preceding the conflict are the same or different.

5.5.4.1 Same Sub-rules

The following simple arithmetic example demonstrates the case of all sub-rules leading up to the precedence conflict being identical. Here we demonstrate two possible parses of the expression $3 + 4 \times 6$:

![Figure 5.9: Same Sub-rule Precedence](image)

This example is the classic case of basic arithmetic precedence. The left diagram is what is usually expected, the multiplication should take precedence. In this example the conflict is detected on the left when we are matching the addition part, and on the right when we are matching the multiplication (i.e. whichever rule is at the root of the tree). So at the point of detection, it is necessary to decide precedence in favour of the lower precedence rule, as the higher precedence rule should be matched earliest.

In summary, for matching sub-rules, we break precedence conflicts in favour of the lowest precedence rule.

5.5.4.2 Different Sub-rules

The following matrix specialisation example demonstrates the case of different sub-rules leading up to a precedence conflict. In Figure 5.10 we examine two possible parses of the expression $a + b \times c$. 

• 151 •
This is a standard example of how macros can allow users to create their own
optimisations. The basic idea is that we catch repeated matrix operations and perform
them in a more efficient fashion. In this case the \texttt{MatrixAddMul} matches on a matrix
addition followed by a multiplication and would be given a higher precedence than the
standard infix operators. The sub-rules that make up the tree structure are clearly
different in each case, and the precedence should be resolved with the macro that is at
the root of each parse tree with the \textit{higher} precedence.

### 5.5.5 Associativity

The following simple arithmetic example demonstrates an associativity conflict for the
simple expression $3 + 4 + 6$:

The Genesis parser will resolve such conflicts in favour of the left associative version
(i.e. the left-hand diagram), unless the macro has explicitly specified it is right
associative.
Associativity in the Genesis definition applies only to binary definitions as these are common in most programming languages. Whilst it is possible to extend the concept of associativity to more complicated definitions, this necessity is rare. An example of such a form is the `?:` conditional operator. For the time being, such occurrences must be handled explicitly by the user.

### 5.5.6 Exceptions

As described in section 5.3.6, macros can throw exceptions to signal various kinds of errors.

The effect on the parse tree of a macro throwing an exception depends on the type of exception being thrown. Exceptions of the type `Warning` or `Error` will only be visible to the end-user if the macro would have been part of the final parse tree. A `QuietParserException` signals that this macro failed in such a fundamental fashion that it should never be considered to be a part of a parse-tree.

Typically, delayed macros are more likely to raise warnings or errors, and non-delayed macros will make extensive use of quiet exceptions. A typical use of quiet exceptions is in classifying tokens in the earliest stages of parsing.

As we will see in section 5.6.2, one of the most basic kind of error that can be raised is `ConditionsNotMet`. If this error is thrown, the parser will not inform the user and will merely attempt to find another macro that will match the subsequence currently being examined.

Delayed macros cause a few more complications as it can not be determined in the early stages of the parse whether or not they will cause exceptions. Any other possibilities that successfully match the same input as a delayed macro must be maintained in case the delayed macro fails.

### 5.5.7 Restrictions

It is unclear how to parse mutually recursive macros. Neither can be fully parsed without previously parsing the definition of the other. For this reason alone, a macro cannot be used within the same file it is declared; standard methods are unaffected by this restriction.
For general simplicity macros can only be used outside the file in which they are declared. While this is perhaps overly restrictive it creates less problems (and less severe problems) than it removes.
CHAPTER 5: GENESIS LANGUAGE DEFINITION

5.6 Standard Environment

Just as the Java language definition must include descriptions of string classes (eg. `String`, `StringBuffer`, etc.) and exception classes (eg. `Throwable`, `Exception`, etc.) as they are used within the language itself for such things as literal strings, string concatenation, and run-time errors, the Genesis language definition must include descriptions of a number of similarly required facilities:

- the abstract syntax that macros both use as arguments and must produce as a result of application;
- the standard exceptions classes that can be thrown by macros;
- the standard facilities for compile-time type checking; and
- a macro reflection mechanism.

5.6.1 Abstract Syntax Classes

Whilst the abstract syntax classes are too numerous to examine in full detail (see Appendix A), the form of the standard abstract syntax still requires a little explanation.

It is possible to provide an abstract syntax that admits incorrect program — it is normally the job of the parser of the concrete syntax to ensure that this does occur. A great deal of care has been taken to design the abstract syntax in such a fashion that syntactically incorrect programs cannot be generated, while still providing as much simplicity as possible.

All Genesis abstract syntax classes must implement the `AbstractSyntax` interface. This is an empty interface that merely gives all classes a common starting point rather than relying on `Object`. It adds a certain level of trust to objects that are created by macros, and at least protects us from simple class of errors such as returning a Java `String` object where a `LiteralString` was intended.

The bulk of the hierarchy portion of the Genesis abstract syntax are interfaces, with classes only appearing as the abstract syntax becomes quite specific. The following diagram represents all classes and interfaces that create the hierarchy. The diagram has classes represented in bold type and all other types are interfaces.

- 155 -
Like AbstractSyntax, the majority of interfaces here are empty, and are merely to create this usable hierarchy.

These interfaces are used for abstract syntax classes to specify their membership into some set; for example, instance variable declarations, blocks, and inner classes can all be considered to be a ClassMemberDeclaration. This replaces the traditional necessity of using some sort of union style datastructure to provide this functionality.

Each of these classes and interfaces (with the exception of BlockStatement) is inherited from by other classes to provide specific functionality; these child classes are not shown here to facilitate the understanding of the underlying structure. For example, MethodDeclaration has children to represent macros, constructors, and abstract methods.

The hierarchy becomes a little more intertwined than this diagram suggest. For example, the abstract method class extends MethodDeclaration and implements InterfaceMemberDeclaration.

A full list of Genesis abstract syntax classes and an associated type hierarchy is available in Appendix A. We examine the use of the Typeable interface in section 5.6.3.
5.6.2 Exceptions

Macros have the ability to throw exceptions to signal compile-time failure. All exceptions thrown by a macro must inherit from a provide `ParserException` class.

![Exception Type Hierarchy](image)

In addition to this base abstract class, defined by default are three children of this class: `Warning`, `Error`, and `QuietParserException`. Macros that wish to fail without signalling the programmer would throw an exception that has `QuietParserException` as its base — this is particularly useful in the early stages of parsing when classifying tokens as symbols, identifiers, or the various categories of literals.

Two more standard exception classes are provided: `TypeMismatch` allows type-checking errors to be raised explicitly by the macro programmer and `ConditionsNotMet` allows a macro to define extra conditions on its applicability to a given set of inputs. The use of the latter allows a context-free grammar to be augmented with further conditions and can even be used to control parsing of context-sensitive grammars.

5.6.3 Compile-time Typing

Compile-time typing is provided via the `Typeable` interface shown in Figure 5.12. This interface defines a single method `type` which takes no arguments and returns a `Type` object if successful, or null if the type is undeterminable. Java reflection is heavily utilised for the discovery of types of members variables and methods.
Calculating the type of an object may involve recursive calls in order to type some of its members. For example, to correctly type a method invocation we must first type all of the expressions that make up its actual parameters, and these in turn may require further typing.

The Type class is a wrapper for the Java reflection class Class with some added functionality. Some examples of these extensions are:

- an extended forName method that correctly handles primitive types and checks all imported classes for matching names;
- construction of types from member names and associated information; and
- construction of corresponding array types from a given object.

### 5.6.4 Macro Reflection

On occasion it can be desirable to manually execute the body of a given macro — particularly when working without the quasi-quotation extension (see section 8.7.1). To this end a macro reflection facility is provided. The current implementation allows for the use of the normal Java reflection system (but at some cost to the user). In particular, the implementation details of macros would be overly exposed.

The macro reflection method attempts to mimic the interface style and operation of the getMethod method from Class. Unfortunately, the Java reflection class Class is final, so macro reflection must be provided as a utility. Additionally, macros have no name, so all matching is performed on the types of the arguments. Parameter lists are slightly more complicated due to the possibility of terminals, and a return type must be specified.

```java
public static Method getMacro(Class parent, Class returnType, ClassOrTerminal[] parameters)
    throws NoSuchMethodException, SecurityException ...

public static Method getMacro(Class returnType, ClassOrTerminal[] parameters)
    throws NoSuchMethodException, SecurityException ...
```

**Code Example 5.8: Macro Reflection Methods**

An additional macro reflection utility is provided that searches all imported classes for a suitable definition.
6.1 Overview

On the path to the implementation of the Genesis language we first review traditional parsing methods. Definitions of macros in the Genesis language have few limitations and the choice of parser is critical to the success of any implementation.

Readers well-versed in parsing theory may choose to defer reading of this parser review chapter (and, perhaps, the new parser design in chapter 7) and move straight to the implementation of the Genesis language (chapter 8).

As will become clear, while they are efficient, the methods used for the implementation of the majority of programming languages are not well suited to extensible programming languages. Compile-time modification of the underlying grammar is particularly difficult to incorporate within traditional implementation techniques. Also, such techniques require restrictions to be placed on the underlying grammar and these restrictions often impose a heavy burden of understanding of the parser creator — in this case, the end-user of the language.

This chapter begins with an introduction to grammars (section 6.2) and in particular, representations of context-free grammars are examined in detail. Grammar classifications are examined in order to introduce a basis for comparison of traditional parsing methods.

Following a brief general introduction to parsers in section 6.3, we look at often used top-down and bottom-up parsing methods (sections 6.4 and 6.5). Emphasis is placed on the grammar restrictions that are inherent in these techniques. Almost all real-world compilers use one of the techniques from these sections (or a minor variant).

In section 6.6 we look at general parsing methods that can handle a larger class of grammars (even ambiguous ones) than the previous techniques.

Finally, in section 6.7, we end with a summary of the relative merits of the parsers and examine each parser’s suitability to extensible languages.

6.1.1 Extensible Language Parsing

A parser for an extensible language suffers from more constraints than standard programming languages. The grammar of a language is typically fixed and as a result,
many optimisations can be applied to produce a successful parser. It is expected that an increase in parser flexibility leads to a corresponding decrease in efficiency.

A modifiable grammar requires consideration of two issues: the level of parser knowledge that is required of the programmer in order to modify the grammar, and the ability to modify the grammar during a parse.

### 6.1.1.1 Usability

The primary basis for judging a parser suitability for parsing an extensible language is whether the user must understand the parser. As programming languages are traditionally fixed entities without users modifying their syntax, it is typical to modify the grammar to suit the parser:

> “In practice, grammars are often first designed for naturalness and then adjusted by hand to conform to the requirements of an existing parsing method.”

[GJ90§3.6.4, pp. 75]

This is unacceptable for use in an extensible language. The macro programmer should be shielded from such awkward manipulations.

Typically, parser efficiency is premium amongst design issues and effort is spent in ensuring this above all other concerns. Indeed, efficiency concerns need only be addressed once as the grammar is unchanging.

> “… making [an efficient] parser for an arbitrary given grammar is 10% hard work; the other 90% can be done by computer.” [GJ90§3.6.4, pp. 75]

We consider the sacrifice of efficiency essential in order to remove the burden of hard work from the macro programmer. Ideally, we would not need to make this choice, but as a starting point for an extensible language we should not unduly restrict our parser.

### 6.1.1.2 Mid-parse Grammar Modification

There is another major issue that needs addressing to suit Genesis’ parsing requirements (and the same is also true of other similar extensible systems).

It is clear that an extensible parser must be capable of handling an augmented grammar and then use this grammar to parse a given input. In addition to this, extensible languages need a mechanism for specifying which modifications are to be made to the
grammar. This is typically done with a mechanism similar to that used to import functions or classes from libraries.

In Genesis, this mid-parse grammar modification the grammar occurs at the level of import statements contained in the file. The parsing of symbols beyond the import statement must be performed with the amended grammar. Maya has a special use keyword that modifies the grammar within the current scope.

Correct handling of such mid-parse grammar modifications is essential to the correct function of extensible languages.

As will be shown, the modification of a grammar midway through a parse places proves to be a harder requirement to meet than reducing the burden on the programmer.
6.2 Grammars

Grammars are common-place in the field of language design and are used so often that little thought is given to their structure. In this section, we first define the basic structure of grammars and introduce some varying classifications of grammars before looking at grammars traditionally used in language description.

6.2.1 Grammar Structure

A grammar (or generative grammar) is a set of rules for string transformation. Each rule consists of a left-hand side set of symbols and a right-hand side set of symbols. The general form of a grammar rule is shown in Figure 6.1; a set of these rules forms a grammar.

\[ \text{symbol}_1 + \ldots + \text{symbol}_n \rightarrow \text{symbol}_{n+1} + \ldots \text{symbol}_m \]

Figure 6.1: Generalised Grammar Rule

To generate a string in the grammar a single start symbol is chosen and rules are applied (in any order as many times as necessary) to rewrite the string. The language of a grammar is the set of all strings that can be generated in such a manner.

The reverse of this process can be applied to a given string to test whether or not it is a member of the grammar’s language.

6.2.2 Chomsky Hierarchy

The Chomsky hierarchy [GJ90] classifies grammars in order of decreasing order of power (and increasing ease of use), the four types of grammar differ in the type of rewriting rule that they allow as shown in Table 6.1 (summarised from [GJ90]).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 0: Unrestricted</td>
<td>Any number of symbols can be transformed into any other number of symbols — there are no restrictions on the form the grammar rules can take. Unrestricted grammars are extremely powerful, but too difficult to use for most purposes.</td>
</tr>
<tr>
<td>Type 1: Context-sensitive (transformational grammar)</td>
<td>The length of the left-hand side must be less than or equal to the length of the right-hand side.</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Type 2: Context-free (phrase structure grammar)</td>
<td>All left-hand sides must contain a single non-terminal symbol. The right-hand side is still free to contain as many symbols as necessary.</td>
</tr>
<tr>
<td>Type 3: Regular (right linear grammar)</td>
<td>All left-hand sides must contain a single non-terminal symbol. The right-hand side may only contain either one non-terminal and one terminal symbol; or a single terminal symbol. Regular grammars are not powerful enough to conveniently describe most languages, but sometimes are used for subsets of languages due to the ease of construction of fast parsers.</td>
</tr>
<tr>
<td>Type 4: Finite-choice</td>
<td>No non-terminals are allowed on the right-hand side. Finite-choice grammars are not in the original Chomsky hierarchy but are included for completeness due to their frequent use as an end-case.</td>
</tr>
</tbody>
</table>

### 6.2.3 Context-free Grammars

The context-free grammars (Type 2 Chomsky) are generally considered to be the best trade-off between expressiveness and easy parser construction. These are the typical grammars for language specifications — all grammars seen in the work are context-free grammars (written in BNF or EBNF as discussed later in this subsection).

Most parsers are written to operate on context-free grammars. Indeed such grammars are preferred for a variety of reasons (summarised from [ASU86]):

- context-free grammars provide a relatively comprehensible, precise specification of the syntax of a language;
- certain classes of grammar lend themselves to automatic parser construction;
- a well designed grammar coupled with appropriate tools allows for a reliable translation into correct object code; and
- languages with precise context-free grammars are easier to extend than those with hand-written parsers.
Despite this, it is surprising that most programming languages aren’t fully describable by context-free grammars. The classic example is that of variable declarations: names generally have to be declared before they can be used, this is not describable with a context-free grammar, and most language systems require an extra type checking phase after initial parsing.

6.2.3.1 Backus-Naur Form

Backus-Naur Form\(^8\) (BNF) is a convenient, and relatively standard, notation for context-free grammars. Each BNF grammar contains a series of declarations (often called *productions*) that contain a single non-terminal on the left-hand side, and a series of options on the right-hand side.

For example, consider the grammar fragment in Figure 6.2. This grammar fragment contains a single declaration that specifies that a non-terminal A can be derived from:

- a single non-terminal B;
- a non-terminal C, followed by a non-terminal D; or
- a non-terminal E, followed by the terminal `with`, followed by the non-terminal F.

The definitions for the non-terminals B, C, D, E, and F are not shown.

\[
A ::= B | C \ D | E with F
\]

Figure 6.2: BNF Context-free Grammar

6.2.3.2 Extended BNF

Extended BNF (EBNF) has a variety of styles, and is itself derived from regular expressions or extensions made to BNF by Niklaus Wirth in [Wir71]. Various bracketing styles and operators are introduced to allow such concepts as optional elements, choices within choices, and list of elements. The following are the most common styles for such concepts:

- Optional elements: e.g. depending on style \([a]\), or \(a?\) means zero or one occurrences of the symbol a.
- Zero-or-more: e.g. \(a^*\) means zero or more occurrences of the symbol a.

\(^8\) Originally, and occasionally still, called Backus Normal Form.
• One-or-more: e.g. $a+$ means one or more occurrences of the symbol $a$.
• Groupings are introduced by parentheses, and can be combined with the above operators as well as the choice operator from standard BNF:
  e.g. $(a | b)^+$ means any sequence made of the symbols $a$ and $b$, except the empty sequence.

All EBNF forms can be decomposed back into BNF form, it is merely a convenient notational shorthand.

6.2.4 Grammar Properties

Apart from classification within the Chomsky hierarchy, grammars have a number of other properties that are useful to describe, such as whether they are left- or right-recursive; or ambiguous.

6.2.4.1 Ambiguous Grammars

Consider the grammar in Figure 6.3.

```
expr ::= expr ++ ++ expr | expr ** ** expr | (( ((    expr )) )) | number
```

Figure 6.3: Ambiguous Simple Expression Grammar

This grammar is ambiguous because, for example, with the input $3+4+6$, there are two possible parses (as previously demonstrated in Figure 5.11). This raises the rather awkward situation where ambiguous grammars lead to parsers needing to discern the programmer’s intent. We would prefer our parsers to find just one suitable parse. It generally doesn’t make sense to allow ambiguity in programming languages — you want each individual program to have exactly one meaning.

For this reason, most grammars have ambiguity explicitly removed [ASU86]. That is, precedence rules are explicitly incorporated into the grammar. For example consider the reworking of the grammar from Figure 6.3 in Figure 6.4.

```
expr ::= expr * term | term
term ::= term * factor | factor
factor ::= number | ( expr )
```

Figure 6.4: Unambiguous Simple Expression Grammar

This grammar reworking allows for us to specify exactly which is the correct parse for each situation, but at some cost to the brevity and simplicity of the grammar. Such a reworking of an ambiguous grammar generally requires a detailed understanding or
which parse is preferable. The classic example is of a “dangling-else” that is common to
many programming languages [ASU86] and requires the final parse to match the short
`if` statement first (i.e. the outer `if` statement has no matching `else`).

### 6.2.4.2 Left- and Right-recursive Grammars

Almost all useful grammars have recursive elements — eg. even our simple expression
grammar of Figure 6.4 allows expressions to contain sub-expressions recursively. The
alternative is to expand the grammar for all possibilities; this is clearly infeasible for
grammars whose language is infinite.

A left-recursive production contains its left-hand side non-terminal in the left-most
position of one of its options, i.e. a rule similar to `A ::= A x`. A left-recursive
grammar contains at least one production that is left-recursive. The definition of right-
recursive grammar follows in a similar fashion.

As we will see in later sections, grammars that are left- or right-recursive can cause
problems for certain parsing techniques. Modifications are generally made to such
grammars to remove either the left- or right- recursion in a similar vein to those for
ambiguous grammars.

The ambiguous grammar of Figure 6.3 is both left- and right-recursive. The
unambiguous grammar of Figure 6.4 still contains two left-recursive productions. Figure
6.5 shows the result of reworking this grammar.

```
expr ::= term rest
rest ::= ++ ++ expr | ε
term ::= factor rest
rest ::= * * * term | ε
factor ::= number | (( (( expr )) ))
```

**Figure 6.5: Non Left-recursive Simple Expression Grammar**

Again, this translation removes the left-recursion but at the expense of the grammar.
This grammar is barely recognisable as accepting the same language as the original
ambiguous one.

An attempt at the removal of recursion can be performed in an automatic fashion
[ASU86], but there is no guarantee that such an attempt will succeed [ASU86].
6.3 Parsers

A parser is designed to recognise inputs that satisfy a particular grammar and reject those that don’t. Translation to another form is often performed as part of the recognition process.

Parsers operate on tokens rather than character streams. These tokens are produced by lexical analysers, or tokenisers. Normal compiler operation interleaves tokenising and the parsing as shown in Figure 6.6. Tokenisers are discussed further in subsection 6.3.3.

![Figure 6.6: High-level Compiler Model](source)

6.3.1 Derivation

For any grammar, there are typically a number of different orders in which the rules of the grammar can be applied in order to decide membership of a particular input. This order of application is called a derivation and it gives a precise description of the construction of a parse tree [ASU86]. In fact, the parse tree can be viewed as a graphical representation of a derivation that removes information about application order [ASU86].

Figure 6.7 contains the leftmost and rightmost derivation for the simple expression grammar of Figure 6.4, and the input $5*8+2$. Both achieve the same final parse, but merely apply the rules in different orders. The choice of leftmost or rightmost derivation does not affect the language recognisable by a parser, only the order in which they are applied.

---

9 Production compilers will often produce an intermediate representation from the parser which is then subject to optimisation and translation to the target platform.
6.3.2 Naming

A standard naming scheme has evolved to broadly classify different parsers. These describe parsers concisely by many parameters, such as order of scan, derivation order, and token lookahead requirements.

6.3.2.1 Derivation Categorisation

For the most general categorisation, parsers are categorised by the order that they scan a program while attempting to interpret it, and the derivation that they produce. Generally, parsers scan from left-to-right, so the two most used general classifications of parsers are:

- LL: scans from left-to-right, and produces the leftmost derivation.
- LR: scans from left-to-right, and produces the rightmost derivation.

6.3.2.2 Lookahead Categorisation

In addition, parsers are subcategorised by the number of symbols they need to examine in advance before making any decision (called lookahead). This information is added in brackets after the original classification, eg. LR(2) specifies a left-to-right, rightmost derivation parser that requires two token lookahead.
When only one token of lookahead is required the extra information is generally omitted, i.e. LL(1) is typically written simply as LL. A general categorisation of all LL parsers would be written as LL(k).

Non-ambiguous grammars exist that cannot be represented by LL parsers and thus would require further lookahead. The same applies for LR parsers. Simple grammars that cannot be recognised by LL or LR respectively are shown in Figure 6.8. In each case, trying to produce a derivation is impossible without further lookahead.

```
X ::= a b c | a b d
```

(a) Non-ambiguous Non-LL Grammar

```
X ::= a b c | Y c d
Y ::= a b
```

(b) Non-ambiguous Non-LR Grammar

Figure 6.8: Non-ambiguous Unparsable Grammars

### 6.3.3 Tokenising

Tokenisers convert the raw text input stream into a more manageable token stream. The tokeniser typically performs the following tasks:

- it matches keywords that appear in the input and passes through to the parser as special tokens;
- passes other words through to the parser as identifiers;
- checks the form of literals (typically characters, strings, and various number formats) and passes them to the parser as a single token;
- ensures that multicharacter symbols are passed through to the parser as a single token;
- rejects input that contains unexpected characters; and
- removes whitespace.

Different languages can have their own slightly different parsing strategy though. For example, Haskell has strict layout rules that are governed by the correct use of whitespace. The Genesis tokeniser performs far fewer early decisions than typical tokenisers (see section 5.4).

A tokeniser is typically implemented as a deterministic finite automaton (DFA) which, in turn, are usually pictured as a graph. DFAs are deterministic as there is only one path
for any given input, finite as there are a fixed number of states, and automatic as the
transition from state to state is simply determined by the input.

In Figure 6.9 a DFA to tokenise whitespace separated identifiers and numbers is shown.
Each state of the DFA is a node in the graph with transitions between states pictured as
edges with labels containing character sets. For each character of input the transition to
the next state is deterministically defined.

![Figure 6.9: Deterministic Finite Automaton](image_url)

Using DFAs for tokenising is simple and efficient and requires only a single pass of the
input. Tokenisers are generally written to provide tokens only as the parser requests
them, so no buffering is required.

### 6.3.4 Parsing Methods

There are typically considered to be three types of parsers: top-down parsers, bottom-up
parsers, and general parsers [ASU86]. Top-down and bottom-up parsers are commonly
used in production compilers, but are more restrictive than general parsers.

In the following sections we look at these different parsing methods. The review is far
from exhaustive (for a fairly comprehensive review see [GJ90]), but highlights the most
common parsers from each category.
6.4 Top-down Parsing

LL(k) parsers are typically called top-down parsers as their operation can be viewed as an attempt to construct the left-most derivation by a pre-order traversal of the resultant parse tree [ASU86].

LL(k) parsers require a non-ambiguous grammar that is free from left-recursion.

In this section we look at the two major implementations of top-down parsers: the first is known as predictive parsing (section 6.4.1) and the second is a parse-table approach which is generally just referred to as LL-parsing (section 6.4.2).

The primary difference, besides the actual implementation, between these two approaches is efficiency (of both space and time); with a little effort it should be possible to construct parsers for the same grammars using both methods.

6.4.1 Predictive Parsing

Recursive descent parsing attempts to apply each grammar rule in turn until a conflict occurs at which point it backtracks until an alternative choice is available. This is implemented with a set of recursive procedures. In general, recursive descent parsing is inefficient and will always fail on left-recursive grammars. As a result, this class of parsers are rarely seen [ASU86].

Predictive parsing is a special case of recursive descent parsing that does not require backtracking. Such parsers are popular because their recursive nature makes them highly suited to implementation by hand (the majority of parsing techniques are not).

The recursive procedure calls that occur as parsing proceeds implicitly define the derivation and, hence, the parse tree [ASU86]. The actions to be performed on the parse tree are contained within these recursive procedures.

For a suitable grammar, construction of a predictive parser requires each BNF rule to be translated into a single function. This function calls other such functions for non-terminals and guarantees any terminals match the current token.

Code Example 6.1 demonstrates a predictive parser implementation for the grammar from Figure 6.5. It performs no actions as a result of the parse, it merely checks for correct syntax. The implementation relies on an overloaded function `match` that
CHAPTER 6: REVIEW OF PARSERS

This example demonstrates the simplicity of providing ad hoc predictive parser implementations.

6.4.1.1 Advantages
The major advantage of predictive parsers is the ease that they can be constructed manually. The resultant parser is also easy to comprehend which can make it easy to discover errors.

6.4.1.2 Disadvantages
Predictive parsers can only handle a non-ambiguous non left-recursive grammars. There are further transformations that may need to be applied to grammars such as those as the example in Figure 6.8(a). Future revisions to the grammar are often difficult to handle, especially when many grammar modifications were required to produce a LL(1) grammar originally. Such highly-modified grammars are difficult to understand by humans and can be unrecognisable from the original [ASU].

However, the major disadvantage of predictive parsers is efficiency. Procedure calls are expensive and parsing requires many such calls. These procedures also require more space than other equivalent techniques:

“The most important disadvantage of generating a recursive descent parser is the size of the parser. A recursive descent parser is usually larger than a table-driven one (including the table).” [GJ90§8.2.6, pp. 178]

Also, in each procedure we may make many failed attempts to match a given input token. No systematic approach is used to decide which production branch to chose given a particular input token.
6.4.2 Parse Table Approach

In order to overcome the efficiency problems of predictive parsers, top-down parsers typically use a non-recursive table-driven approach. Such parsers can be constructed for an equivalent class of grammars as predictive parsers but without the overhead of recursive procedure calls. This is achieved by maintaining an explicit stack.

A model of a table-driven parser consists of the input, a stack, a parse table, and the output as shown in Figure 6.10.

![Figure 6.10: Table-driven Predictive Parser Model](image)

The input consists of tokens and the stack contains the current sequence of recognised grammar symbols. The parser functions by using the current token and the current grammar symbol to guide its next action. The actions to be performed are stored in the parse table.

The stack is initialised with the start symbol and at each stage in parsing the parser has three possibilities [ASU86]:

- if both the input stream and the stack are empty the parse has successfully completed;
- if the top of the stack is a terminal and is equal to the current input token, then it is removed from the stack and the current token is advanced. When not equal, an error has occurred; or
- if the top of the stack is a non-terminal, both the top of the stack and the current input token are used as lookups into the parse table. The parse table entry specifies either an error has occurred, or a sequence of grammar symbols
(corresponding the right-hand side of the grammar production) with which to replace the top of the stack.

A table-driven parser for the simple expression grammar from Figure 6.5 is shown in Code Example 6.2.

```java
boolean boolean boolean boolean parse(Queue input, Symbol start, Table table) {
    Stack stack;
    input.add("$");       // add a special terminal symbol to the end of input
    stack.push(start);    // push start symbol onto the start initially
    while (!stack.isEmpty()) {
        Symbol top = stack.pop();
        Token current = input.front();
        if (top.isTerminal()) {
            if (current.equals(top)) return false; // error
            input.dropFront();
        } else {
            Stack topush = table.lookup(top, current);
            if (topush == null) return false; // error
            stack.push(topush);
        }
    }
    return input.isEmpty();
}

boolean expr(Queue input) {
    Table table;          // table is constructed as shown in Table 6.2
    return parse(input, Symbol.EXPR, table);
}
```

**Code Example 6.2: Table-driven Simple Expression Parser**

Code Example 6.2 is a restatement of the previous description of a table-driven parser. The input stream is represented as a queue, the state of the parser is represented as a stack, and the exact implementation of the parse table is not specified. Parse table lookup errors are returned as null references.

One minor detail simplifies the implementation a small degree: a special terminal symbol (shown as $) is added to the input to allow easy handling of empty input cases within the parse table directly. Having completely parsed the input with non-terminals remaining on the stack is not necessarily an error, it merely represents the situation where we have successfully parsed and have recursive calls to unwind.

The parse table for the simple expression grammar of Figure 6.5 is shown in Table 6.2. A LL(1) parse table is a two dimensional array with terminal symbols along the column axis, and non-terminals down the row axis. Each state contains either:

- a series of non-terminal and terminal symbols;
the empty symbol, signifying recursive call unwinding; or

- a blank, signifying a parse error.

The size of a parse table is proportional to $O(\text{terminals} \times \text{non-terminals})$. For the simple expression grammar of Figure 6.5, the parse-table contains 30 entries.

**Table 6.2: Simple Expression LL Parse Table**

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>(   ) + * $</td>
</tr>
<tr>
<td>expr</td>
<td>term rest_e term rest_e</td>
</tr>
<tr>
<td>rest_e</td>
<td>$\varepsilon$ + term rest_e $\varepsilon$</td>
</tr>
<tr>
<td>term</td>
<td>factor rest_t factor rest_t</td>
</tr>
<tr>
<td>rest_t</td>
<td>$\varepsilon$ $\varepsilon$ * factor rest_t $\varepsilon$</td>
</tr>
<tr>
<td>factor</td>
<td>number ( expr )</td>
</tr>
</tbody>
</table>

Manual construction of parse tables is systematic but tedious and error-prone, automatic generation is generally preferred. Such automatic generation of parse tables are beyond the scope of this work (for an in-depth treatment, see [ASU86]).

If, after construction, the parse table contains duplicate entries, then the grammar cannot be parsed by a LL(1) parser. Or, put another way, a grammar that generates a table that contains no duplicate entries is a LL(1) grammar. Grammars that are not LL(1) may be able to be modified, or it might be necessary to introduce further lookahead, but the resultant parse table size is prohibitive:

“Strong-LL(k) parsers with $k > 1$ are seldom used in practice, because the parse tables are huge, and there are not many languages that are LL(k) for some $k > 1$, but not LL(1).” [GJ90§8.3, pp. 180]

### 6.4.2.1 Advantages

The major advantage of a parse table approach is that of efficiency. Maintaining an explicit stack is typically much more efficient (in both space and time) than handling many recursive calls on the program stack [ASU86].
6.4.2.2 Disadvantages

The table-driven approach to LL(1) parser construction suffers from the same grammar problems as predictive parsers outlined in 6.4.1.2. The class of acceptable grammars is identical [ASU86].

As demonstrated in Code Example 6.2 and Table 6.2 respectively, the resultant parser is understandable, but the parse tables that drive it are not.

The size of the parse table for parsers requiring lookahead greater than one is generally unacceptable as each successive increase of lookahead increases the size of the parse table by a factor proportional to the number of non-terminals.

6.4.3 Suitability to Extensible Languages

LL parsers are unsuitable for use with extensible languages as they require heavy modification of a grammar before use and disallow the modification of the grammar midway through the parse.

6.4.3.1 Usability

LL parsers are easily foiled by left-recursive grammar (as shown in Figure 6.8(a)) and such forms are often used to provide a convenient way to express many languages. The burden of translation of a grammar away from left-recursion is too much for a programmer and may not even be possible.

6.4.3.2 Mid-parse Grammar Modification

An LL parser may be deep within a recursive call when the grammar is modified and as such may be unable to incorporate the new grammar rules. In fact, top-down recognition is invalid if the entire grammar is unknown. There is no way to ascertain that a rule will be matched without the rules that generate it being known.
6.5 Bottom-up Parsing

LR(k) parsers are typically called bottom-up parsers as they effectively produce a derivation from right-to-left, i.e. from the base of the resultant parse tree to the top.

LR(k) parsers accept a larger class of grammars than top-down parsers, in particular, both left- and right-recursive grammars are permitted.

In this section we introduce the general concept of shift-reduce parsing (section 6.5.1), and then two implementations: operator-precedence parsing (section 6.5.2) and the most commonly used implementation referred to as LR parsing (section 6.5.3).

Three LR parsers are examined: LR (also known as canonical LR), SLR (simple LR), and LALR (look-ahead LR). The primary differences between these parsers is the size of the parse-table and the class of grammars they accept.

6.5.1 Shift-reduce Parsing

Shift-reduce parsing attempts to produce a right-most derivation traced out in reverse [ASU86]. Figure 6.11 shows the reductions taken by a shift-reduce parser for the simple expression grammar from Figure 6.4 and the input 5*8+2.

```
number * number + number
⇒ factor * number + number
⇒ term * number + number
⇒ term * factor + number
⇒ term + number
⇒ expr + number
⇒ expr + factor
⇒ expr + term
⇒ expr
```

Figure 6.11: Shift-reduce Parser Reductions

This is, in fact, identical to the right-most derivation shown in Figure 6.7(b) traced out in reverse.

Shift-reduce parsers are typically implemented by use of a stack and a parse-table in a similar fashion to the table-driven approach to top-down parsers shown in 6.4.2. Although a shift-reduce stack contains a mixture of terminals and non-terminals, unlike the stack in the table-driven top-down parser approach which contains only non-terminals. A shift-reduce parser has the generalised form shown in Figure 6.12.
For each step in the parse, there are four possible actions that can be taken [ASU86]:

- **shift**: the next input symbol is shifted onto the top of the stack;
- **reduce**: a right-hand side of a grammar rule has been matched on the top of the stack and a series of symbols from the start is replaced with the corresponding left-hand side;
- **accept**: the parser successfully completes; and
- **error**: an error has occurred.

The first two actions are most commonly applied and it is from these that shift-reduce parsers get their name. A shift is performed when the current token is expected and moves the focus onto the next input symbol. A reduce is performed when a rule is complete and ready to be replaced by a single corresponding non-terminal symbol.

A shift-reduce parser requires some form of control mechanism for it to be able to determine when to shift and when to reduce. Ad hoc implementations are possible, but typical grammars are too complicated for this to be possible and more automated techniques are used. These automated techniques involve construction of a table of information that is used to provide control information [ASU86].

### 6.5.2 Operator-precedence Parsing

Operator-precedence parsing is a form of shift-reduce parsing that requires that the grammar does not contain two adjacent non-terminals and no empty symbols. However, it can handle ambiguity in the grammar by careful representation within the operator-precedence table.
Operator precedence can be implemented either in a table-driven fashion or via precedence functions [ASU86]. The focus of this subsection is on the table-driven approach as it bears closer relation to the parsers in the surrounding subsections (for the alternative approach see [GJ90]). The general structure of a table-driven operator-precedence parser is shown in Figure 6.13.

The stack in an operator-precedence parser contains both operators and non-terminals. The operators are used to calculate precedences and the non-terminals are somewhat inconsequential [GJ90].

![Figure 6.13: Operator-precedence Parser Model](image-url)

A special terminal symbol is added to the top of the stack and the end of the input and at each stage in parsing the parser has three possibilities [ASU86]:

- if the top of the stack has precedence lower or equal to the current input token, then the token is pushed on the stack and the current token is advanced;
- if the top of the stack has precedence greater than the current input token, symbols are popped off the stack until the terminal on the top of the stack has precedence less than the most recently popped terminal; or
- if the precedence relation between the top of the stack and the current input token is undefined an error has occurred.

Code Example 6.3 illustrates an operator-precedence parser for the simple expression grammar of Figure 6.3.
```java
boolean parse(Queue input, Symbol final, Table table, Rules rules) {
    Stack stack;
    input.add("$");       // add a special terminal symbol to the end of input
    stack.push("$");      // push special terminal symbol onto stack
    while (/* more left on the stack than the final symbol */) {
        Symbol top = stack.topTerminal();
        Token current = input.front();
        if (table.lookup(top, current).isLessThanOrEqualTo()) {      // shift
            stack.push(current);
            input.dropFront();
        } else if (table.lookup(top, current).isGreaterThan()) {     // reduce
            Queue queue;
            do {
                queue.add(stack.pop());
                while (table.lookup(stack.top(), top).isGreaterThanOrEqualTo());
                stack.push(rules.reduce(queue));
            } else {
                return false;
            }
        }
    }
    return true;
}

boolean expr(Queue input) {
    Table table;          // table is constructed as shown in Table 6.3
    Rules rules;          // E=E+E, E=E*E, E=(E), E=number
    return parse(input, Symbol.EXPR, table, rules);
}
```

Code Example 6.3: Simple Expression Operator-precedence Parser

The size of a precedence table is proportional to $O(\text{terminals} \times \text{terminals})$. Table 6.3 contains the operator-precedence table for the simple expression grammar of Figure 6.3, it contains 36 entries. Each entry in the table represents the relative precedences between each operator. The differing set of precedence relations for addition and multiplication demonstrate the resolution of inherent ambiguity in the original grammar.

### 6.5.2.1 Advantages
Operator-precedence parsing is easy to implement by hand either using precedence functions or with a precedence table.

The original ambiguous expression grammar can be used directly along with other information specifying the precedences of each operator. No grammar mangling is required.

### 6.5.2.2 Disadvantages
Only a restricted set of languages can be handled by operator-precedence parsers as they disallow all grammars with adjacent non-terminals.
### 6.5.3 LR Parsing

LR parsing\(^{10}\) is a general concept of a shift-reduce parser driven by a state machine. LR parsers use their current state and the current input token to drive the parse. The general structure of a LR parser is shown in Figure 6.14.

![LR Parser Model](image)

In addition to the four possible actions stored in each entry in the parse-table (renamed as the *action* table), LR parsers contain extra information called *goto*. When a rule is reduced, the algorithm uses this goto field to jump to a parse state that represents the reduced non-terminal symbol.

---

\(^{10}\) Sometimes referred to as canonical LR to more adequately distinguish it from the family of LR parsers.

---

**Table 6.3: Simple Expression Operator-precedence Table**

<table>
<thead>
<tr>
<th>Terminal</th>
<th>Terminal</th>
<th>( )</th>
<th>+</th>
<th>*</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td></td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>(</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>)</td>
<td></td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>+</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>*</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>$</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
</tbody>
</table>
For each step in the parse, the same four action types as general shift-reduce parsers apply, but with changes to both shift and reduce [ASU86]:

- **shift**: the parser shifts the current input symbol and the next state onto the stack;
- **reduce**: the parser pops the number of symbols and states required by the right-hand side of the reducing grammar rule and looks up the goto for the state now on the top of the stack;
- **accept**: the parser completes successfully; and
- **error**: the parser has discovered an error and calls an error recovery routine.

The driver program is identical for all LR parsers, it is only the parse table construction method that changes from one LR parser to another [ASU86]. Code Example 6.4 demonstrates a LR parser implementation for the simple expression grammar of Figure 6.4. The methods in the following subsection (SLR and LALR) differ to LR parsing only by the contents of the parse table.

Construction of LR parse tables involves the construction of a state machine (or finite automaton). Typically, a non-deterministic finite automaton (NDFA) is constructed and then translated into a deterministic finite automatic (DFA) [ASU86]. For the simple expression grammar of Figure 6.4 the NDFA contains 77 states, and the corresponding DFA contains 23 states. Both are too complicated to reproduce here.

The size of an action table is proportional to \( O(\text{states} \times \text{terminals}) \) and the size of the goto table is proportional to \( O(\text{states} \times \text{non-terminals}) \). The size of the entire parse table is hence proportional to \( O(\text{states} \times (\text{terminals} + \text{non-terminals})) \). For the simple expression grammar of Figure 6.4, the parse-table contains 207 entries and is shown in Table 6.4.

The states are shown as the rows, and all the terminals (including the end-of-input symbol \( \$ \)) and non-terminals appear along the columns. The terminals symbols specify actions to be taken at each step in the parse, and non-terminal symbols specify the gotos to be taken after a reduction has occurred.
Table 6.4 is much larger than the parse table for the operator-precedence parser or the predictive parser. As in previous parse tables, blanks signify a parse error has occurred. States such as 1 and 4, and 17 and 18 are actually identical, even though their contents within the finite automaton are not. Perhaps states such as 5 and 7 can actually be combined and picked up as errors later. It should be clear that many improvements can be made to this parse table. LR parsers produce a state machine that is truly massive in size.

“LR(1) parsers [are] impractical in that the space required for their deterministic automata [is] prohibitive. A modest grammar might already require hundreds of thousands or even millions of states.” [GJ90§9.6, pp. 211]

For instance, the C programming language has over 10,000 LR states [Ast’05]. This highlights the need for some techniques to reduce the size of the parse tables.
### Table 6.4: Simple Expression LR Parse Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s7 s2</td>
<td>E 10</td>
</tr>
<tr>
<td>1</td>
<td>s9 s3</td>
<td>E 18</td>
</tr>
<tr>
<td>2</td>
<td>s9 s3</td>
<td>E 19</td>
</tr>
<tr>
<td>3</td>
<td>s7 s5</td>
<td>E 12</td>
</tr>
<tr>
<td>4</td>
<td>s9 s3</td>
<td>E 18</td>
</tr>
<tr>
<td>5</td>
<td>r4 r4 r4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r6 r6 r6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r4 r4 r4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r6 r6 r6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>s4 a</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>r2 s13 r2</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>r1 s13 r1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>s7 s5</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>r3 r3 r3</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>r2 s16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>s9 s3</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>r3 r3</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>s23 s20</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>s23 s20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>s9 s3</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td>r1 r1 s16</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>r5 r5 r5</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>r5 r5 r5</td>
<td></td>
</tr>
</tbody>
</table>

### 6.5.3.1 LALR Parsing

LALR (Lookahead LR) parsing is a LR variant that attempts to collapse similar automaton states thus reducing the size of the parse table. As a result of this collapsing, the algorithm is slightly less powerful than an LR parser for the same grammar [GJ90].
“The idea is now to collapse the automaton but to keep the look-ahead information (and collapse it too, but not discard it). The surprising fact is that this preserves almost all the original look-ahead power and still saves an enormous amount of memory.” [GJ90§9.6, pp. 211]

Some grammars that are LR are not LALR, but this is rarely the case and has few examples in the real world [GJ90]. Sometimes the improvement from LR to LALR is huge: LALR drops the number of states required for a C language parser from over 10,000 to around 350 [Ast+05].

Table 6.5 shows the improvement in parse table size from the LR parse table in Table 6.4. This parse-table has 108 entries (down from 207). Whilst this is a great improvement in size, it is still significantly larger than the tables required for precedence parsing or table-driven LL parsing.

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
<th>num</th>
<th>(</th>
<th>)</th>
<th>+</th>
<th>*</th>
<th>$</th>
<th>E</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s6</td>
<td>a</td>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>s7</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r4</td>
<td>r4</td>
<td>3</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td>s4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r6</td>
<td>r6</td>
<td>5</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s5</td>
<td>s4</td>
<td>6</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s5</td>
<td>s4</td>
<td>7</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s11</td>
<td>s6</td>
<td>8</td>
<td>r1</td>
<td>r1</td>
<td>s7</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
<td>r3</td>
<td>10</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>r5</td>
<td>r5</td>
<td>11</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Of the LR class of parsers, LALR(1) has the best combination of power and practicality:

“LALR(1) parsers are powerful, almost as powerful as LR(1) parsers, they have fairly modest memory requirements, … and they are time-efficient.” [GJ90§9.6 pp. 211]

In fact, LALR(1) parsing is the technique of choice for the majority of parsers [GJ90].

6.5.3.2 SLR Parsing

SLR (Simple LR) parsing is similar to LALR in that they both try to combine automaton states. The combinations used by SLR discard information related to the current state of the parser and combine states based only on the following symbol information. As a direct result of this process SLR parsing is weaker in power with no reduction in complexity to the LALR method:

“Since SLR(1) parsers have the same size as LALR(1) parsers and are considerably less powerful, LALR(1) parsers are generally preferred.” [GJ90§9.6.4, pp. 217]

6.5.3.3 Advantages

LR parsers have many advantages [ASU86]:

- they can be constructed for almost all programming language constructs that are representable by context-free grammars;
- they can be implemented as efficiently as other shift-reduce parsing methods and are the most general of this kind of parser;
- the class of grammars recognisable by LR parsers is a proper superset of those that can be recognised by predictive parsers; and
- syntax errors will be detected as soon as possible on a left-to-right scan of the input.

In fact, this error detection property means that parser will stop at the first incorrect token without performing any further shifts or reduces. This is desirable as it allows the maximum amount of state information to be retained for error recovery [GJ90].

Of all known parsing techniques, LALR parsers are considered to be the best trade-off between power and efficiency.
6.5.3.4 Disadvantages

The principal drawback of LR parsers is that manual parse-table construction is tedious and error-prone. It is simply too much work to construct a parse-table for a typical programming language without using a LR parser generator [ASU86].

While such automated tools simplify parse-table generation considerably, they are not perfect. Grammars that are not LR lead to shift/shift, shift/reduce, or reduce/reduce conflicts. These are the result of combining conflicting actions within the same entry in the parse table. Users must understand why each conflict may arise and know how to modify their grammars accordingly [GJ90, pp. 215].

Additionally, as LR grammars are too large, parser generators typically use the LALR technique (e.g. Berkeley Yacc [Cor00]). This creates an extra source of conflicts that is not so easily handled by users.

“The situation is worse for those (relatively rare) grammars that are LR(1) but not LALR(1). The user never really understands what is wrong with the grammar: the computer should be able to make the right parsing decisions, but it complains that it cannot. Of course there is nothing wrong with the grammar; the LALR(1) method is just marginally too weak to handle it.” [GJ90§9.6.3, pp. 216]

6.5.4 Suitability to Extensible Languages

Of the shift-reduce parsers, operator-precedence parsers only accept operator grammars, LR parsers are impractical in size, SLR is the same size but weaker than LALR, therefore LALR would be the likely choice of shift-reduce parser. As a result while considering the suitability of shift-reduce parsers to extensible languages we will consider LALR parser primarily.

6.5.4.1 Usability

LR methods are unsuitable for extensible languages. The simple examples of Figure 6.8(b) show that is a simple matter to produce grammars that cannot be handled by such parsers. Using a LALR parser only serves to provide more subtle unacceptable grammar errors.
6.5.4.2 Mid-parse Grammar Modification

LR parsers are unsuitable for mid-parse grammar modification as the entire state machine would need to be regenerated upon grammar modification and it is unclear if a corresponding state could be found for the current state.

An ad hoc shift-reduce parser may be able to handle grammar modification well, but such parsers are difficult to construct.
6.6 General Parsing

Unlike the parsers of the previous sections, general parsing allows our (context-free) grammar to be in any form. These parsers should at least be able to meet our first requirement for extensible language parsing — i.e. ease of use for the programmer.

We examine two general parsing mechanisms CYK and Earley parsing and briefly describe the class of chart parsers.

6.6.1 CYK Parsing

The CYK (named after its independent co-creators Cocke [CS70], Younger [You67], and Kasami [Kas65]) algorithm provides parsing of ambiguous grammars, but the standard version requires grammars to be in Chomsky Normal Form (CNF). Context-free grammars can be converted to CNF without too much difficulty [GJ90], so CYK parsing still serves as a good starting point for a general parser. The standard algorithm can be extended to handle forms that are not CNF, but at the cost of a more difficult to implement algorithm.

CYK parsing considers all possible subsequences of the input string starting with those of length one, then two, etc. Once a rule is matched on a subsequence, the left-hand side is considered a possible valid replacement for the underlying subsequence.

Typically standard CYK is implemented using multidimensional Boolean arrays [GJ90]; each entry representing the successfulness of applying a rule to a subsequence (represented by a start index and a sequence length). An algorithm for standard CYK parsers using a multidimensional array is shown in Code Example 6.5.

```c
int N = /* number of input tokens */
int R = /* number of rules in CNF grammar */
bool array[N][N][R];

foreach token T at index I in the input
  foreach rule R -> T
    array[I][1][R] = true

foreach I = 2..N
  foreach J = 1..N-I+1
    foreach K = 1..I-1
      foreach rule R -> S T
        if P[J][K][S] and P[J+K][I-K][T]
          array[I][R] = true
```

Code Example 6.5: CYK Algorithm
CYK parsers operate in a non-directional bottom-up fashion. They are non-directional as they match rules of a given length at all places in the input.

Figure 6.16 shows the execution of a CYK parser for the grammar of Figure 6.15 with input $5 \times 8 + 2$. The Chomsky Normal Form of our simple expression grammar highlights the increase complexity that comes with such a transformation.

| $E$ ::= $E$ $E_2$ | $E$ $E_3$ | ( $E_4$ | num
| $E_2$ ::= + $E$
| $E_3$ ::= * $E$
| $E_4$ ::= $E$
| Figure 6.15: Expression Grammar in Chomsky Normal Form

| $E$: $5 \times (8+2)$
| $E$: $(5 \times 8)+2$
| $E_3$: *$(8+2)$
| $E$: $8+2$
| $E$: $5 \times 8$
| $E_3$: *$8$
| $E_2$: +$2$
| $E$: $5$
| $E$: $8$
| $E$: $2$
| $5$ $*$ $8$ $+$ $2$
| Figure 6.16: CYK Expression Parsing Recognition Table

### 6.6.2 Earley’s Algorithm

Earley’s algorithm [Ear70] is described as a breadth-first top-down parser with bottom-up recognition.

The algorithm maintains a list of states which each contain a list of partially complete rules. These partially complete rules are written with a dot representing the currently examinable position in a rules right-hand side. For example, in $X ::= ab\cdot c$ the terminals $a$ and $b$ have been examined and $c$ is the next terminal to be examined.

At each stage in the parse, the following three actions occur in turn: prediction, scanning, and completion. At each iteration of the algorithm any of these actions may add a partially completed rule to either the current state or the next state. Each rule has associated with it a source state.
Prediction adds to the next state each of the rules for which a non-terminal appears immediately to the right of the most recently parsed symbol in that rule.

Scanning adds to the next state all partially complete rules that expect the current input symbol as their next symbol.

For each completed rule in the current set, completion adds to the current state all rules from the corresponding source state, that have most recently examined (i.e. in which the dot appears immediately to the right of) the entire right-hand side of the completed rule.

Code Example 6.6: Earley’s Algorithm

<table>
<thead>
<tr>
<th>repeat until input is exhausted</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = /* current input symbol */</td>
</tr>
<tr>
<td>k = /* current state index */</td>
</tr>
<tr>
<td>repeat until no more states can be added</td>
</tr>
<tr>
<td>foreach state (X ::= A•YB, j) in state[k] // prediction</td>
</tr>
<tr>
<td>foreach rule (Y ::= C) state[k].add( state (X ::= •C, k) )</td>
</tr>
<tr>
<td>foreach state (X ::= A•aB, j) in state[k] // scanning</td>
</tr>
<tr>
<td>state[k+1].add( state (X ::= AaB, j) )</td>
</tr>
<tr>
<td>foreach state (X ::= A•, j) in state[k] // completion</td>
</tr>
<tr>
<td>foreach state (Y ::= A•B, i) in state[j]</td>
</tr>
<tr>
<td>state[k].add( state (Y ::= A•XB, i) )</td>
</tr>
</tbody>
</table>

Code Example 6.6 shows an algorithm for Earley parsing. In this algorithm, the symbols X and Y represent any non-terminal; and A, B, and C represent any sequence of symbols.

6.6.3 Chart Parsers

There is another form of general parser known as the chart parser. In actuality, both chart parsing and CYK parsing have a number of variants and some of these are identical [GJ90]. The difference between the two approaches is largely implementation based and conceptually they are very similar. The approach taken by both is to repeatedly scan the input looking at larger and larger substrings.

The diagram in Figure 6.17 is a chart based representation of the recognition table from Figure 6.16. The two diagrams contain identical information.

Chart parser variants are often used in the field of natural language processing and are too numerous to list here. As they are so similar, in future sections generally only CYK parsing will be discussed.
6.6.4 Suitability to Extensible Languages

Unlike previously reviewed parsers, general parsers at least partially satisfy our requirements for extensible language parsing.

6.6.4.1 Usability

CYK and Earley parsing do not suffer from the limitations of traditional top-down and bottom-up approaches to parsing. Such general parsers seem well suited to providing a simple to use system for end-users. A general parser should be capable of parsing any grammar the user creates.

A system built with such a parser must have facilities for resolving ambiguity and reporting when ambiguities are not resolved.

6.6.4.2 Mid-parse Grammar Modification

CYK parsing considers all substrings of length one up to the length of the input. This process is unsuitable if we do not know all of the rules in advance.

Earley parsing speculatively keeps track of partially accepted rules in a top-down fashion (even though the rules are accepted in a bottom-up order). If we introduce new rules via an import statement, then these speculations are incomplete.
6.7 Analysis

In this section we first review the relative merits of the parsers reviewed in this chapter both in terms of power and efficiency. Secondly, we examine these parsers suitability to use in extensible languages.

6.7.1 Power

The reviewed parsers can be ordered in terms of power as follows (from weakest to strongest):

- operator-precedence;
- recursive descent and table-driven LL;
- SLR;
- LALR;
- canonical LR;
- CYK; and
- Earley’s algorithm.

With the exception of operator-precedence and CYK, the grammars that each of these parsers can recognise is a proper subset of those of the following parser.

Operator precedence parsing and CYK parsing are the hardest to classify.

On one hand, operator precedence can handle ambiguous grammars as the general methods but it is also restricted to grammar without consecutive non-terminals. The latter is considered to be more restrictive and led to its placing.

As described, CYK parsing requires the grammar to be in Chomksy Normal Form, but translation into CNF is not particularly hard and is far less restrictive than the necessity of removal of left-recursive forms for the LL parsers, for example. Also, with effort CYK parsing can be extended to handle arbitrary grammars.

Earley’s algorithm is the only reviewed method that allows parsing of an arbitrary grammar without any modification.
6.7.2 Efficiency

Of the often used parsers (which all have a lookahead of 1) running times are proportional to $O(\text{length of input})$. However, further to this, the reviewed parsers can be ordered in terms of time efficiency as follows (from slowest to fastest, with complexities given in terms of $n$ input tokens):

- Earley’s algorithm and CYK: $O(n^3)$;
- recursive descent: $O(n)$; and
- table-driven LL, and LR: $O(n)$.

In terms of space efficiency (from largest to smallest, with complexities given in terms of $n$ input tokens, $t$ terminal grammar symbols, $u$ non-terminal grammar symbols, and $s$ states):

- CYK: $O(un^2)$;
- Earley’s algorithm: $O(n^3)$;
- canonical LR: $O(s(t + u))$;
- LALR and SLR: $O(s(t + u))$ where the number of states is less than or equal to that of LR;
- recursive descent; and
- table-driven LL: $O(tu)$.

6.7.3 Suitability to Extensible Languages

As we have seen in previous sections, none of the traditional parsers have exactly met our requirements for extensible language parsing, namely the requirements of programmer usability (i.e. no unusual grammar restrictions) and mid-parse grammar modification (to allow for arbitrary grammar changes).

6.7.3.1 Usability

Keeping things simple for the user suggests that all LL and LR methods are unsuitable. The simple examples of Figure 6.8 show that is all too easy to produce grammars that cannot be handled by such parsers.

CYK and Earley parsing do not suffer from these limitations and these general parsers seem well suited to providing a simple-to-use system for end-users.
6.7.3.2 Mid-parse Grammar Modification

As previously stated, CYK parsers first consider substrings of length one, then of length two, and so forth, up to the entire length of the input. This process is clearly inappropriate for grammars for which we do not know all of the rules in advance.

Similarly, Earley parsing uses a form of speculation by allowing partially accepted rules. If we introduce new rules via an import statement, then these speculations are incomplete.

Both LL parsers and LR parsers suffer from similar problems to Earley’s method. With LL parsing the parser may be deep within a recursive call when the grammar is modified and as such may be unable to incorporate the new grammar rules. In fact, top-down recognition is invalid if the entire grammar is unknown. There is no way to ascertain that a rule will be matched without the rules that generate it being known. The LR parser’s entire state machine would need to be regenerated upon grammar modification and it is unclear if a corresponding state could be found for the current state.

An ad hoc shift-reduce parser may be able to handle grammar modification well, but such parsers are difficult to construct.

Indeed, none of the traditional methods seem to fit the domain of extensible languages perfectly and another solution must be found — even if it comes at the cost of efficiency.
7.1 Overview

As discussed in section 6.7.3, in order to successfully provide a parser for extensible languages we require two things:

- end-user simplicity; and
- the ability to handle mid-parse grammar modification.

The usability of the system led to the necessity of general parsing mechanisms as handling a subclass of context-free grammars is confusing for the macro programmer. To this end it was also desirable not to have to mangle the grammar before use.

Handling mid-parse grammar changes requires us to provide a bottom-up parser that examines the input in a left-to-right fashion.

No existing method fitted these requirements so a new parser was constructed which we call Graph Expansion Parsing.

This chapter first shows the development of the parser in section 7.2. We then look at an efficiency improvement in section 7.2.3 which provides the final algorithm.

The efficiency of Graph Expansion Parsing is examined in section 9.5.

7.1.1 Similarities to Chart Parsing Methods

Graph expansion parsing is clearly a type of chart parser. However it was initially developed in isolation without reference to any general parsing method.

The single pass algorithm (see subsection 7.2.2) could be viewed as a modified form of a CYK parser that has been extended to handle grammars other than those in Chomsky Normal Form.

The final algorithm (see subsection 7.2.3) uses techniques to limit the construction of paths and to simultaneously compare multiple rules against the current path that are similar in intent to Earley’s improvements over top-down breadth-first parsing techniques.
7.2 Development

Two initial algorithms were produced for graph expansion parsing. The first attempt was to test the feasibility of providing generality and as a result was very inefficient, but importantly, it met the goal of mid-parse grammar modification. The second algorithm was designed to have a stronger concept of completion.

Lastly, an optimised version of the second algorithm from this section is introduced.

7.2.1 Multipass Method

The original technique for building a graph of all possible parses and subparses involved making a series of passes through the entire input string.

The algorithm begins with a graph of the input — in all examples in this section just a trivially linear graph, but the algorithm does not restrict the complexity of the input. Each vertex in the graph contains a forward edge that contain a single token of input — for example, the input for the simple string $5*8+2$ is shown in Figure 7.1(a).

The algorithm iterates through each of the vertices in order. All forward paths (that are no longer than the longest right-hand side of any grammar rule) from each vertex have their edge values matched against all grammar rules of equal length. If a match is found, a new edge is added from the beginning to the end of the path with its value being the left-hand side of the grammar production.

One complete iteration through the vertices does not produce all of the possible rule reductions, so this process is repeated until no further additions are made to the graph. An algorithm for this multipass method is shown in Code Example 7.1.

```
repeat until no changes made
    foreach vertex V in original set
        foreach forward path (V, U) of length less than or equal to longest rule
            foreach rule R in rule set of equal length to the path (V, U)
                if R's right-hand side matches path values
                    add a new edge from (V, U) with R's left-hand side as value
            if there exists an edge from start vertex to end vertex the entire input has been recognised
```

Code Example 7.1: Multipass Graph Expansion Algorithm
An example execution of this technique for the grammar of Figure 6.3 with the initial input $8*5+2$ is shown in Figure 7.1 (with $\text{expr}$ abbreviated to $E$). Even for such a simple test case, the algorithm requires four entire passes of the input be completed.

There is one major problem with this technique. The multiple passes are inefficient, and each pass must consider an increasing number of path possibilities as the complexity of the graph increases. The final pass only serves to provide termination yet it takes the most time.

This algorithm succeeds in parsing arbitrary grammars and at mid-pass grammar modification. Though the multiple passes are inefficient, they do allow a newly modified grammar to be applied to the entire input.
The multipass method is a non-directional bottom-up parser. If restricted to paths of a fixed (but increasing) length for each pass, it acts as a generalised CYK parser.

### 7.2.2 Single Pass Method

The major problem with the multipass method is that for each vertex iteration in a pass, rules are being matched against paths that contain vertices that have not been examined in the current pass. This means matching has not been performed between the ruleset and the majority of the subpaths of the path currently being examined (i.e. all subpaths not containing the left-most vertex).

The idea behind the single pass method is to scan the vertices from left-to-right, but to consider the paths from the current vertex from right-to-left (i.e. backward paths not forward paths). This approach ensures that all subpaths have been fully compared against the rule set. It also ensures that once the last vertex is examined that all possible parses have been generated.

The single pass graph expansion algorithm is shown in Code Example 7.2.

```plaintext
foreach vertex V in original set
  foreach backward path (U, V) of length less than or equal to longest rule
    foreach rule R in rule set of equal length to path (U, V)
      if R's right-hand side matches path values
        add a new edge from (U, V) with R's left-hand side as value
      if there exists an edge from start vertex to end vertex the entire input has been recognised

Code Example 7.2: Single Pass Graph Expansion Algorithm
```

Figure 7.2 shows an example execution of the single pass technique for the grammar of Figure 6.3 with the initial input of $8 \times 5 + 2$. The current node is highlighted at each step. The resultant graph is identical to that produced by the multipass technique shown in Figure 7.1 but it is produced in a more efficient fashion.
The paths that are examined by the algorithm during the production of Figure 7.2 and when new edges are added to the graph are demonstrated in Table 7.1. The only non-terminal in the grammar of Figure 6.3 is for that of $expr$, so each time a new expression is found it is given a subscript so that the process is easier to follow. Each generated path is no longer than the longest rule in the grammar and must be compared to all grammar rules for a match.

The single pass method is left-to-right scanning bottom-up parser. On non-ambiguous grammars it will produce a single right-derivation traced out in reverse in a similar fashion to a LR parser.
Table 7.1: Single Pass Graph Expansion Evaluation

<table>
<thead>
<tr>
<th>Iteration / Vertex</th>
<th>Examined Path</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(8)</td>
<td>→ E₁ on edge (0, 1)</td>
</tr>
<tr>
<td></td>
<td>(E₁)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(*)</td>
<td>→ E₂ on edge (2, 3)</td>
</tr>
<tr>
<td></td>
<td>(8, *)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₁, *)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(5)</td>
<td>→ E₃ on edge (0, 3)</td>
</tr>
<tr>
<td></td>
<td>(*, 5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(8, *, 5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₁, *, 5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₂)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(*, E₂)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(8, *, E₂)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₁, *, E₂)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₃)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(+)</td>
<td>→ E₄ on edge (4, 5)</td>
</tr>
<tr>
<td></td>
<td>(5, +)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(*, 5, +)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₂, +)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(*, E₂, +)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₃, +)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(2)</td>
<td>→ E₅ on edge (2, 5)</td>
</tr>
<tr>
<td></td>
<td>(+, 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(5, +, 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₂, +, 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₃, +, 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₄)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(+, E₄)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₂, +, E₄)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₃, +, E₄)</td>
<td>→ E₆ on edge (0, 5)</td>
</tr>
<tr>
<td></td>
<td>(E₅)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(*, E₅)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(8, *, E₅)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₁, *, E₅)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₆)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(E₇)</td>
<td>→ E₇ on edge (0, 5)</td>
</tr>
</tbody>
</table>
7.2.3 Optimised Method

Many of the paths examined in Table 7.1 are clearly not going to match any of the rules of the simple expression grammar. For example, no rule ends with a + symbol yet in iteration 4 we examined six paths that end with a + symbol.

The optimised algorithm does not continue to expand paths unnecessarily. It keeps track of how many possible rules contain the current path as the right-most part of their right-hand side, and when this falls to zero the current path is abandoned. Its execution produces the same order of graph additions as the single pass method of the previous subsection.

Code Example 7.3 contains an algorithm for this optimised method. Paths are generated incrementally by the recursive algorithm `check`.

```plaintext
parse
   foreach vertex V in original set
      check (V, V)
      if there exists an edge from start vertex to end vertex the entire input has been recognised
      check (path (U, V))
         foreach backward edge (T, U)
            if path (T, V) matches a grammar rule R's right-hand side
               add a new edge from (T, V) with R's left-hand side as value
               if further possibilities end with this subsequence
                  check (T, V)
```

Code Example 7.3: Final Graph Expansion Algorithm

The difficult part of this algorithm is determining whether the current path matches the right subsection of the right-hand side of a rule (or set of rules). To aid in this process a tree of partial matches is used. Figure 7.3 contains this tree of partial matches for the simple expression grammar of Figure 6.3. The arcs show the matched tokens, and the nodes contain the addition rules that represent the reduction of rules.
Figure 7.3: Simple Expression Partial Match Tree

Although this tree contains add actions only at the nodes of the tree this is not typical of more complicated grammars. These add actions may appear at any node in the tree. For example, with a right-to-left parse of the classic “dangling-else” grammar both the if-then statement and the if-then-else statement are contained in a single path in the tree (see Figure 7.4). A more contrived left-to-right example would be for a grammar containing bracketed expressions and single parameter procedure calls.

Table 7.2 shows the paths examined by the final Graph Expansion Parser for the simple expression grammar with input $8*5+2$. These are a strict subset of those in Table 7.1.

Table 7.2: Final Algorithm Graph Expansion Evaluation

<table>
<thead>
<tr>
<th>Iteration / Vertex</th>
<th>Examined Path</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(8) (E₁)</td>
<td>$\rightarrow E₁$ on edge (0, 1)</td>
</tr>
<tr>
<td></td>
<td>(E₂)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(*)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(5) (E₁, *, E₂)</td>
<td>$\rightarrow E₂$ on edge (2, 3)</td>
</tr>
<tr>
<td></td>
<td>(E₃)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4</td>
<td>(+)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(2)</td>
<td>→ $E_4$ on edge (4, 5)</td>
</tr>
<tr>
<td></td>
<td>($E_4$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(+, $E_4$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>($E_2$, +, $E_4$)</td>
<td>→ $E_5$ on edge (2, 5)</td>
</tr>
<tr>
<td></td>
<td>($E_3$, +, $E_4$)</td>
<td>→ $E_6$ on edge (0, 5)</td>
</tr>
<tr>
<td></td>
<td>($E_5$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(*, $E_5$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(8, *, $E_5$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>($E_1$, *, $E_5$)</td>
<td>→ $E_7$ on edge (0, 5)</td>
</tr>
<tr>
<td></td>
<td>($E_6$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>($E_7$)</td>
<td></td>
</tr>
</tbody>
</table>
Implementation

1. Introduction
2. Defining extensibility
3. Assessing extensibility
4. Reviewing extensibility
5. Designing the language
6. Reviewing parsing
7. Implementing a parser
8. Language implementation
9. Evaluation
10. Conclusion
8.1 Overview

As illustrated in Figure 8.1, the Genesis compiler:

- takes a single Genesis source file as input;
- tokenises this file;
- parses this file (using the graph expansion parser from the previous chapter), performing macro translations where appropriate;
- produces Java source files;
- compiles these with the standard Java compiler (javac); and
- finally produces Java executables (.class files).

In addition, the parser initially imports standard macros and may import further macros as specified by the Genesis source file.

![Figure 8.1: Genesis Compiler Structure](image)

The major components of the Genesis compiler implementation are the tokeniser and parser.

There are two facets to Genesis source file translation: translation of macro calls to standard Java forms and the translation of macro definitions. The latter requires a representation to be chosen that allows for future importing in other source files via the macro import mechanism.

This chapter begins by describing the chosen representation for macro definition translation (section 8.2) and the import mechanism (section 8.3).

Following this, the tokeniser (see section 8.4) and the parser (see section 8.5) implementations are described.

Section 8.6 describes the actual usage of the compiler.

Finally, the implementation of standard library extensions is detailed in section 8.7.
8.2 Macro Definition Translation

The translation of macro definitions requires the preservation of macros in a form that is convenient to our modified import statement. It was chosen from the onset to embed macros within classes directly and to use reflection for retrieval of macros at compile-time. By using standard Java features, we allow macros to be easily packaged with their associated classes, with no need for extra files, new file formats or any other such complexity.

8.2.1 Basic Translation

Take as way of illustration the following code snippet in Code Example 8.1(a). Here we have a class which has two macro definitions in amongst other standard code (this code could use macros, but there are no further macro definitions).

```java
class Example {
    // some code
    ...
    Expression macro (...) precedence 0.6 { ... }
    ...
    // some other code
    ...
    LiteralList macro (...) throws SyntaxError { ... }
    // yet more code
}
```

(a) Before Translation

```java
class Example {
    // some code
    ...
    // some other code
    ...
    // yet more code
    static class Macros {
        static Expression mangledName1(...) { ... }
        static LiteralList mangledName2(...) throws SyntaxError { ... }
    }
}
```

(b) After Translation

Code Example 8.1: Basic Macro Translation

Upon translation, all macro declarations found within a class are collected as a group inside a static inner class Macros – as a result of this decision it is necessary to ensure that no program is allowed to directly define an inner class of this name. So for this example code the translation would be as shown in Code Example 8.1(b).
All macros become static methods of the static class Macros. Their return types, any 

`throws` declarations, and their body are maintained without change. As we will see in 

the following sections the formal parameter list and name of each method depends on 

the formal parameters of the macro; its precedence and associativity; and, in some 

cases, modifications made to avoid name clashes.

### 8.2.2 Name Mangling

In C++, in order to ensure there are no name clashes due to overloading, each C++ 

name is *mangled* into an unique identifier that is dependent on the properties of the 

original function. In this work, a similar process is applied to each macro.

Within the inner class Macros each macro is placed in a method that contains all of the 

macro’s properties in both its name and its arguments. This relatively simply name 

mangling process relies on placing special significance on both the “$” and “_” symbols 

in order to be able to reconstruct the full macro upon its subsequent import.

#### 8.2.2.1 Terminals and Non-terminals

Each argument in a macro definition is mangled into the final name of the macro. Each 

terminal becomes part of the macro name as it appears, and in place of each non-
terminal a dollar symbol is placed. For the arguments, all terminals are dropped, and all 

non-terminals appear with their original ordering. Mangling is demonstrated in Code 

Example 8.2.

```c++
macro For (forall, FormalParameter p, in, Expression e, Statement b) ...
```

(a) Before Mangling

```c++
static For forall$in$(FormalParameter p, Expression e, Statement b) ...
```

(b) After Mangling

**Code Example 8.2: Basic Name Mangling**

The final mangled name of a macro clearly shows the number and position of both 
terminals and non terminals, and the parameters show the types of each the terminals. It 
is simply a matter of pulling this information out upon importing a macro at a later time.

There are a few situations that may arise that cannot be translated directly using the 
above scheme alone, none cause large problems however, and can all be dealt with a 

single solution: the introduction of leading underscore(s). These leading underscores are
discarded when the name is demangled. The next few subsections detail these situations and provide examples of their translation.

**Java Reserved Words**

A macro can be defined that contains a single terminal that happens to be a Java reserved word. This is done extensively in defining the Java primitives, but as always, a programmer may wish to override these also.

For example, the Java primitives `true` and `false` are simply defined as shown in Code Example 8.3(a), and standard translation would produce the mangling in Code Example 8.3(b). It may not be immediately apparent what the issue is here, but `true` and `false` are Java reserved words, and as such cannot ever be used as identifiers. The solution is simply to introduce a leading underscore into the mangling as shown in Code Example 8.3(c).

```
macro LiteralBoolean true() { return new LiteralBoolean(true); }
macro LiteralBoolean false() { return new LiteralBoolean(false); }
```

(a) Before Mangling

```
static LiteralBoolean true() { return new LiteralBoolean(true); }
static LiteralBoolean false() { return new LiteralBoolean(false); }
```

(b) After Basic Mangling

```
static LiteralBoolean _true() { return new LiteralBoolean(true); }
static LiteralBoolean _false() { return new LiteralBoolean(false); }
```

(c) Final Mangling

Code Example 8.3: Java Reserved Word Mangling

Reserved words appearing in the midst of a more complicated macro cause no issues and do not require the leading underscore mangle.

**Non-Java Terminals**

Java identifiers are not allowed to begin with a digit (although they are allowed to contain digits at any other point), whereas there is no such restrictions placed on macro terminals. When a straight conversion occurs of a macro that has a terminals that begins with a digit as its first argument it is necessary to append a leading underscore so that the resulting mangle does not begin with a digit.
**Name Conflicts**

The same macro may appear in different imports with the same name and types without causing conflict to the compiler. A macro with the same arguments with differing return types may even appear in the same import.

Of the three macros in Code Example 8.4, only one will successfully match on a given token, but all three will make an attempt to match. In order to remove the conflict that these definitions produce (Java doesn’t allow overloading on return type alone), one or more leading underscores may be appended to the mangled name.

```
macro LiteralString (Token t) throws ConditionsNotMet ...
macro LiteralChar (Token t) throws ConditionsNotMet ...
macro LiteralInteger (Token t) throws ConditionsNotMet ...
```

(a) Before Mangling

```
static Expression printf_040$_044$_041(LiteralString s, Arguments list) ...
```

(b) After Mangling

**Code Example 8.4: Mangled Name Conflict Resolution**

The choice of which macro(s) to append the underscores to is arbitrary, as this information will be discarded upon import.

**8.2.2.2 Symbols**

Symbols may appear at any argument position within a macro. Symbol terminals cannot be translated as is, as most symbol characters are not acceptable Java identifier characters. Each symbol character is translated to an underscore plus three digits that represent the decimal ASCII value of the symbol. Code Example 8.5 demonstrates the mangling of symbols for a simple macro.

```
macro Statement (printf, (, LiteralString s, ",", Arguments list, )) ...
```

(a) Before Mangling

```
static Expression printf_040$_044$_041(LiteralString s, Arguments list) ...
```

(b) After Mangling

**Code Example 8.5: Symbol Mangling**

---

11 Unicode source files are not currently supported by the Genesis research compiler.
**Multicharacter Symbols**

It is possible to define a macro that uses a multi-character symbol, or a macro that requires two consecutive single symbols. These two cases must be differentiated. A multi-character symbol will be mangled as an underscore followed by multiple three digit groups, e.g. "( )" will be mangled to _040041, whereas "(" ")" will be mangled to _040_041.

### 8.2.2.3 Precedence

A macro’s precedence is specified by a real number between 0 and 1 inclusive and defaults to 0.5. This precedence is mangled by appending a prefix to a previously mangled name; an underscore followed by three zeros, followed by the significant part of the precedence. Figure 8.2 contains some examples of the mangling of precedences.

![Figure 8.2: Precedence Mangling Examples](image)

The special case of having a precedence of 1.0 requires its own special mangle and will be simply mangled to _001.

### 8.2.2.4 Associativity

Macros that specify that they are right-associative (rather than the default left-associative) have a prefix of _002 added.

### 8.2.2.5 Delayed Macros

Macros that specify that their execution is to be delayed until the surrounding context has been parsed have a prefix of _003 added.

### 8.2.2.6 Mangling Grammar and Algorithm

The grammar for producing a mangled name (and for demangling) is provided in Figure 8.3.

The mangling process will always produce mangled names that follow this strict ordering of precedence–associativity–delayed, but the demangling process can be more robust and is free to detect these in any order, just in case.
The algorithm for mangling a macro definition into a Java method call is shown in Figure 8.4 (in pseudo-code).

```java
name = ""
arguments = empty

if macro specifies precedence
    name += "_00" + stripDecimalPoint(precedence)

if macro specifies right-associativity
    name += "_002"

if macro specifies delayed execution
    name += "_003"

foreach macroParameter // (in order)
    if terminal
        if symbol
            name += "_" + asciiValuesOf(macroparameter)
        else
            name += macroParameter
    else
        name += "$"
        arguments += macroParameter

while name begins with digit or is reserved word or conflicts
    name = "_" + name
```

Figure 8.4: Name Mangling Algorithm
8.3 Import Mechanism

Upon encountering a class import statement, the implementation interrogates the class via reflection to see if it contains a static inner class called Macros. If such a class is detected, all methods of that class are examined.

The mangled name of each method is examined, and a structure is built-up that contains:

- a precedence (defaulting to 0.5);
- an associativity (defaulting to left);
- a list of terminals and non-terminals;
- a return type; and
- a reference to the static method (to be used later as the action of this macro).

A full description of the data structures used for storing imported macros is discussed in subsection 8.5.2.

Any method that is encountered within the Macros inner class that does not represent a well-formed macro will cause a warning to be generated. Such errors occur upon encountering:

- a non-static method;
- zero argument macro definitions;
- a method that does not contain a matching number of parameter placeholders and formal parameters; or
- parameters that do not inherit from or implement AbstractSyntax.

No further information is extracted from the class. For the purposes of this implementation, reflection is used to glean all member information and when performing type checks — none of this information is cached at any time. However, a record of each import must still be maintained to enabled expansion of shortened names when using this reflection technique. For example following import somewhere.Utils, any use of Utils.someMethod must expand to somewhere_UTILS.someMethod.
8.4 Tokeniser

The tokeniser is implemented directly from the transition diagram from section 5.4.4, but unlike most language tokenisers it does not produce a lazy stream of tokens but a fully constructed graph ready for the Graph Expansion Parser. In simple cases this graph appears as a list. For example, consider the code fragment from Code Example 8.6.

```c
if (frogs > toads) x = -x;
```

Code Example 8.6: Code Fragment Without Multi-character Symbols

For this simple `if` statement the tokeniser would produce the graph shown in Figure 8.5.

![Figure 8.5: Graph Produced From Tokenising Code Example 8.6](image)

The tokeniser constructs these graphs with a single type on each arc: Token. No distinction is made between tokens that are fully alphanumeric or symbolic. Indeed, even string and character literals are passed to the parser as this generic token type.

A general policy of the tokeniser design is to not make decisions about the type of any character sequences — this is the job of macros that deal directly with tokens, in this way the tokeniser provides the parser with the most flexible input. The tokeniser acts more as a device to separate tokens rather than to classify them.

8.4.1 String and Character Literals

Both string and character literals are detected at the tokenising stage, and added to the graph with their enclosing quotes intact. It is left to macro definitions to provide correct categorisation of these tokens.

Such macros merely take a single token as input, ensure that the token is in the correct form (raising a quiet exception if this is not the case) and convert it into a more meaningful form — such as a literal or an identifier. Many such macros are likely to attempt (and fail) on each token of the input until the token is correctly categorised.
8.4.2 Multi-character Symbols

As discussed in section 5.4.3 multi-character symbols require special treatment. The production of symbol combinations described in section 5.4.3.4 fits snuggly with Graph Expansion Parsing. Each combination is represented as an extra arc on the graph. For example, consider the code fragment in Code Example 8.7.

```
x+= (y4) - 400.3;
```

Code Example 8.7: Code Fragment Containing Multi-character Symbols

For this simple expression statement, the graph shown in Figure 8.6 is produced.

![Figure 8.6: Graph Produced From Tokenising Code Example 8.7](image)

The number of arcs required for a symbol sequence grows quadratically with the number of paths growing exponentially, but, as previously discussed, the length of multi-character symbols is typical just a few characters.
8.5 Parser

The parser is an implementation of Graph Expansion Parsing, but with a few interesting additions and optimisations.

8.5.1 Sub-type Non-terminal Matching

The standard operation of GEP is that the type of each non-terminal symbol from the right-hand side of a rule must be matched exactly with an arc on the graph. The Genesis parser allows matching to take place if the type of each object on the arc is a sub-type of the corresponding non-terminal symbol from a macro.

This allows for the set of Java abstract syntax classes to be defined as a traditional object-oriented class hierarchy (see Appendix A). Additionally, such a technique also allows for implicit optimisations as macros do not have to be created to coerce abstract syntax classes into types higher up the abstract syntax hierarchy.

For example, for the intuitive ambiguous simple expression grammar of Figure 6.3, it is possible to create a class (or interface) for an expression and four subclasses that correspond to addition, multiplication, bracketing, and a simple number. There is no need to create a class for representing the number and another class for an expression that is simply a number, these two classes can be combined. On a larger scale this technique has the potential to provide a much simplified grammar.

8.5.2 Partial Match Tree

The partial match tree as described in subsection 7.2.3 and Figure 7.3 is implemented, with efficiency as the prime concern, with a hash-table at each node. When following a path through the graph, each lookup in the table is a near constant operation.

The partial matches tree is built incrementally as each new macro is imported. The hashtables initially start at a very small size, but will automatically grow as required, so the space overhead for such a scheme is not as expensive as it may appear at first glance.

The sub-type matching scheme outlined in the previous subsection requires that each sub-type possibility is represented in the tree for maximum efficiency. The alternative
approach is to merely check for each sub-type as the path is followed. It is possible to construct a hybrid approach where sub-types are only expanded after their first use.

### 8.5.3 Abstract Syntax Tree

In addition to storing the abstract syntax objects creation by macro expansion as the parse progresses, the parser stores references to the grammar rule and all of the source edges that were successfully matched. This provides the necessary information for disambiguation, if necessary, at a later stage in the parse.

When a macro is delayed, an uninitialised placeholder object of its return type is placed in the graph so that the parse can continue as expected in an inside-out fashion. Once the delayed macro has been expanded in an outside-in fashion, the placeholder object is updated to reflect the object returned.

For macros that are delayed and may fail if further conditions are not satisfied, the parser also stores a list of further macros that may take its place.

### 8.5.4 Error Handling

The Genesis parser provides limited automatic syntax error handling, but does provide the user powerful facilities for explicit detection of errors within macro expansions.

#### 8.5.4.1 Syntax Errors

Determining the exact location of syntax errors is difficult with Graph Expansion Parsing. Detecting a syntax error is simple, if an arc does not appear in the graph that spans the entire input then a syntax error has occurred. Where this syntax error occurred is not a simple matter to determine from examining the graph.

The Genesis parser simply finds the areas of the graph where the least matching of macros has taken place and signals the user that these might be where the error has occurred. At the simplest level of error, an unrecognised symbol, this scheme will work very well. It is less clear that this approach will be as useful for more subtle syntax errors however. In particular, a syntax error for a particular construct might still be partially correct for another and the simple error detection scheme is unlikely to identify this partially matched area as a possible source of error.
8.5.4.2 Exception Errors

The parser allows macros to throw any exception that extends `ParserException`. If a quiet exception is thrown, the parser will continue as if nothing happened. When errors or warnings are thrown, the parser stores this information until it is sure that the macro that caused the exception will contribute to the final parse.
8.6 Standard Usage

The Genesis compiler is invoked at the command line by `genc <filename>`. Source files must end with a `.gen` file extension.

In addition to standard operation there are a number of command-line arguments that can be specified.

8.6.1 Command-line Arguments

The Genesis compiler supports command line arguments for: modifying the classpath, modifying the default import classes, and producing Java output.

8.6.1.1 Classpath

Like `javac`, `genc` uses the environmental variable `CLASSPATH` and provides a `classpath` switch to allow for easy overriding of this location.

8.6.1.2 Default Imports

By default the compiler imports all of the standard Java abstract syntax classes, as well as the Genesis import statement and macro declaration classes. These default classes can be added to by the use of the `import` switch, or completely overridden by the `importonly` switch.

The former allows easy extension of the Java language without the necessity for an import in every source file. In effect, the Genesis compiler can act as a compiler for an extension transparently to the end-user.

The latter allows for languages to be created that are entirely free from Java syntax, but nonetheless produce valid Java code. This is possible because macro definitions are only necessary for matching source code, the compiled macros are still capable of producing Java abstract syntax. Using this switch with the Haskell subset extension (as defined in section 3.4.2.3) can allow source files to contain nothing but Haskell subset definitions. See section 9.2.6.6 for more explanation.
8.6.1.3 Production of Java Source Code

The compiler can output intermediary Java code rather than compiling it. Output code will be placed in a file with the same name as the Genesis source, but with a .java extension rather than .gen.

Production of the full translated Genesis source code as its Java equivalent allows for a low-level detection of bugs in both user-defined macros and the fledgling Genesis compiler.
8.7 Standard Library Extensions

Genesis provides extensions as standard to aid with macro construction.

In subsection 8.7.1 facilities are provided for quasi-quotation, unquoting, and hygiene as these are considered essential for producing concise and human-readable code. In addition to these shorthands, macros are provided to simplify the declaration of macros in subsection 8.7.2.

8.7.1 Quasi-quotation

The initial implementation of quasi-quotation is actually relatively straightforward. However, it does rely on each object on the parse tree "knowing" which macro was invoked in order to produce it. The syntax for quasi-quotation in Genesis is shown in Code Example 8.8(a).

\[
\{\{ \text{throw } \text{new } \text{TestAssert.AssertionError("Assertion Failed")}; \}\}
\]

(a) Throw Statement with Quasi-quotation

```java
new Throw(new Creation( 
    new Type(new Name("TestAssert"), add("AssertException")), 
    new Arguments(new LiteralString("Assertion Failed")) 
))
```

(b) Handwritten Code to Produce Throw Statement

```java
Statement.Macros.$0006throw$_059(Expression.Macros.new$_040$_041( 
    Type.Macros.$(Name.Macros.$(new new new new Identifier("AssertionError"))), 
    Expression.Macros.$(new new new new LiteralString("Assertion Failed")) 
))
```

(c) Quasi-quotation Code to Produce Throw Statement

Code Example 8.8: Genesis Quasi-quotation

Code Example 8.8(b) contains typical handwritten code for producing the same effect as the quasi-quotation in Code Example 8.8(a). Whilst it would be feasible to provide creation of such code as the translation of the quasi-quotation form, it would rely on constructors being made available in such a form that would be simple to use by the translator. Such handwritten code has various shortcuts for easing the burden on the programmer that would be more difficult to utilise in an automatic translation.

As described in subsection 8.5.3, among the information stored during the parsing process is the macro used for creation of each object in the parse tree. The production of quasi-quotation translation code of Code Example 8.8(c) utilises this information.
Quasi-quotiation translation creates new objects directly for objects that are produced directly from tokens in the parse to avoid having to handle `ConditionsNotMet` exceptions for these simple cases.

Indeed, if macros that may throw exceptions are used within a quasi-quotiation they must be handled by the method containing the quasi-quotiation, either by enclosing the quasi-quotiation in an appropriate `try-catch` statement or by adding them to the `throws` clause.

The basic concept of the implementation of quasi-quotiation is shown in Code Example 8.9. The basic concept is to undertake a treewalk and reproduce the call to the macro that produced each encountered node.

```java
Expression produceQuasi(AbstractSyntax a) {
    ArcValue v = Utils.getAbstractSyntaxInfo(a);
    ExpressionList args = new ExpressionList();
    forall (ArcValue w) in v.sources args.add(produceQuasi(w.data));
    return new MethodCall(v.rule.action, args);
}
macro Expression ("{{", AbstractSyntax a, "}") { return produceQuasi(a); }
```

**Code Example 8.9: Partial Basic Quasi-quotiation Definition**

Abstract syntax classes representing identifiers and literals (i.e. anything taking a token as its argument and promoting it to some more useful type) require slightly more work. As these macros are not delayed but may throw `ConditionsNotMet` exceptions it is simpler for the end user if they are given special treatment and translated into an instantiation instead. Such a facility is not provided for user-defined macros that have the same usage pattern and such exceptions must be handled by the end-user.

Despite this limitation, quasi-quotiation is general enough to handle all user-defined types automatically. This one simple framework will successfully produce code that will reproduce the required syntax tree.

### 8.7.1.1 Hygiene

Hygiene requires another modification of the simple definition of Code Example 8.9. Each occurrence of a variable declaration encountered during the treewalk must be translated into code that generates a fresh variable name — much as a programmer would do by hand to avoid name conflicts. As an addition complication, any

- 224 -
occurrences of the variable name in the following subtree must be replaced with references to the freshly created variable name. This process is implemented via another treewalk and is relatively straightforward.

### 8.7.1.2 Unquoting

The implementation of unquoting creates more complications than that of quasi-quotation. The result of performing an unquote must provide the parser with the correct typing information so that the parse can proceed as expected. Unfortunately there is no way around this situation but to provide an unquote macro for each abstract syntax class that it is desirable to unquote. While the quasi-quotation definition works for any user-defined type, unquoting support must be explicitly provided.

Code Example 8.10 demonstrates the basic requirements for implementing unquoting for a given abstract syntax class. In this example, unquoting is provided for two statement classes: `For` and `While`. For each class for which unquoting is required, a class that allows parsing to proceed as required yet stores the unquoted expression must be created.

```java
interface Unquoted {
}
class ForUnquoted extends For implements Unquoted {
    Expression storedForLater;
    ...
}
class WhileUnquoted extends While implements Unquoted {
    Expression storedForLater;
    ...
}
delayed macro ForUnquoted (`, Expression e) { ... }
delayed macro WhileUnquoted (`, Expression e) { ... }

Code Example 8.10: Unquoting Implementation
```

Changes to the quasi-quotation definition are again required to handle the addition of unquoting — although the modification is trivial. On encountering an unquote the stored expression must be added unchanged to the built list of creations.

At the point at which an unquote is reached the possible interpretations are numerous and cause a large number of edges to be added to the parse graph — the majority of these edges lead to successful parses however. In some cases the number of possibilities can be quite large, for example an unquoting of a statement will lead to all possible statement unquotings being carried through the entire parse. For this reason, unquoting
must be delayed to ensure that the actual static type of the unquoted expression can be used to correctly resolve the ambiguity.

### 8.7.2 Macro Definition Shorthands

Many macro definition shorthands are provided:

- automatic generation of abstract syntax tree construction macro;
- automatic list class and associated macro generation;
- optional macro parameters; and
- statically type-checked parameters.

These macro definitions ease the burden of producing repetitive code from the macro programmer. Of these four shorthands, only optional macro parameters has a fully fleshed out implementation, the other three use much of the same techniques and little would be gained by demonstrating their implementations also.

#### 8.7.2.1 Automatic Construction Macros

As so many macros are simply used for the construction of abstract syntax classes, automating this to a degree seems like an obvious improvement. The \texttt{macroSyntax} macro simply creates a macro with an identical header to how it was originally called and a body containing a call to a constructor for its return type passing through all of its non-terminal arguments. For a straight-forward example of the use of this macro, see Code Example 8.11.

To simplify things further, it is possible to provide multiple definitions for a single return type as demonstrated in Code Example 8.12.

#### 8.7.2.2 Automatic Lists

Much use is made of lists of abstract syntax classes (these lists themselves are also abstract syntax). To simplify this process a \texttt{macroList} macro is provided that specifies the name of the list class to be created, the type of abstract syntax class to collect, and an optional separator. This macro expands into a class with appropriate constructors and two associated macros that allow for construction of the list.

For a simple example, see Figure 8.7(b).
8.7.2.3 Optional Macro Parameters

Optional macro parameters remove the burden of producing multiple similar macro definitions from the macro programmer.

A simplified version of the method declaration fragment from Figure 5.2 is restated in Figure 8.7. The EBNF definition of a simplified method declaration grammar demonstrates both optional components and list definitions. Demonstrated in Figure 8.7(b) is the usage of optional macro parameter shorthand (i.e. Modifiers m?) and the list shorthand (i.e. macroList Modifiers(Modifier)).

\[
\text{method ::= \{modifiers\} return_type name ( ) block}
\]

\[
\text{modifiers ::= modifier modifiers | modifier}
\]

(a) EBNF Definition with Optional Components and a List Definition

\[
\text{macro MethodDeclaration (Modifiers m?, Type t, Identifier i, (, ), Block b) \{ }
\]
\[
\text{macroList Modifiers(Modifier);}
\]

(b) Macro Definitions Using Shorthands

Figure 8.7: Method Declaration Fragment

The implementation defines:

- an extended macro formal parameter class;
- new definitions in order to collect extended parameters (along with normal macro formal parameters) into a parameter list; and
- a set of macros defining new macro definitions with extended parameter lists.

**Optional Macro Parameter Class**

We first define `ExtendedMacroParameter` to serve as a wrapper for `MacroParameter` and then define `OptionalMacroParameter` as a child of this class. This extra layer in the hierarchy is intended to allow easy implementation of other shorthands such as multiple occurrences of an argument. This implementations of the base class and the optional macro parameter class are provided in Code Example 8.11.
Extended Macro Parameter Class

Unfortunately, the extended parameter list cannot be automatically constructed using the definition from subsection 8.7.2.2. The class ExtendedMacroParameters is responsible for combining lists of normal and extended macro parameters so it requires more constructors than the automatic list generation shorthand provides.

Code Example 8.12 contains a partial implementation of the list with the constructors omitted for simplicity. The macros definitions provided ensure that any occurrences of extended macro parameters within a macro declaration (regardless of where in the list they appear) results in creation of an ExtendedMacroParameters object.

Extended Macro Definitions

Code Example 8.13 describes the basic structure of the macro for handling optional arguments. The macro enumerates the optional components producing multiple similar macros as described in subsection 5.3.4. Each occurrence of an optional parameter doubles the required number of expanded definitions.
8.7.2.4 Statically Type-checked Parameters

As demonstrated in subsection 4.8.4.2, specialisation macros are useful to provide specialised code for different static-types. In Genesis it is possible to define multiple macros with the same arguments but to reject expansion (by throwing an exception) if the compile-time static type is inappropriate (see subsection 5.5.6).

Such macros must be delayed until type information is available, include a suitable exception in their throws clause, and provide an explicit type check. The static type-checked parameter macro allows for the production of such to be automated.

This extension provides a further extension to macro formal parameters. An abstract syntax type may specify a static type restriction with the following syntax:

AbstractSyntaxType:StaticType Identifier. Code Example 8.14 demonstrates the use of this syntax for a factorial function specialisation for literal integers.

Code Example 8.13: Implementation Outline of Macro Definitions With Optional Arguments

```java
macro ClassMemberDeclarations
(macro, Type t, (, ExtendedMacroParameters ps, ), Block b) {
// a list of declaration replace this single macro definition
ClassMemberDeclarations ds;

// create a static method that contains the body of the macro definition
MethodDeclaration md;

// generate all permutations of options and add this to the list also
// each of these definitions has a call to md as its body
return ds;
}
```

(a) Definition Using Statically Type-checked Parameters

```java
macro LiteralInteger (Expression:LiteralInteger e, !) {
// calculate a compile-time factorial
}
```

(b) Expansion of Statically Type-checked Parameters

```
delayed macro LiteralInteger (Expression e, !) throws ConditionsNotMet {
if (!e.type().equals(LiteralInteger.class)) throw new ConditionsNotMet();
// calculate a compile-time factorial
}
```

Code Example 8.14: Factorial Literal Specialisation

Multiple static-type checks within a single definition are expanded and checked in the order they appear within the macro definition.
Statically type checked parameters are demonstrated more extensively in the implementation of iteration (i.e. `forall`) in subsection 9.2.2.
Analysis and Comparison

1. Introduction
2. Defining extensibility
3. Assessing extensibility
4. Reviewing extensibility
5. Designing the language
6. Reviewing parsing
7. Implementing a parser
8. Language implementation
9. Evaluation
10. Conclusion
9.1 Overview

The successfulness of the Genesis language definition (as defined in chapter 5) and its implementation (from the previous chapter) is assessed by a variety of methods: implementation of benchmark test cases, qualitative assessment, and a comparative assessment with Maya.

Firstly, in section 9.2, the power and flexibility of Genesis is shown with a proof-by-implementation of the benchmark test cases from section 3.4. Pertinent details of the implementations are provided, with much code omitted for the later complex examples. A review of these implementations is provided in section 9.2.7.

Section 9.3 contains a qualitative assessment of Genesis with a general discussion of issues relating to its power, usability, and error handling. Also, Genesis is rated against the criteria developed in section 3.3.

A detailed comparison of Genesis and Maya is provided in section 9.4 on issues such as: implementation of the benchmark test cases (and also MultiJava), length of code, and the criteria for extensible languages.

In section 9.5, the Graph Expansion Parsing method is compared against other general parsing methods by discussion of the class of grammars they accept and both time and space efficiency. Such analysis is sufficient at this stage of development, as algorithmic complexity improvement was the only form of optimisation applied to Graph Expansion Parsing or the Genesis system as a whole. Currently a direct speed comparison to other Java parsers would be strongly biased in favour of production compilers.
9.2 Implementation of Test Cases

In this section we provide implementations of the test cases from section 3.4. While explanations are provided for all of the approaches taken for translation, exacting code is omitted where it would add little to the explanation but a great deal to the length of this section.

Demonstrations of the differences between Genesis’ coding styles are provided for assertions and iteration. In particular, the improvement in code readability and conciseness is highlighted. Later examples use only the improved techniques.

Subsection 9.2.7 provides a review of the successfulness of the following implementations, with more general issues of Genesis’ code quality discussed further in section 9.3.

9.2.1 Assertions

The addition of assertions is the simplest of all of the test cases. Java 1.4 already provided for an assert statement that throws an AssertionError (a descendant of java.lang.Error). The following implementation reuses AssertionError for simplicity, although it would be a trivial matter to provide our own error class.

9.2.1.1 Basic Implementation

In Code Example 9.2, we provide the implementation without use of quasi-quotation. The translation of an assertion merely checks the condition, and upon failure outputs the offending expression to the standard error stream, and throws an AssertionError.

```java
public class Assertions {
  macro Statement (assert, Expression e) {
    Block b = new Block();
    b.add(new ExpressionStatement(
      new MethodCall(new Simple(new Name("System").add("err").add("println")),
        new LiteralString("Assertion failed: " + e.toString()))));
    b.add(new Throw(
      new Creation(new Type("AssertionError"),
        new List(new LiteralString("Assertion Failed"))));
    return new Block().add(new IfThenElse(e, null, b));
  }
}
```

Code Example 9.1: Basic Assertion Implementation
9.2.1.2 Quasi-quote Implementation

It should be clear from the definition in Code Example 9.2 that the use of quasi-quotation improves the readability of the implementation. The contents of the quasi-quotation closely matches the code that will be produced after expansion which is a great improvement over Code Example 9.2.

```java
macro Statement (assert, Expression e) {
  return {
    if (!e) {
      System.err.println("Assertion Failed: " +
        (new StringLiteral(e.toString())));
      throw new AssertionError("Assertion Failed");
    }
  };
}
```

Code Example 9.2: Quasi-quote Assertion Implementation

9.2.1.3 Implementation Issues

This version of assertions does not explicitly check the type of the expression given to the `assert` statement, it would be possible to do this, but for simplicity it was chosen to let the later stages of compilation pick up this error, as it will cause a type error for the generated `if` statement. See section 8.5.4 for a discussion of error handling.

9.2.2 Iteration

To illustrate the improvement in code, both in simplicity and conciseness, we provide multiple definitions of iteration constructs. Successive re-implementations demonstrate quasi-quotation, hygiene, and static-type matching.

9.2.2.1 Basic Implementation

Code Example 9.3 is the most low-level implementation of the `foreach` macro. This implementation has two `forall` macro definitions; this is to allow for optional brackets around the formal parameter argument. These brackets are merely syntactic sugar and play no part in the expansion. Unfortunately the optional brackets cannot take advantage of the optional parameter macro (see subsection 8.7.2.3) as they must either both be present or absent.
CHAPTER 9: ANALYSIS AND COMPARISON

IMPLEMENTATION OF TEST CASES

Code Example 9.3: Basic Iteration Implementation

9.2.2.2 Quasi-quote Implementation

In Code Example 9.4 we rewrite the `forall` method using quasi-quotation and unquote. The use of quasi-quotation provides a readable version of the `for` loop expansion for iteration. This still requires the explicit creation of a unique identifier for use within the quasi-quotation.

Code Example 9.4: Quasi-quote Iteration Implementation
9.2.2.3 Hygienic Implementation

Code Example 9.5 makes use of hygiene. Each variable declaration within a quasi-quotation is replaced with a unique variable name. Effectively, producing the version from the previous subsection.

```
static For forall(FormalParameter p, Expression e, Statement b) throws TypeMismatch {
    if (!e.type().isSubType(Iterator.class)) throw new TypeMismatch();
    return {{
        for (Iterator i = (`e).iterator(); i.hasNext(); ) {
            `(p.type()) `(p.getIdentifier()) = (`(p.type())) i.next();
            `b
        }
    }};
}
```

Code Example 9.5: Hygienic Iteration Implementation

9.2.2.4 Static-type Matching Implementation

The final implementation in Code Example 9.6 of the `forall` method demonstrates the static-type matching shorthand defined in section 8.7.2.4.

```
static For forall(FormalParameter p, Expression:Iterator e, Statement b) {
    return {{
        for (Iterator i = (`e).iterator(); i.hasNext(); ) {
            `(p.type()) `(p.getIdentifier()) = (`(p.type())) i.next();
            `b
        }
    }};
}
```

Code Example 9.6: Static-Type Matching Iteration Implementation

Note that this use of the static-type matching shorthand is on a static function, not a macro, but it works equally well on both. Indeed, had we chosen to only provide one syntax for this extension, we could have used this static-type matching and it would have added the delayed modifier to the macro.

This implementation removes the necessity to explicitly declare that this macro may throw an exception and removes the explicit check. This final version provides a very concise, readable definition of the `forall` macro.

9.2.3 Type-safe Formatted Output

The implementation of the `printf` macro is the most complicated of the simple test cases, as it requires us to generate an arbitrary amount of new code. Its implementation is not as straightforward as the previous cases, and doesn’t so closely match the code it
produces. It illustrates the benefits of having the full language at our disposal for the implementation of macros.

The code in Code Example 9.7 demonstrates the implementation of the printf macro with some parts abbreviated and some auxiliary functions omitted. The omitted function `match` breaks the string literal into its component parts, using `%` followed by a character as a separator. The definition of the omitted exception classes `TooManyActualParameters` and `TooManyPlaceHolders` are unremarkable.

The basic idea here is to break the string up into components, and to simultaneously iterate through these components and the list of supplied arguments. An expression that concatenates a list of strings is the result of this iteration. Upon detection of a placeholder from the literal string argument, the corresponding argument is verified to be of a suitable type, and is added to the expression. Any unrecognised placeholders from the literal string argument are treated as strings, and all these and all actual string components are added as is to the expression.

If there are either too many placeholders, or too many arguments, a corresponding exception is thrown to alert the user.

This code makes liberal use of quasi-quotation, but the code is only slightly more concise because of this. The code generation here is quite complex, and as a result the expansion code doesn’t resemble the resultant expansion.
```java
delayed ExpressionStatement (printf, (, LiteralString s, ",", Arguments list, ))
throws TypeMismatch, TooManyActualParameters, TooManyPlaceHolders {
    Expression exp = new LiteralString("\n");
    Vector parts = match(s.s, "%\n");
    Iterator i = list.iterator();
    Iterator j = parts.iterator();
    while (j.hasNext() && i.hasNext()) {
        Expression e = (Expression) i.next();
        Type t = e.type();
        // if t cannot be typed, throw syntax error – not done here for simplicity
        boolean keepLooking = true;
        while (keepLooking) {
            String placeHolder = (String) j.next();
            keepLooking = false;
            if (placeHolder.equals("%s") {  
                if (t.equals(String.class)) {  
                    exp = {{ \n + e }};
                } else {  
                    throw new TypeMismatch( "TYPE MISMATCH: string expected");  
                }
            } else if (placeHolder.equals("%d") {  
                if (t.equals(int.class)) {  
                    exp = {{ \n + (e) }};
                } else if (Integer.class)) {  
                    exp = {{ \n + e }};
                } else {  
                    throw new TypeMismatch("TYPE MISMATCH: integer expected");  
                }
            } else if (placeHolder.equals("%f") {  
                // similar to %d
            } else if (placeHolder.equals("%c") {  
                // similar to %d
            } else {  
                exp = {{ `exp + `\n LiteralString(placeHolder) }};
                keepLooking = true;
            }
        }
        if (i.hasNext()) {  
            throw new TooManyActualParameters("TOO MANY ACTUAL PARAMETERS: " + matches(s.s, "%\n") + " expected, " + list.size() + " found.");
        }
    }
    while (j.hasNext()) {
        String placeHolder = (String) j.next();
        if (placeHolder.equals("%s") || placeHolder.equals("%d") || placeHolder.equals("%f") || placeHolder.equals("%c") {  
            throw new TooManyPlaceHolders("TOO MANY PLACEHOLDERS: " + list.size() + " expected, " + matches(s.s, "%\n") + " found.");
        } else {  
            exp = {{ \n + \n LiteralString(placeHolder) }};
        }
    }
    return {{ System.out.println(`exp); }};
}
```

Code Example 9.7: Partial Type-safe Formatted Output Implementation
9.2.3.1 Example Expansion

Code Example 9.8 demonstrates an example printf usage and its expansion.

\[
\text{printf("hello %s, %d.", "world", 42);}\\
\]

(a) printf Macro Use

\[
\text{System.out.println("" + "hello " + "world" + ", " + (42) + ".");}\\
\]

(b) After Macro Expansion

Code Example 9.8: Type-safe Formatted Output Expansion

9.2.4 SQL Subset

The implementation of the SQL subset requires that we first match SQL syntax and then communicate with a database via the standard Java libraries. At first glance this appears to be more complicated than hand-written SQL strings. It is worth reiterating that the improvement here is that the run-time SQL is now guaranteed to be correct. In this research implementation the SQL is still re-checked for correctness at compile-time as implementing database classes from scratch was not the intent of the example.

Code Example 9.9 contains the main SQL abstract syntax class. All other classes used in the SQL implementation are omitted as these are essentially just construction classes and their implementation is straightforward (although they do define a method toExpression which is detailed in a moment). The macros that perform this construction are shown on lines 32–45. Many of these can utilise the macroSyntax shortcut, whereas the others require their terminal arguments for construction to take place (perhaps some extension in future work could better address these kinds of constructions).

The macro for SQL selection appears form lines 16–29 and performs a simple translation from the abstract syntax into run-time code by use of a common method called toExpression and a run-time call to a static method that performs the database query. The toExpression method operates as would be expected and produces an expression to reproduce the SQL query. Generally this consists of a sequence of literal strings, but Java expressions within the SQL must be preserved.
The user is required to initialise this SQL class with a valid connection by using the static `init` method on line 3 and this connection is used within the implementation of the `select` method (lines 5–14). This method performs the SQL query and is generally unremarkable. The final result array is examined and either a single value or a vector of values is returned to the caller — this code is omitted for increased clarity of the important points.

The only point to stress in this straightforward implementation occurs on line 28. The use of `select` within the quasi-quotation is an unbound reference and when translation takes place it is correctly bound to the static method with the SQL class. It is effectively
replaced with a call to `any.earlier.path.SQL.select`. Such an approach ensures that referential transparency is maintained.

### 9.2.5 Generators

The implementation of generators follows a similar approach to that of the C++ macro implementation covered in section 2.3.3.3, with the major complication being the replacement of the C++ switch statement with a cleaner version in Java. To aid in the implementation, we first introduce a helper class (see Code Example 9.10) that every generator will extend. Each generator implementation need only implement the abstract method `hasNext`, and everything else will work as a result.

```java
abstract class GeneratorBase implements Iterator {
    private Object nextVal;
    protected int reentry = 0;

    protected boolean suspend(Object n, int r) {
        nextVal = n;
        reentry = r;
        return true;
    }

    protected int position() { return reentry; }

    abstract public boolean hasNext();

    public Object next() { return nextVal; }

    public void remove() { throw new UnsupportedOperationException(); }
}
```

**Code Example 9.10: Generator Helper Class**

The `GeneratorBase` class extends `Iterator`, and as we will see, with some clever implementation this allows us to get the use of the `forall` macro for free. Code Example 9.11 shows the implementation of the `suspend` statement.

```java
class Suspend implements Statement {
    Expression expr;
    Suspend(Expression e) { expr = e; }
}

macro Statement (suspend, Expression e) {
    return new Suspend(e);
}
```

**Code Example 9.11: Suspend Statement Implementation**

If only everything was this easy! This translation merely adds a place holder for later erasure. The macro definition in Code Example 9.12 provides the implementation details for actual expansion of a generator method, some details have been omitted, but are covered in the following subsections.
Each generator translates into a method of the same name, whose body merely instantiates a copy of an inner class\textsuperscript{12} that extends GeneratorBase. This inner class contains the real implementation, but wrapping in a method allows for there to be no requirement for any implementation on the calling side; use of generators does not even need to be detected.

The inner class requires that each generator formal parameter be translated into a field declaration as local variables cannot retain their values upon generator resumption. In addition to this, each of these variables must be initialised via a standard constructor that mimics the original generator definition. Any local variable declarations within the generator body must also be translated to field declarations. These two translations are relatively straightforward, and are covered in the following two subsections.

\textsuperscript{12} An anonymous class was considered, but rejected, due to complicated initialisation code.
The final translation requirement is to transform the generator body into a number of cases (corresponding to a base case, and one case for each occurrence of `suspend`) for use within the `hasNext` method. This implementation is difficult, and is covered in subsection 9.2.5.3.

Genesis’ ability to mix abstract and concrete syntax within quasi-quotations is equivalent to that of MS² (see subsection 4.4.4.2). This is an extremely concise and natural programming technique that abstracts away many of the tedious details of concrete syntax.

### 9.2.5.1 Translation of Formal Parameters

As shown in Code Example 9.13, the translation of the formal parameters is trivial. One field declaration, and one initialisation must be created for each formal parameter. The only thing to note here is the explicit creation of a `FieldDeclaration` class was used simply because it was more readable than the quasi-quote alternative.

```java
forall (FormalParameter p) in params {
    declarations.add(new FieldDeclaration(p.type, p.name));
    initialisation.add( {{ this. (p.name) = (p.name); }}
}
```

**Code Example 9.13: Translation of Formal Parameters**

### 9.2.5.2 Translation of Local Variable Declarations

The translation of the local variable declarations requires us to examine the entire tree representing the method body, and to transform each detected local variable declaration into a corresponding field declaration. For local variable declarations that contain initialisers, an assignment must be generated in order to produce the same semantics.

```java
/* perform a recursive-style tree-walk through the block, making changes each time a local variable declaration is discovered */
LocalVariableDeclaration d = /* current local variable declaration */;
declarations.add(new FieldDeclaration(d.type, d.name));
i.set( (d.initialiser == null) ? {{ null; }} :
    {{ (d.name) = (d.initialiser); }} );
```

**Code Example 9.14: Translation of Local Variable Declarations**

The outline of the code to perform these actions is shown in Code Example 9.14. The inspection of the statement tree is accomplished by use of tree-walk through the block. Such code is relatively straightforward to write, but cumbersome, therefore it is omitted as it adds little to the understanding of the process.
9.2.5.3 Generation of Resumable Code

The C++ macro implementation of generators relied upon an unusual usage of the C++
\texttt{switch} statement (see subsection 2.3.3.3). Java's \texttt{switch} statement does not allow
equivalent usage, so a different method must be employed.

The method followed is described in [LM02] and essentially produces a continuation for
each location that execution may (re)commence in the body of the generator. Each
continuation must employ statement reducibility analysis [GJSB00§14.19] to ensure
successful Java compilation.

9.2.5.4 Example Expansions

Code Example 9.15 demonstrates the definition and expansion of the simplest example
of a non-terminating generator, it merely repeats its argument forever.

```java
generator static String repeat(String s) {
    while (true) suspend s;
}
```

\textbf{(a) Before Translation}

```java
static Generator repeat(String s) {
class repeat extends GeneratorBase {
    String s;

    public repeat(String s) { this.s = s; }

    public boolean hasNext() {
        switch (position()) {
          case 0:
            while (true) return suspend(s, 1);
          case 1:
            // nothing...
            while (true) return suspend(s, 1);
        }
        return false;
    }

    return new repeat(s);
}
```

\textbf{(b) After Translation}

\textbf{Code Example 9.15: Repeating Generator Expansion}

We can observe the duplication of all parameters to the generator as field declarations,
and the single constructor mimics the surrounding method call. This generator has no
local variables, so there are no extra field declarations. The reuse of names (eg.
\texttt{repeat} for both the method and the inner class) makes these translations a little harder
to comprehend, but is valid Java, and removes the necessity of new name generation.
The generation of the resumable cases is trivial, the base case is the whole body, and so is the single resumption. It should be clear that the 

`hasNext` method could be greatly simplified: the two resumptions are identical, making the `switch` statement redundant; the `while` loops are redundant; and the final `return` statement is unreachable. These kind of optimisations are beyond the scope of this extension at this time.

Code Example 9.16 contains the definition and translation of a Fibonacci generator and demonstrates slightly more difficult to translate resumptions and local variable use.

This example uses `Integer` rather than `int` as the current implementation only supports class types (see subsection 9.2.5.6). Again, the translation (shown in Code Example 9.16) results in code much more verbose than the original generator. In this translation, the local variables `x` and `y` have been recreated as field declarations, and their initialisations are translated into assignments.

We have three resumptions, a base case, and one each for the two `suspend` statements.

- The base case contains code up to the first `suspend` statement, and all following code is pruned as it is unreachable.
- The second case begins from the statement immediately following the first `suspend`, and stops upon the encountering of the second. Notice here that the `while` statement does not appear at all as the condition can never be checked on this resumption.
- The third, and final, case begins from the statement immediately following the second `suspend`, and then must contain the entire `while` statement again (albeit in a pruned form).
generator static Integer fib() {
    int x = 0, y = 1;
    while (true) {
        suspend new Integer(y);
        x = x + y;
        suspend new Integer(x);
        y = x + y;
    }
}

static Generator fib() {
    class fib extends Generator {
        int x;
        int y;

        public fib() {
        }

        public boolean hasNext() {
            switch (position()) {
                case 0:
                    x = 0;
                    y = 1;
                    while (true) {
                        return suspend(new Integer(y), 1);
                    }
                case 1:
                    x = x + y;
                    return suspend(new Integer(x), 2);
                case 2:
                    y = x + y;
                    while (true) {
                        return suspend(new Integer(y), 1);
                    }
            }
            return false;
        }
        return new fib();
    }
}

(a) Before Translation

static Generator fib() {
    class fib extends Generator {
        int x;
        int y;

        public fib() {
        }

        public boolean hasNext() {
            switch (position()) {
                case 0:
                    x = 0;
                    y = 1;
                    while (true) {
                        return suspend(new Integer(y), 1);
                    }
                case 1:
                    x = x + y;
                    return suspend(new Integer(x), 2);
                case 2:
                    y = x + y;
                    while (true) {
                        return suspend(new Integer(y), 1);
                    }
            }
            return false;
        }
        return new fib();
    }
}

(b) After Translation

Code Example 9.16: Fibonacci Generator Expansion

9.2.5.5 Explicit Use of GeneratorBase

The GeneratorBase class is exposed, so users are free to use it in other code, or even as part of the definition of other generators. Code Example 9.17 demonstrates the definition, translation, and usage of a take function that generates the specified number of elements from another generator. This allows forms such as that in Code Example 9.17(c) that outputs the first twenty Fibonacci numbers from the infinite sequence Fibonacci generator from Code Example 9.16.
The definition of `take` uses `GeneratorBase` explicitly as a parameter, and uses standard iteration code to select only the first `n` elements. The translation follows the same approach as other generators.

```java
static Object take(int n, GeneratorBase g) {
    while ((n-- > 0) && (g.hasNext()) suspend(g.next());
}
```

(a) Before Translation

```java
static Generator take(int n, GeneratorBase g) {
    class take extends Generator {
        int n;
        GeneratorBase g;
        public take(int n, GeneratorBase g) {
            this.n = n;
            this.g = g;
        }
        public boolean hasNext() {
            switch (position());
            case 0:
                while ((n-- > 0) && (g.hasNext())) return suspend(g.next(), 1);
            case 1:
                // nothing...
                while ((n-- > 0) && (g.hasNext())) return suspend(g.next(), 1);
                return false;
        }
        return new take(n, g);
    }
    return new take(n, g);
}
```

(b) After Translation

```java
forall (Integer i) in take(20, fib()) { System.out.print(" "+ i); }
```

(c) Example Usage

**Code Example 9.17: Sub-sequence Generator Expansion and Use**

A fully worked generators implementation would provide this functionality as a method of `GeneratorBase` so that expressions such as `fib().take(20)` can be written.

### 9.2.5.6 Implementation Issues

The implementation for generators only supports `Object`, and so manual boxing is required for primitive types.

Translation of local array variables with array intialiser lists are currently not handled correctly; they require a slightly more complicated translation.
9.2.6 Haskell Subset

Each of the abstract syntax classes used to implement Haskell subset grammar (see Figure 3.2) implements the FunObject interface (as shown in Code Example 9.18), which defines three methods: createSelf, funType, and eval.

```
interface FunObject extends AbstractSyntax {
    Creation createSelf();
    FunType funType() throws TypeMismatch;
    FunObject eval(BindingList bindings);
}
```

Code Example 9.18: FunObject Interface

The child classes of FunObject serve a variety of purposes, they are used:

- to drive the parse at compile-time and provide syntax checking;
- for compile-time type-checking;
- at run-time to represent the structure of the functional program; and
- for run-time lazy evaluation.

Each of the child classes of FunObject are implemented with construction macros. The compile-time implementation of these classes for parsing is unremarkable (and hence not shown), but their other uses are interesting.

Calling the createSelf method on a functional object will produce a creation expression that will reproduce the current datastructure fully. This is used at compile-time to create run-time code that will store the functional objects (see section 9.2.6.1).

Calling the eval method performs the lazy evaluation at run-time (see section 9.2.6.3).

9.2.6.1 Construction

Creation of code that will, at run-time, produce the required Haskell subset objects is handled by the createSelf method. This method produces a creation expression that reproduces the current object. An example of the straightforward implementation of such a method is provided for the FunCons class in Code Example 9.19.

The run-time objects created by the code that createSelf produces are used in the interpreted evaluation of Haskell code (see subsection 9.2.6.3).
9.2.6.2 Type Abstract Syntax Classes

The abstract syntax classes used for the Haskell subset type system serve two purposes:

- parsing of type signatures; and
- compile-time type-checking.

Four classes are defined, corresponding to the four basic types described in section 3.4.2.3. Each of these classes inherit from the FunType base class and must implement the equals method. This equals method is used in type-checking operations as demonstrated in 9.2.6.4.

The equals function is calculated according to the following rules:

- if both arguments are of type FunArbitrary, the result is true if their ident fields are equal;
- if one argument is of type FunArbitrary, the result is true;
- if the arguments are not of the same type (and neither is of type FunArbitrary) the result is false;
- if both arguments are of type FunTypeFun, the result is true if both their left fields have the same type and their right fields have the same type;
- if both arguments are of type FunTypeList, the result is true if their element fields have the same type; and
- if both arguments are of type FunTypeInt, the result is true.

Code Example 9.19: Haskell Subset Cons Abstract Syntax Class
abstract class FunType implements AbstractSyntax {
    abstract boolean equals(FunType t);
    macroSyntax FunTypeInt (int);
    macroSyntax FunTypeList ("[", FunType t, "]");
    macroSyntax FunTypeFun ("(" , FunType t, ")", FunType u, ")");
    macro FunTypeArbitrary (Token t) throws ConditionsNotMet {
        if (t.value.length() != 1 || !Character.isUpperCase(t.value.charAt(0)))
            throw new ConditionsNotMet();
        return new FunTypeArbitrary(t.value.charAt(0));
    }
    class FunTypeInt extends FunType {
    }
    class FunTypeList extends FunType {
        FunType element;
    }
    class FunTypeFun extends FunType {
        FunType left;
        FunType right;
    }
    class FunTypeArbitrary extends FunType {
        char ident;
    }
}

Code Example 9.20: Type Abstract Syntax Classes for the Haskell Subset

The rules for equality of functional types are summarised in Table 9.1.

Table 9.1: Rules for Function Type Equality

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
<th>List</th>
<th>Function</th>
<th>Arbitrary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>List</td>
<td>false</td>
<td>equal elements</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>Function</td>
<td>false</td>
<td>false</td>
<td>equal left and right</td>
<td>true</td>
</tr>
<tr>
<td>Arbitrary</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>equal identifiers</td>
</tr>
</tbody>
</table>
9.2.6.3 Evaluation

The run-time evaluation of functional constructs is performed by an interpreter. This choice is for simplicity in this proof-of-concept implementation and does not imply that more efficient techniques are not applicable.

Evaluation is provided through the `eval` method defined for all functional objects. In addition to the evaluation function provided for `FunObjects`, a similar function is provided for functional operators and the `BindingList` class.

For the majority of the child classes of `FunObject`, evaluation implementation is straightforward:

- the evaluation function for `let` expressions merely introduces a new binding — which may override other bindings previously in scope;
- the evaluation function for infix operators evaluates both left and right arguments — no lazy evaluation is provided for the standard arithmetic operators;
- the evaluation function for identifiers merely finds the binding that is currently in scope for the identifier; and
- the evaluation functions for lambda functions, literals, nil lists, and lists created with `cons` simply return the current object — these are, for the most part, treated as atomic elements.

There is one exception however to this treatment of atomic elements: lists created with `cons` can be broken apart by use of the special functions `head` and `tail`. These functions are treated as a special case of the application of functions. The implementation of application is shown in Code Example 9.21.
Normal function application will take place when the left side of a function call evaluates to a lambda function. Even if the function is specified by name, the name’s binding evaluates to a lambda function. The application itself merely introduces a new binding form with a unique name. If the left side evaluates to `head` (or `tail`), the argument is evaluated as far as a cons, and the head (or tail) of the list is evaluated and returned.

Lazy evaluation is guided only by the `if` construct which is the only conditional component of the Haskell subset. Once the evaluation proceeds to a point where an `if` expression is the next thing to be evaluated, the Boolean condition (and any associated work) must be evaluated. This is handled in the expected way as shown in Code Example 9.22.

Code Example 9.21: Evaluation of Function Application
9.2.6.4 Type Checking

Type checking occurs once the entire embedded Haskell block is parsed. At this stage all Haskell declarations are known so type checking can proceed. This is still performed by non-delayed macros as the surrounding type information is not required.

Each functional declaration is checked against its type signature in turn. Both the type signature and expression abstract syntax classes contain the `funType` method so it is a simple matter to check that the two types match — although the polymorphic type `*` complicates this checking a little.

9.2.6.5 Embedded Usage

In addition to the Haskell classes of the previous subsections, it is still necessary to add macros defining the two wrappers of Figure 3.3. Code Example 9.23 outlines the implementation of both the Haskell definition wrapper and the embedded Haskell function call wrapper.

Any use of the Haskell function call wrapper is simply translated into a standard Java method call to a static evaluation method of a static inner class — the use of this inner class side-steps hygiene as it is not declared within the quasi-quotation.

The inner class is created by the Haskell definition wrapper. The evaluation function simply calls the `eval` method on its provided argument (an arbitrarily complex Haskell expression) and with the entire set of Haskell definitions considered to be its initial list of bindings.
Chapter 9: Analysis and Comparison

Implementation of Test Cases

Code Example 9.23: Embedded Haskell Wrapper Definition

The Haskell definition wrapper is also responsible for type-checking all the Haskell declarations. This is straightforward and consists of little more than checking the type of the signature to the type of the Haskell expression — a process already defined with the use of the `funType` method from subsection 9.2.6.4.

In order to use the embedded Haskell subset both `Haskell` and `HaskellWrappers` must be imported. The subset usage is split into two files so that the subset can be used as a embedded fashion and also in a standalone fashion.

9.2.6.6 Standalone Usage

The Haskell subset can be used without any Java code appearing in the source file at all with the use of the `importonly` switch (as described in 8.6.1.2). This switch is used to import only two classes: `Haskell` and `HaskellStandalone`.

Use of Haskell in standalone mode requires the specification of a module name within the source file — this is used to name the resultant Java translation class. An example of this syntax is shown in Code Example 9.24. This example uses the extended forms defined in the next subsection.

```
module HaskellTest where
main :: int -> [int]
main x = [ a * a | a <- range 1 x ]
range a b = if (a <= b) then (a : range (a+1) b) else []
```

Code Example 9.24: Standalone Haskell Module
Each standalone Haskell file must also declare the function `main`. The definition of this function is permitted to contain an arbitrary number of integer arguments that can be specified from the command line.

The HaskellStandalone class functions much as the wrapper for macro definitions in that it collects all of Haskell subset declarations and checks that each function matches its signature. However it produces a class that has an automatically generated Java main function that will accept arguments from the console, perform a functional calculation, and finally output results to the console.

```java
public static void main(String[] args) {
    if (args.length != 1) return;
    System.out.println(
        new FunApply(new FunExpr("main"), new FunLiteral(args[0])).eval()
    );
}
```

**Code Example 9.25: Standalone Haskell main Method**

### 9.2.6.7 Extended Forms

The Haskell implementation contains a few extra forms that are not specified in the subset. We first define quasi-quotation for the Haskell subset to aid in the following definitions.

Using this extended quasi-quotation facility, illustrative examples of the power of Genesis are provided for function declarations, operator currying, and simple list comprehensions. The techniques shown here could easily be used to add further functionality such as: `where` clauses, pattern matching, and type classes.

**Quasi-quotation**

As discussed in subsection 8.7.1, the quasi-quotation macros automatically handle new additions to the abstract syntax. However, we must manually provide unquoting for abstract syntax classes that we wish to use in such a fashion. For the Haskell subset this entails creating unquote definitions for declarations through to expressions.

**Function Declarations**

Haskell provides a syntactic sugar for function declarations that doesn’t require the use of lambda functions. For example, instead of \( f = \lambda x \rightarrow y \rightarrow x + y \) we could write \( f \ x \ y = x + y \). Using our newly defined quasi-quotation definition, we can provide this extension to our Haskell subset by using a mixture of Haskell and Genesis forms.
CHAPTER 9: ANALYSIS AND COMPARISON  
IMPLEMENTATION OF TEST CASES

Code Example 9.26: Function Declarations Definition

Code Example 9.26 contains the definition of function declarations. It simply constructs lambda functions from the identifiers in argument list (in reverse). Genesis code for creating the list of functional identifiers is omitted.

**Operator Currying**

Haskell allows binary operator application to omit either parameter to provide a partial application. For example the expression \((1+)\) returns a function that adds one to its argument. With the use of quasi-quotation, Code Example 9.27 shows the simplicity of adding this to Haskell subset.

Code Example 9.27: Operator Currying Definition

**Simple Single Source List Comprehensions**

Simple single source list comprehensions can be provided by translation into use of the \(\text{map}\) function. An example of this for a simple fragment is shown in Code Example 9.28(a). This translation is so simple it can be provided in a single line as shown in Code Example 9.28(b).

Code Example 9.28: Simple Single Source List Comprehensions
Multiple Condition Single Source List Comprehensions

It is not much more complicated to provide list comprehensions that while still drawing from a single list source have a number of predicate conditions. The translation here requires that the standard function filter be applied to the list once for each specified condition (although this could also be done with one pass by combining all the conditions into one expression).

Code Example 9.29 demonstrates this translation. It is very similar in construction to the function declaration example.

```perl
macro FunExpr ([, FunExpr e, |, FunIdentifier i, <-, FunExpr f, ",", FunExprList ps, ]}) { // ps was comma seperated
    FunExpr filteredList = f;
    forall forall forall forall (FunExpr p) in in in in ps {
        filteredList = {{ filter (\`i -> `p) `filteredList }};
    }
    return {{ map (\`i -> `e) `filteredList }];
}
```

Code Example 9.29: Multiple Condition Single Source List Comprehensions

9.2.7 Implementation Review

The macro definitions for assertions and iterations demonstrated how the use of shorthands such as quasi-quotation and static-type matching greatly simplified definitions and improved their readability. Comparison of such techniques are not addressed here but are delayed to later sections.

Table 9.2 assesses the successfulness of the final implementation of each of the test cases.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 assert</td>
<td>Assertions have the simplest definition of all of the benchmark test cases as no inspection of arguments is required and the translation is uniform for all arguments. A simple and easily understandable definition is possible with Genesis’ quasi-quotation facilities.</td>
</tr>
<tr>
<td>2 foreach</td>
<td>The definition of iteration is simple and highlights the</td>
</tr>
</tbody>
</table>
conciseness that can be gained from using hygiene and static-type matching shorthands.

However, it also illustrates a restriction of Genesis — namely that macros cannot be used within the file in which they are declared. For this definition it leads to the necessity of defining another method that both macro definitions call.

### 3 printf

The implementation of typesafe formatted output is unwieldy, but this is due to standard Java rather than a shortcoming in Genesis.

Quasi-quotation is used extensively to build an expression over time. The resulting expression is similar to what would be produced manually, but without the concatenation of successive string literals. However, the Java language definition requires that such strings are combined on compilation [GJSB00], so there is no penalty in run-time performance.

### 4 SQL

Genesis is easily capable of providing exact SQL syntax and disallows inclusion of syntactically incorrect forms. Genesis has no trouble providing a limited expression definition for sole use within SQL statements, despite the obvious overlap with standard Java expressions.

The SQL subset implementation highlights the frequency of occurrence of construction macros and the utility of the `macroSyntax` shorthand. Only once a full `SELECT` statement is recognised does any translation occur.

The actual translation is simply a production of an SQL string to pass to a standard run-time SQL system. Despite the end call being the same, it should be clear
that the advantage that Genesis provides is in early error detection. Such errors will be automatically handled by Genesis’ syntax error mechanisms.

### 5 Generators

The generators implementation shows the powerful nature of code translation possible with Genesis macros. Generator translation is not a simple pattern based translation (like the assertion and iteration macros), but rather a sophisticated code transformation.

Genesis provides quasi-quotation for the portions of this translation that can be generated through a pattern based approach, and allows easy splicing of constructed forms via unquoting.

The result is that the general form of the translation is easily definable (and hence understandable) and the more complicated parts can be dealt with in isolation. This greatly simplifies the construction of such code.

Genesis allows the generator primitives to be exposed to the user in such a way that allows for powerful combinations of techniques.

The major problem with the generators implementation is that its reliance on Java1.4 disallows simple creation of generators of primitive types. Java1.5 would both provide auto-boxing/unboxing and allow a cleaner generics based definition.

### 6 Haskell

Genesis handles the Haskell subset syntax with ease, despite its lack of similarity to standard Java syntax.

The Haskell subset implementation demonstrates the flexibility of Genesis abstract syntax types. Allowing such types to be created by the user as standard Java classes allows them to be used in a variety of powerful
ways at both compile-time and run-time.

In implementing the Haskell subset, the abstract syntax types are used to drive the parse, perform compile-time type-checking, to represent the run-time program, and for run-time evaluation. In fact, due to the use of an interpreter, very little actual translation is performed at compile-time.

By use of a combination of Genesis macros and quasi-quotation for Haskell it was a simple matter add additional Haskell constructs to the subset.

The full flexibility of Genesis is demonstrated by the standalone usage of the Haskell subset. If desired, Genesis can be truly syntax independent.
9.3 Qualitative Assessment

The qualitative assessment of Genesis begins with a general discussion of issues relating to its power, usability, and error handling (subsections 9.3.1, 9.3.2, and 9.3.3). Where appropriate, comparisons are drawn to the languages reviewed in chapter 4.

Subsection 9.3.4 contains an evaluation of Genesis in respect to the criteria for rating extensible languages from section 3.3.

In the following section (section 9.4), a detailed comparison of Genesis and Maya is provided.

9.3.1 Power

The arbitrary syntax creation facilities are the most impressive advantage of Genesis compared to other systems. Compared to the reviewed languages in chapter 4, Genesis provides the most flexible grammar construction facilities. Genesis’ expressive power is wide-ranging and allows for a host of sophisticated extensions – the implementations of the benchmark test cases and the standard library facilities nicely illustrate this (these are discussed in the following subsections).

Most of the reviewed extensible languages provide similar facilities for syntax interrogation via some form of abstract syntax classes and Genesis is no exception. All of these facilities are on a similar level of power. Complex hierarchies of abstract syntax classes (like those of Genesis) offer benefits over S-expressions or skeletal syntax trees at the cost of increased complexity.

Genesis provides access to static-type information via the use of delayed macros. These macros are expanded only after the entire parse has been successfully completed and typing issues can hence be resolved. This is not without cost to the simplicity of the system (this is further discussed in subsection 9.3.2). Delayed macros can be difficult to reason about and complicate final parse resolution but allow for some of the more sophisticated meta-programming techniques to be applied, such as specialisation and compile-time static-type checking and resultant error production. Even without delayed macros, Genesis provides superior power to the reviewed systems.
9.3.1.1 Benchmark Test Cases
As demonstrated in the previous section, Genesis is easily capable of providing an implementation for each of benchmark test cases.

Most notably, the specified syntax of the Haskell subset was easily created with Genesis’ powerful syntax creation facilities. However, an exact implementation of Haskell would not be possible as it uses a layout-based approach to scoping. Further tokeniser flexibility would be required to allow such a technique in Genesis (see subsection 10.2.1).

Beyond those requiring increased lexical flexibility it is unclear if there exists any real-world language constructs that Genesis is incapable of expressing.

9.3.1.2 Quasi-quotation Implementation
Genesis’ ability to provide a set of macros for quasi-quotation is a testament to its expressive power. Most other meta-programming systems are incapable of such a construction — with the most obvious notable counter-example being Lisp as its S-expressions lend themselves nicely to this kind of manipulation. Genesis’ more complicated set of abstract syntax classes are not as simple to use in this respect, but they provide other benefits such as guaranteed syntactic correctness.

However, the implementation of unquoting highlights the difficulty of working with a mixture of delayed and non-delayed macros. The number of possible parses examined increases considerably when using unquoting, as the static type of the expression being unquoted is not yet known. Here our choice of Graph Expansion Parsing lets us down a little as even though local variables declarations (and any previous instance variable declarations) have been speculatively parsed, there is no way to connect these to the speculative parse of the unquoting. Perhaps there are further parsers refinements that can smooth out these rough edges (see subsection 10.2.2).

9.3.1.3 Other Standard Library Macros
Genesis’ power allows the basic language definition to remain simple and for user shorthands and syntactic sugar to be provided by extensions. The standard library provides facilitates the creation of static-type matching macros (particularly useful for specialisation), simple construction macro generation, and optional macro parameters. These basic extensions demonstrate the beginnings of how the Genesis macro system
could build upon itself — much the same way as this takes places within the Lisp community. Most other meta-programming languages are too limited to provide such extensions. Although we have seen small extensions provided by Maya and JSE for automatic generation of macro support code, the possibility of further extensions is limited by their lack of power.

9.3.2 Usability

Genesis macro definitions are incredible simple, they closely mimic Java method calls, but do require their arguments to be comma separated — unlike Maya which infers which arguments are types and which are terminals. The Maya solution is more lightweight, but perhaps not as clear. JSE and MS² require many more symbols to appear in macro declarations for little, if any, extra benefit.

Genesis requires no understanding of parser in order to be able to write macros. Most other reviewed extensible systems either gave the impression that no knowledge was required, until the user caused a parser conflict, or had strict rules on the placement and structure of macros.

Like Template Haskell, Genesis does not regard macros and syntax classes as particularly special. There is little distinction between run-time and compile-time functions and abstract syntax classes are implemented as standard Java classes. Macros are provided in such a fashion that the Genesis compiler knows that they are to be interleaved with the grammar and executed at compile-time, and unlike Template Haskell, they have no requirement for explicit identification when being called.

The quasi-quotation macro uses the direct representation of Genesis macros in its implementation (albeit at compile-time) and run-time functions can access these functions in a similar fashion. The Haskell subset makes extensive use of its abstract syntax classes to perform a variety of tasks other than parsing at both compile-time and in its run-time system.

Genesis shares with MS² the ability to insert structures into quasi-quotes that are free of their original syntax — this was used extensively in the implementation of generators (see subsection 9.2.5) and is a powerful and concise tool.

Support for hygiene is provided through the quasi-quotation mechanism or by the use of explicit name generation.
9.3.3 Error Handling

The Genesis abstract syntax classes are standard Java classes and utilise Java’s type-checking mechanisms to ensure that all abstract syntax is correct.

Additionally, syntax errors within Genesis’ quasi-quotation facility are handled no differently than syntax error within Genesis code. No special checking is required within the quasi-quotation mechanism, if a syntax error occurs within a quasi-quote, it will not be successfully matched by the parser.

Genesis has strong support for explicit detection of compile-time errors. All macros can declare a series of possible exceptions to be thrown via a `throws` clause which is essentially equivalent to that for normal Java methods. Such exceptions are handled by the parser and can allow the macro to provide extra conditions for its matching or to provide compile-time warnings or errors.

Support for syntax errors detection during parsing is less comprehensive. Syntax errors are difficult to pinpoint with Graph Expansion Parsing. Detection of such errors (generally trivial for other parsers) is the weakest part of the Genesis implementation. For syntactically correct files, an attempt is made to report the source of errors once translation to standard Java has occurred. While the source of the error may be discovered, the exact point in the expansion in which the error occurred may be difficult to determine.

9.3.4 Extensibility Criteria Assessment

Table 9.3: Genesis Extensibility Criteria Assessment

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Syntax Creation</td>
<td>Genesis’ primary contribution is that of arbitrary syntax creation. It is unencumbered by its parser and allows virtually any syntactic construct to be created — from the smallest statement or expression level macro to language embeddings.</td>
</tr>
<tr>
<td>1.2 Syntax Interrogation</td>
<td>Like the majority of its counterparts, Genesis provides simple to use facilities for syntax interrogation via a set</td>
</tr>
</tbody>
</table>
1.3 Syntax Overloading

Genesis need make no special provision for syntax overloading as it has no concept of reserved words — support for syntax overloading is a nice side-effect of providing a general parsing scheme.

Any symbol or identifier can be used within a macro definition regardless of whether or not it is considered to have special meaning within a standard Java program.

In fact, through its concept of priorities, a Genesis programmer can even provide syntax replacements, rather than just overloading. It is possible to provide an exact replica of a standard syntactic construct, but with a higher priority. This allows users to modify the semantics of standard Java (or indeed any extension).

1.4 Static Type Interrogation

Genesis provides support for static-type interrogation via its abstract syntax classes and by allowing macros to throw compile-time handled exceptions. The explicit use of the `delayed` keyword is required for macros that wish to query static-types.

While Genesis’ approach is flexible, it puts the onus on the programmer when using static-type matching on macro definitions. For this reason the Genesis standard environment provides macros that allow such forms to be simply and conveniently expressed. Most situation use these standard macros and so explicit use of the `delayed` keyword is rare.

1.5 Expressiveness

Genesis is capable of expressing simple macros in a simple concise way, large scale language modifications, and it is even possible to replace the standard syntax entirely.
As evidenced by the implementation of the benchmark test cases, Genesis is capable of providing:

- small limited use syntax additions (e.g. assert, forall, printf, etc.);
- embeddings of domain-specific languages (e.g. SQL, Haskell, etc.); and
- an entirely new syntax without any reference to Java syntax at all.

### 2.1 Simplicity

Genesis macro definitions are as similar as possible to standard Java method declarations. The only restriction to macro definitions is that they must have at least one argument.

The requirement of the `delayed` keyword when using static-typing facilities adds to the general complexity of Genesis macro definitions. Whilst the use of standard environment macro definitions alleviates some of this complexity, the delayed keyword could still potentially be the source of much confusion.

By Genesis’ use of a general parser, users are able to write macros without any understanding of parser theory. There are no special cases that can cause confusion. Macros can be written with left- or right-recursion and can contain any symbols required.

However, by removing restrictions on the parser it is possible to create a macro (or worse, a set of macros) that can create ambiguous parses. Such poorly written macros could be a major source of confusion.

### 2.2 Brevity

As demonstrated in section 9.2.1.1 and 9.2.2.1, low-level Genesis code is cumbersome at best. This low-level code can still be useful in simple cases and can be freely mixed with more sophisticated techniques (as
shown in section Code Example 9.13).

Thankfully, Genesis is powerful enough to provide a quasi-quotation facility as a standard environment extension. Quasi-quotation code can be interleaved within normal Java code, and vice versa if unquoting is used.

Genesis code using quasi-quotation provides definitions that are similar in complexity to other extensible (or meta-programming) systems (see section 9.4.1.1).

2.3 Robustness

Genesis provides facilities for both explicit name clash avoidance and automatic hygiene.

Fresh name generation is provided in a form analogous to the traditional use of gensym in Lisp.

Automatic hygiene ensures that variables declared within quasi-quotations are freshly generated.

3.1 Syntactic Correctness

Genesis provides guarantees of syntactic correctness through its abstract syntax classes. Use of these classes requires the user to create correct syntax.

3.2 Error Detection

Genesis has strong support for detecting compile-time errors. All macros can declare a series of possible exceptions to be thrown via a throws clause which is essentially equivalent to that for normal Java methods.

These exceptions are handled by the parser and can allow the macro to provide extra conditions for its matching or to provide compile-time warnings or errors.

There are good facilities for automatic static-type checking and error reporting via the standard environment.
### 3.3 Error Reporting

Explicit checks allow macros to report warnings or errors.

An attempt is made to report the source of errors once translation to Java has occurred, but it is doubtful if the error messages will be of much use for more complicated macros.

Syntax errors during parsing with the Graph Expansion Parser are difficult to pinpoint and Genesis provides little to help the user in this regard. Detection of such errors (generally trivial for other parsers) is the weakest part of the Genesis implementation.
9.4 

Maya Comparison

Of the previous attempts at extensible languages covered in chapter 4, Maya (section 4.8) is the most directly comparable to Genesis. Both extend Java and allow arbitrary syntactic forms to be created whereas the other reviewed languages do not.

In subsection 9.4.1, a comparison is provided between the Genesis implementation of the benchmark test cases (from section 9.2) and the direct Maya implementation of the simple test cases (assert, foreach, and printf) and Maya’s capacity to implement the complex test cases (SQL, Generators, and Haskell).

Maya provides a MultiJava extension to Java as a proof-by-implementation. In subsection 9.4.2, the ability of Genesis to provide an equal implementation is compared with Maya’s implementation.

A comparative rating of Genesis and Maya on the extensibility criteria is provided in section 9.4.3.

9.4.1 Benchmark Test Cases Comparison

Table 9.4 provides a qualitative comparison of Maya’s implementation of the simple benchmark test cases versus those provided for Genesis and also a discussion of Maya’s capacity for implementing the complex test cases compared to Genesis’ implementations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Summary Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 assert</td>
<td>Assertions are handled equally well by both Genesis and Maya with the only difference being those of syntax and the amount of support code that the macro programmer is required to provide. Maya requires both an abstract and concrete Mayan to be provided.</td>
</tr>
<tr>
<td>2 foreach</td>
<td>Like assertions, the major distinction between the two implementations of iteration is at the level of minor syntax and overhead.</td>
</tr>
</tbody>
</table>
### 3 printf

Type-safe formatted output is difficult to compare as it was originally coded in both systems because it is heavily dependent on a large degree of standard Java code. This code is dependent on both the style of implementation and the scope of the implementation.

With such code stripped out, all that is left to compare are the two systems relative ability to match the chosen syntax — both handle it with ease. Again, the major differences are in the exact syntactic representation.

### 4 SQL

Maya is not capable of supporting the SQL syntax due to its outside-in evaluation strategy. The similarity of SQL expressions to standard Java expressions would likely cause Maya’s LALR parser some problems.

### 5 Generators

Maya should be capable of providing a generator implementation provided that the use of `suspend` is replaced with `return`. This solution is not as optimal as what can be provided in Genesis, which can match the required syntax exactly.

Despite these syntactic differences, implementations in Maya and Genesis would be very similar as both have similar facilities for syntax interrogation and construction.

### 6 Haskell

Maya is not capable of supporting the Haskell syntax due to its outside-in evaluation strategy. Even if it were, it is unlikely that Maya’s LALR parser could handle the conflict between functional expressions and standard Java expressions.

Both Maya and Genesis provide a similar level of functionality for providing simple macros. The major difference is that Maya requires both abstract and concrete syntax declarations for even the most simple macros. This requires a high level of knowledge.
about the differences between concrete and abstract syntax and interactions between them. The benefit from this approach is that Maya can use this information to provide quite sophisticated pattern-matching on macro arguments.

Genesis unifies the concepts of abstract and concrete syntax to a degree by allowing the mixing of both within macro argument lists. For simple macros this comes at no cost to the user. Such macros simply match on a sequence of syntactic forms and do not require the creation of any new abstract syntax.

Abstract Mayans simplify the creation of abstract syntax to a degree, but simultaneously restrict the power of such additions. In Genesis, the onus is on the programmer to provide abstract syntax definitions as standard Java classes, but these can be used in very flexible and powerful ways. A prime example of this power is the multiple uses of the abstract syntax classes in the Haskell subset implementation (see subsection 9.2.6).

For many simple definitions providing standard Java classes for construction macros is cumbersome, but some of this is alleviated by use of the macroSyntax macro.

Maya’s outside-in evaluation strategy precludes the large-scale modification of the underlying grammar and therefore cannot provide syntax in such an exacting form as to permit language embeddings. Genesis has no such restrictions on macro creation and therefore is capable of implementing the complex constructs.

### 9.4.1.1 Lines of Code Comparison

Both Genesis and Maya are compared on the simple test cases in terms of actual lines of code. The comparison is shown in Table 9.5; blank lines and lines containing only punctuation characters (e.g. opening and closing braces) are subtracted from the total lines of code.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Maya Lines of Code</th>
<th>Genesis Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lines</td>
<td>blank</td>
</tr>
<tr>
<td>1 assert</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>2 foreach</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>3 printf</td>
<td>24</td>
<td>3</td>
</tr>
</tbody>
</table>
In order to provide a fair basis for comparison, the implementations of these benchmarks are modified from the Genesis implementations of section 9.2 and the implementations provided with the Maya distribution. The assertion test case was implemented without Maya’s superficial check for side-effects. The iteration test case was implemented only for Iterator and without syntax options. The type-safe formatted output test case was implemented in a skeletal fashion with the majority of the resultant parse tree calculated by external methods. This was considered the best approach for the printf implementation because it is so heavily dependent on a large amount of standard Java code. See Appendix B for the code used in this comparison.

Genesis code requires less overhead for a typical implementation. Maya requires both abstract and concrete Mayan definitions, whereas for these simple examples Genesis does not. Additionally, Maya either exposes its implementation at cost to the user or at least requires declaration of a collection of exported Mayans.

Maya’s lexical scoping and lack of default Mayan imports also adds to the total number of lines of code.

In general, the code for each of these implementations is no more or less understandable in one language or another, the primary difference is simply in the amount of overhead required to declare macros, use macros, and import the standard environment. Maya tends to require more verbose code for shorter definitions but generally provides more succinct code for generating large quantities of abstract syntax.

### 9.4.2 MultiJava

Maya provides a MultiJava [Cli01] implementation as a partial proof-by-implementation show of its power. MultiJava is a Java extension that provides open classes and multiple dispatch via augmenting methods and multimethods. Augmenting methods allow the programmer to add new methods to a class without the necessity for recompilation. Multimethods provide polymorphic dispatch based on the types of all arguments, not just the first.

The original MultiJava implementation was direct to Java bytecode, but translations to Java code (i.e. by erasure) were provided in [Cli01]. This translation has the following features:
• Each overloaded group of multimethods must have a single dispatcher method created that chooses at run-time which is the most appropriate method to call. The methods themselves are renamed to avoid conflicts. All calls to multimethods can remain unchanged as they resolve to calls to the dispatcher.

• Each augmenting method is translated to a static method with an extra parameter `this_`, and must have its body translated in order to make any implicit use of `this` explicit, and then all use of `this` must be replaced by calls to `this_`.

• Each overloaded group of augmenting methods is wrapped in a single anchor class which contains a instance of a inner dispatcher class.

• The calling of augmenting methods requires a change at the call site to access this dispatcher within an anchor class.

In order to implement MultiJava by erasure, we must be able to parse several occurrences of multimethods and then recombine these to produce dispatch methods. A similar ability is required for translation of augmenting methods, but also that code is translated to add implicit use of `this`, and to modify all implicit or explicit `this` calls to refer to `this_` instead. Any calls to augmenting methods must also be detected and translated — no easy task as they appear like normal method calls.

Both Maya and Genesis have the ability to override the syntax for method calls and to check each occurrence to see if it is either an augmenting method or multimethod.

Genesis is also capable of deferring the creation of dispatcher and anchor classes until all methods of a class have been examined. Macros can be provided that simply construct an abstract syntax object for each multimethod encountered and these objects can be coalesced by overriding the surrounding class declaration.

### 9.4.3 Extensibility Criteria Comparison

#### Summary

Table 9.6: Genesis and Maya Extensibility Criteria Comparison

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Syntax Creation</td>
<td>Genesis allows arbitrary syntax creation with the only restriction being that macros must have at least one argument. Maya provides for arbitrary creation of new</td>
</tr>
</tbody>
</table>
syntax but with the sizeable restriction that Mayans may not rely on other Mayans.

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.2 Syntax Interrogation</strong></td>
<td>Genesis and Maya provide comparable abstract syntax classes and their facilities for interrogation of these classes utilise standard Java constructs. Maya offers pattern matching in formal argument lists as a shorthand for these standard Java facilities.</td>
</tr>
<tr>
<td><strong>1.3 Syntax Overloading</strong></td>
<td>Both Genesis and Maya allow the overloading of the default behaviour of standard Java forms.</td>
</tr>
<tr>
<td><strong>1.4 Static Type Interrogation</strong></td>
<td>Both Genesis and Maya allow for the explicit interrogation of static types in a relatively equivalent fashion. Genesis requires the programmer to annotate their macro definitions with the <code>delayed</code> modifier. Maya provides a built-in facility for pattern matching on static-types whereas Genesis provides this as a standard environment facility. Both facilities offer an equivalent level of power and ease-of-use.</td>
</tr>
<tr>
<td><strong>1.5 Expressiveness</strong></td>
<td>Maya is only capable of providing small modifications to Java syntax due to its outside-in evaluation strategy. Genesis uses a combination of inside-out macro evaluation (for construction of abstract syntax trees or translation not requiring the surrounding context) and outside-in evaluation. As a result of this approach, Genesis is able to provide concise code for simple macros and large scale modifications to the original syntax. By use of command-line specified imports, Genesis is also able to entirely replace its grammar and act as a framework for other languages. Of course, such language must still translate into standard Java code for...</td>
</tr>
</tbody>
</table>
2.1 Simplicity

Concrete Mayan definitions have a potentially more lightweight syntax than Genesis macro definitions, but the requirement to declare arguments as lazy can produce definitions that are in fact more verbose than the equivalent Genesis definitions.

Genesis requires the user to have less understanding of parser theory than Maya. There are none of the obtuse grammar conflicts of LALR parsing with Graph Expansion Parsing.

It is difficult to say if the explicit use of the `delayed` keyword for static type checking macros combined with the inside-out then outside-in approach to macro evaluation is more difficult to understand than Maya’s lazy parser scheme. Both require the programmer to carefully think about the interactions between macros during expansion. Although simple non-delayed macros in Genesis are easier to understand than the equivalent abstract and concrete Mayan definitions.

2.2 Brevity

Both Genesis and Maya provide equivalent quasi-quotation/unquoting facilities and also allow direct, yet more cumbersome, use of the abstract syntax classes.

Maya provides more concise shortcuts for creating new abstract syntax classes but with none of the flexibility of such forms in Genesis.

Genesis has a more lightweight definition of macros than Maya, with each declaration in Maya needing to be exported (and this is a Mayan shortcut for an even more cumbersome low-level approach).

Maya requires Mayans to be both imported and brought into local scope, whereas Genesis has no local
scope facility at all, but instead brings all macros into scope automatically.

<table>
<thead>
<tr>
<th>2.3 Robustness</th>
<th>Both Genesis and Maya provide for explicit name-clash avoidance and automatic hygiene.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Syntactic Correctness</td>
<td>Genesis and Maya both guarantee that valid abstract syntax trees are produced by their macros.</td>
</tr>
<tr>
<td>3.2 Error Detection</td>
<td>Both Genesis and Maya will detect syntax errors within macro calls at compile-time and the resulting expansions are type checked as standard Java. Genesis and Maya both provide facilities for explicitly detecting further errors and signalling the parser and hence the user. Genesis also provides facilities for quiet macro failure where the user need not be informed and also allows macros to report warnings.</td>
</tr>
<tr>
<td>3.3 Error Reporting</td>
<td>Genesis and Maya both allow explicitly detected errors to provide detailed error messages with the source of the error clearly identified. Maya’s use of a restrictive LALR parser allows it to easily detect syntax errors, whereas Genesis’ use of Graph Expansion Parsing provides power, but at the cost of difficult error tracking. Errors caught further down the compilation process are a little harder to track. It is likely that Maya fairs little better than Genesis in this area.</td>
</tr>
</tbody>
</table>
9.5 Graph Expansion Parsing

Graph Expansion Parsing was designed specifically for the implementation of Genesis, and in this section we examine its performance against the general parsers of Earley’s algorithm and the CYK parser.

Issues relating to the general performance of these and more commonly used parsers have been discussed previously in section 6.7.

9.5.1 Acceptable Grammars

Graph Expansion Parsing can operate on any context-free grammars without empty symbols\(^\text{13}\). This class of grammars is far larger than those than can be accepted by CYK, but smaller than Earley’s algorithm which allows empty symbols.

The lack of empty symbols does not overly restrict the languages that can be accepted; it is an easy process to remove empty symbols and while the result is more verbose but no less understandable.

9.5.2 Efficiency

In this section, the efficiency of Graph Expansion Parsing is compared theoretically against the general parsers of both Earley and CYK. Also, empirical results are compared to Earley’s algorithm with the same set of tests as his original paper [Ear70]. In most tests, Graph Expansion Parsing performs on par with the Earley parser.

9.5.2.1 Theoretical Performance

Given \( n \) input tokens, both Earley and CYK parsers require at worst \( O(n^3) \) time. However, \( O(n^3) \) is a requirement for CYK but merely an upper bound for Earley. On bounded state grammars [Ear70] (this includes most LR(k) grammars) Earley’s algorithm operates in linear time. Earley describes three grammars which generate similar languages (shown in Figure 9.1) that take \( O(n) \), \( O(n^2) \), and \( O(n^3) \) time respectively.

\(^{13}\) The Genesis GEP implementation can actually handle context-sensitive grammars as well, as each accepting macro may choose to fail if further specified conditions are not met.
Graph Expansion Parsing has worst case time complexity of $O(n^3)$, but like Earley’s algorithm, it can perform with better complexities on certain grammars. GEP operates on the grammars of Figure 9.1 in $O(n^2)$, $O(n^3)$, and $O(n^3)$ time respectively.

Given $n$ input tokens, both Earley and CYK parsers require $O(n^2)$ space. However, $O(n^3)$ is an upper bound for Earley but a requirement for CYK. These complexities are for recognising a given string, not for producing all possible parse trees. For example, the grammar of Figure 9.1(c) produces an exponential number of possible parses for a given input string, so any algorithm that provides all such parses can do no better than $O(2^n)$ space complexity.

Similarly, the space requirements of Graph Expansion Parsing are dependent upon how ambiguity is handled. If ambiguities are fully resolved as the parse progresses then the space requirements are bounded by $O(n^2)$, if not, the bound is $O(2^n)$.

### 9.5.2.2 Empirical Results

Earley compares his algorithm with a variety of backtracking techniques [Ear70]. It is clearly shown that his algorithm is superior to other general parsers. Graph Expansion Parsing was compared with Earley parsing on all of these grammars.

All of the following time complexities are calculated based on primitive operations. For Earley’s method, the primitive operation used is the act of adding a state to the state set, and for GEP it is attempted matching of a path. GEP paths are built incrementally so each check is effectively a constant operation.

In Table 9.7 the time complexities of Earley parsing and GEP are compared. Shown for GEP is both a forward and backwards scan of the input and also the number of edges.
added to the graph. The first three grammars compared demonstrate left-, right-, and centre-recursive forms respectively. The fourth grammar effectively contains all three recursive forms.

Both Earley’s method and GEP parse all these grammars in linear time, although GEP generally has a smaller constant factor than Earley’s method. No significant difference is seen with GEP between scanning the input left-to-right or right-to-left.

Table 9.7: Earley Versus GEP Time Complexity

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Sentence</th>
<th>Earley</th>
<th>GEP LR</th>
<th>GEP RR</th>
<th>GEP adds</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ::= Ab A ::= a</td>
<td>Ab</td>
<td>ab^n</td>
<td>4n+7</td>
<td>6n+1</td>
<td>6n+1</td>
</tr>
<tr>
<td>S ::= aB B ::= aB</td>
<td>b</td>
<td>a^nb</td>
<td>6n+4</td>
<td>6n+1</td>
<td>6n+1</td>
</tr>
<tr>
<td>S ::= ab</td>
<td>aSb</td>
<td>a^nb^n</td>
<td>6n+4</td>
<td>7n-3</td>
<td>7n-3</td>
</tr>
<tr>
<td>S ::= AB A ::= a</td>
<td>Ab B ::= bc</td>
<td>bB</td>
<td>Bd</td>
<td>ab^ncd</td>
<td>18n+8</td>
</tr>
</tbody>
</table>
Table 9.8 compares Earley parsing and GEP on more complicated grammars with mutually recursive components. The third grammar is the most representative of a real programming language grammar. The choice of strings is taken from [Ear70] so that a direct comparison could be made.

Graph Expansion Parsing performs the most favourably on the third grammar which is a representation of a propositional calculus. As this is the most “real world” of the grammars, GEP seems well suited to non-theoretic use.

With two of these three grammars sizeable differences are visible between performing a left-to-right scan of the input to performing a right-to-left scan. In general the left-to-right scan performs considerably better. The largest difference is in the first grammar and is due to the predominance of left-recursive elements.
## Table 9.8: Earley Versus GEP Comparison

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Sentence</th>
<th>Earley</th>
<th>GEP LR</th>
<th>GEP RR</th>
<th>GEP adds</th>
</tr>
</thead>
<tbody>
<tr>
<td>X ::= a</td>
<td>ededeab^2</td>
<td>33</td>
<td>37</td>
<td>35</td>
<td>11</td>
</tr>
<tr>
<td>X ::= Xb</td>
<td>ededeab^2</td>
<td>45</td>
<td>77</td>
<td>98</td>
<td>27</td>
</tr>
<tr>
<td>X ::= Ya</td>
<td>ededeab^3</td>
<td>63</td>
<td>137</td>
<td>188</td>
<td>51</td>
</tr>
<tr>
<td>Y ::= e</td>
<td>ededeab^3</td>
<td>633</td>
<td>2037</td>
<td>3038</td>
<td>811</td>
</tr>
<tr>
<td>Y ::= YdY</td>
<td>ededeab^3</td>
<td>79</td>
<td>123</td>
<td>152</td>
<td>43</td>
</tr>
<tr>
<td>S ::= AB</td>
<td>ededeab</td>
<td>194</td>
<td>292</td>
<td>363</td>
<td>119</td>
</tr>
<tr>
<td>S ::= AB</td>
<td>ededeab</td>
<td>251</td>
<td>371</td>
<td>460</td>
<td>159</td>
</tr>
<tr>
<td>A ::= a</td>
<td>ededeab</td>
<td>33</td>
<td>45</td>
<td>63</td>
<td>10</td>
</tr>
<tr>
<td>A ::= a</td>
<td>ededeab</td>
<td>44</td>
<td>79</td>
<td>123</td>
<td>63</td>
</tr>
<tr>
<td>B ::= b</td>
<td>ededeab</td>
<td>8</td>
<td>18</td>
<td>33</td>
<td>51</td>
</tr>
<tr>
<td>B ::= b</td>
<td>ededeab</td>
<td>114</td>
<td>154</td>
<td>203</td>
<td>328</td>
</tr>
<tr>
<td>C ::= c</td>
<td>ededeab</td>
<td>123</td>
<td>165</td>
<td>217</td>
<td>359</td>
</tr>
<tr>
<td>C ::= c</td>
<td>ededeab</td>
<td>194</td>
<td>292</td>
<td>363</td>
<td>119</td>
</tr>
<tr>
<td>D ::= d</td>
<td>ededeab</td>
<td>251</td>
<td>371</td>
<td>460</td>
<td>159</td>
</tr>
<tr>
<td>D ::= d</td>
<td>ededeab</td>
<td>44</td>
<td>79</td>
<td>123</td>
<td>63</td>
</tr>
<tr>
<td>F ::= C</td>
<td>ededeab</td>
<td>33</td>
<td>37</td>
<td>35</td>
<td>11</td>
</tr>
<tr>
<td>F ::= C</td>
<td>ededeab</td>
<td>45</td>
<td>77</td>
<td>98</td>
<td>27</td>
</tr>
<tr>
<td>F ::= S</td>
<td>ededeab</td>
<td>63</td>
<td>137</td>
<td>188</td>
<td>51</td>
</tr>
<tr>
<td>F ::= S</td>
<td>ededeab</td>
<td>633</td>
<td>2037</td>
<td>3038</td>
<td>811</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>79</td>
<td>123</td>
<td>152</td>
<td>43</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>194</td>
<td>292</td>
<td>363</td>
<td>119</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>251</td>
<td>371</td>
<td>460</td>
<td>159</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>33</td>
<td>37</td>
<td>35</td>
<td>11</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>45</td>
<td>77</td>
<td>98</td>
<td>27</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>63</td>
<td>137</td>
<td>188</td>
<td>51</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>633</td>
<td>2037</td>
<td>3038</td>
<td>811</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>79</td>
<td>123</td>
<td>152</td>
<td>43</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>194</td>
<td>292</td>
<td>363</td>
<td>119</td>
</tr>
<tr>
<td>F ::= U</td>
<td>ededeab</td>
<td>251</td>
<td>371</td>
<td>460</td>
<td>159</td>
</tr>
</tbody>
</table>
Conclusion and Future Work

1. Introduction
2. Defining extensibility
3. Assessing extensibility
4. Reviewing extensibility
5. Designing the language
6. Reviewing parsing
7. Implementing a parser
8. Language implementation
9. Evaluation
10. Conclusion
10.1 Conclusion

The original design goals of Genesis are summarised as following (from section 5.2):

- provide arbitrary syntax creation;
- allow for compile-time interrogation of syntax trees;
- outward language simplicity (a simple to use system);
- programmer support (via quasi-quotation, hygiene, and guarantees of syntactic correct translations);
- a lack of complicated parser restrictions;
- inward language simplicity (a small definition with extensions written with it to provide further facilities); and
- provide decent error reporting.

Genesis is very effective in meeting these original design goals.

Its power of expression surpasses that of other extensible languages and it does this with a very simple macro syntax and a relatively clean design. It has proven itself to be capable of providing programmer support in the form of quasi-quotation, unquoting, optional macro parameters, automatic list class generation, and automatic static-type checking extensions. The number of such shorthands will no doubt increase with time.

Graph Expansion Parsing supports Genesis’ simple macro definitions and places a low burden on the macro programmer. It provides for a intuitive approach to creating syntax and defining transformations in a way that other such systems are not quite capable of matching. However, it does require the programmer to sometimes think about issues of ambiguities that are more complicated than those that occur in traditional languages.

Genesis’ implementations of the benchmark test suite provide strong evidence of the success of its design. In particular, the ability to program in a Haskell subset free of any legacy Java code is a facility that would normally be possible only with compiler-compilers. The Haskell subset is the most clear demonstration of the power and flexibility of macro definitions in Genesis. Not only is the Haskell syntax matched exactly (ignoring true Haskell layout rules), the implementation itself is relatively straightforward due to Genesis allowing multiple uses of its user definable abstract syntax classes.
More work needs to be done on the provision of quality error messages. Graph Expansion Parsing does not lend itself easily to pinpointing the exact cause of errors and more research is required into this area (see section 10.2.8). The current implementation of Genesis requires direct output of Java source files for compilation with an external compiler. This greatly complicates the possibility of tracking errors to their source. This will hopefully be addressed by future research (see subsection 10.2.7).
10.2 Future Work

There are many possibilities for future work with the Genesis programming language: modifications to the language design, improvement of the current implementation, further research into the created parsing scheme, and the implementation of more sophisticated extensions within the current language.

It is hoped that such continued research will ultimately lead to extensible concepts being incorporated into the larger programming community.

10.2.1 Flexible Lexical Analysis

The current tokeniser makes only a few decisions as to the meaning of sequences of symbols and characters: Literal strings and characters are detected early, and $ and _ characters are treated as alphanumeric characters rather than symbols.

In addition, floating-point forms are accepted that contain spaces in between numbers and other characters. This is undesirable but is unavoidable with the current tokeniser strategy. The choice could have been made to detect floating-point literals within the tokeniser, but this may have affected the use of “.” within other expressions.

For the tokeniser to be as flexible as possible it should be able to allow users to define all of these literal forms explicitly.

It should be possible to reuse macro forms but to apply them to character strings at the tokeniser stage. Perhaps this would be accomplished by defining a class TokenChar and allowing users to write macros for combining them — such macros could be automatically recognised by the compiler as special tokeniser macros. Code Example 10.1 demonstrates the possible usage of such a system to explicitly handle string literals — something currently not possible with Genesis.

```
macro StringLiteralToken (", StringLiteralCharList, ") { ... }
macroList StringLiteralCharList(StringLiteralChar);
macro StringLiteralChar (TokenChar t) { ... }
macro StringLiteralChar (\, TokenChar c) throws ConditionsNotMet {
    // make sure this is a valid escape
}
```

Code Example 10.1: Extended Tokeniser Possibility
10.2.2 Delayed Macros

The delayed macros facility is instrumental in allowing both construction macros and static-type checked macros to operate in tandem. The usage of delayed macros is the source of much of Genesis’ power, but could perhaps also be a source of programmer confusion — maybe we can do better.

Consider the program fragment in Code Example 10.2. This fragment uses a specialisation of `forall` for arrays that allows the user to drop the type of iteration variable as it can be inferred from the type of the array. This is a prime example of the use of delayed macros, both for expansion of `forall` and for `printf`: the inside-out parse allows construction of the parse-tree and the outside-in parse performs the translations.

```plaintext
int[] array = { 1, 11, 27, 42 };
forall num in array {
    printf("%d\n", num);
}
```

Code Example 10.2: Macro Expansion Requiring Delayed Macros

There are a number of possibilities for improvement on the current scheme. Either use of delayed could be inferred by some mechanism, or a parser improvement could be made to provide more previous scope information by the time a macro expansion was reached (although this may mean restricting macros static-typing abilities to declarations preceding the macro call).

Inference of delayed macros could simply occur at macro expansion time if an attempted type-check fails to find the type, or by some method that checks for calls to the type system when a macro is compiled.

If some form of modified backtracking top-down parsing scheme (that could handled left-recursion) could be applied to the Genesis programming language the type of array would be known by the time `forall` was reached, and by the time `printf` was reached perhaps some mechanism for querying the half-matched `forall` macro as to the type of its arguments could be constructed.

10.2.3 Zero Argument Macros

As previously discussed in subsection 5.3.4, the Genesis definition does not support macros that have no arguments. Such macros would possibly allow a more natural
construction of definition with optional parts, although the optional parameter extension in subsection 8.7.2.3 provides quite an elegant method for this.

The inclusion of zero argument macros was avoided as it complicates the construction of the parser. It would, however, be possible to extend the parsing method to include such forms.

The most obvious implementation would be to add arcs that construct zero argument macros with the same start and end node to the parse graph. This would either be performed for each node without fail or with some more sophisticated approach that adds such arcs only when there is a possibility that they will be required by some later macro.

It remains to be seen if the inclusion of zero argument macro forms is important enough to justify the extra implementation cost.

### 10.2.4 Migration to Java 1.5

Genesis was implemented with Java 1.4 which means that it was not able to take advantage of generics. As a result, many components of the current implementation are not as neat as they could be. For example, the `macroList` extension would likely prove to be completely unnecessary with a Genesis implementation that uses generics, or at least would not need to create a new list each time it was used. The implementation of the generators test case would also benefit from generics.

The typing system could perhaps benefit from generics — when static-types are known they could be passed around with the current expression as a type parameter. Macros could be prevented from matching on these types directly without any explicit checks.

Java 1.5 annotations would most likely allow the process of mangling to be simplified or completely removed. More advanced uses of such metadata may help in the writing of macros and reduce the number of new classes that are created in order to create a particular extension.

### 10.2.5 Parser Efficiency

Section 7.2 already detailed some optimisations to the parser, but nonetheless the current Graph Expansion Parser has much room for improved performance.
It may prove to be possible to discover sub-graphs that have no possibility of further additions and such forms could be ignored for the rest of the parse. The current algorithm performs many checks that are required and does so repeatedly. Any graph pruning technique would provide quite a boost in efficiency.

Another minor improved to optimisation could come from collapsing some of the information in the partial match tree. A simple example of this kind of operation is if a grammar contains a rule that converts a token into an identifier and a rule that converts an identifier into a simple expression then upon successful conversion of the token into an identifier we can produce an expression simultaneously with further matching. Well constructed abstract syntax class hierarchies already perform similar optimisations, but it would be desirable to provide this functionality in a more general way. It may even be possible to apply this approach in a more general way to improve efficiency.

### 10.2.6 Context-sensitive Graph Expansion Parsing

It was briefly mentioned that Graph Expansion Parsing can parse context-sensitive forms due to the ability of macros to throw exceptions when further conditions are not met. It may prove interesting to explore these abilities in more detail, perhaps even writing a GEP parser generator in Genesis itself.

### 10.2.7 Integration of Genesis Parsing and Java Compiling

The Genesis compiler makes use of a standard Java compiler to actually produce its final output. Much could be gained by producing an integrated system. The most obvious area for improvement would be with increased ability to track errors.

### 10.2.8 Improved Error Tracking

Genesis’ major failing is in both the pin-pointing of syntax errors and errors that occur after expansion has taken place. As just mentioned, it would be simpler to track errors if the entire compilation was performed by an integrated system.

Improved syntax error detection for parsers such as GEP is an open question and requires much extensive research. Indeed, it may not be possible to greatly improve this situation and another parsing approach may ultimately be required.
10.2.9 Usability Surveys

Genesis has had limited usage to this point and it would be both interesting and informative to see how others took advantage of its facilities. Feedback from such use would only serve to improve the language.

10.2.10 Library Support

Genesis could no doubt benefit from an increased number of shortcuts for tasks that are identified as often occurring and repetitive. The identification of such required shortcuts would no doubt occur with increased usage of Genesis (as discussed in the previous section).

A class (or set of classes) for providing generalised transformations on arbitrary syntax trees would be of great use for many complicated extensions. Implementations of generators, Haskell, and MultiJava could all benefit from such a library.

Extensible libraries for SQL, HTML, XML, or regular expressions could provide increased performance over currently available systems.

Many extensions could be created to showcase Genesis’ flexibility and power. Implementations of other Java extensions could be undertaken such as (the previously mentioned) MultiJava or Pizza.

10.2.11 Improved Embedded Haskell

The embedded Haskell subset is of great interest. If after further testing of the current subset it is found to be as useful as it appears at first glance, it would be desirable to look at providing compilation rather than interpretation of the subset.

If compilation proves successful, the subset could be increased to handle the full core of Haskell and then macros could build upon this in order to produce a full embedded Haskell implementation.

Such an embedding of Haskell would bring the language to a much larger audience and the appeal of using a clean functional language for calculation and a traditional imperative language for control and user interaction is particularly appealing.
Also, in further extensions it would no doubt be interesting to allow the calling of appropriate Java functions from within Haskell code as well. Such mixing of imperative and functional constructs is not new, but perhaps mixing of two large scale languages is.

### 10.2.12 Ultimate Aim

The ultimate aim of this work (and other similar works) is adoption of extensibility amongst the wider audience. The Java programming language seems an ideal vehicle for such an occurrence as it has been repeatedly updated through its community review process. As has been repeatedly stated, if extensibility had been a part of Java from the very beginning, the changes that it has undergone would have been possible in a much different fashion. Indeed, there is a large number of extensions to Java that have not seen wide-spread adoption, perhaps with an extensible Java these languages would have the capacity to reach a larger audience.

Continued work on the Genesis language will hopefully add much that is interesting (and perhaps some that is useful) to the domain of extensible programming languages.


Chapter 10: Conclusion and Future Work


• 295 •
http://www.brics.dk/bigwig/publications/macro.pdf

http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm


http://www.cs.iastate.edu/~cclifton/multijava-data/papers/TR00-06.pdf


http://www.cse.ucsc.edu/~abadi/Papers/jmgrammar.ps

[Cor00] Robert Corbett: Berkeley Yacc, June 2000. Available from:


http://people.redhat.com/drepper/nptl-design.pdf

http://www.scheme.com/tspl3/


● 296 ●


CHAPTER 10: CONCLUSION AND FUTURE WORK

• 298 •


http://www.gamers.org/dEngine/quake/spec/quake-spec34/index1.htm

http://research.microsoft.com/Users/simonpj/Papers/conc-ffi/conc-ffi.ps


http://www.masswerk.at/algol60/report.htm


http://research.microsoft.com/Users/simonpj/papers/marktoberdorf/


● 300 ●


A.1 Abstract Syntax Classes

This appendix provides details of the abstract syntax classes used in the implementation of the Genesis compiler. Class hierarchy diagrams are supplied for groups of classes and interfaces that benefit from extra explanation.

A.1.1 High-level Abstract Syntax Classes

Figure A.1 is a reproduction of class hierarchy in Figure 5.12. It is provided as a point of reference to aid in the understanding of the expansion of the definition of some of the classes within this hierarchy in later subsections. The rest of this subsection lists the classes not covered by this hierarchy and the children of the Statement class.

![High-level Abstract Syntax Class Hierarchy]

Code Example A.1 lists the classes used for the basic structure of a Java source file. A compilation unit consists of an optional package declaration, import declarations, and some type declarations (covered in subsection A.1.2).

```java
// classes for creating the end result of parsing a file
class CompilationUnit implements AbstractSyntax;
class PackageDeclaration implements AbstractSyntax;
class ImportDeclaration implements AbstractSyntax;

// list declarations
class ImportDeclarations extends List implements AbstractSyntax;
```

Code Example A.1: Compilation Unit Classes
The classes of Code Example A.2 are used for typing and within declarations and statements that require a type.

```java
// class for providing static type information
class Type implements AbstractSyntax;
// list declarations
class Types extends List;
```

Code Example A.2: Type Classes

Code Example A.3 contains the classes for statements. Local variable declarations are given special treatment as they are only allowed to appear within blocks.

```java
// classes for providing statements and lists of statements
// interface for statements that are allowed to appear within blocks
interface BlockStatement extends AbstractSyntax;
// variable declarations are the only statement that MUST be part of a block
class LocalVariableDeclarationStatement extends LocalVariableDeclaration implements BlockStatement;
// statements
interface Statement extends BlockStatement;

class EmptyStatement implements Statement;
class Labeled implements Statement;
class Break implements Statement;
class Continue implements Statement;
class Throw implements Statement;
class Return implements Statement;
class Synchronized implements Statement;
class Assert implements Statement;
class Catch;
class Try implements Statement;
class ExpressionStatement implements Statement;
class IfThenElse implements Statement;
class While implements Statement;
class Do implements Statement;
class Switch implements Statement;
class SwitchLabel implements AbstractSyntax;
class SwitchBlock implements AbstractSyntax;
class For implements Statement;
// a block is allowed to appear in a class as initialisation code
class Block implements Statement, ClassMemberDeclaration;
// list declarations
class Statements extends List;
class BlockStatements extends List;
class SwitchLabels extends List;
class SwitchBlocks extends List;
class SwitchStatements extends List;
class Catches extends List;
```

Code Example A.3: Statement Classes
A.1.2 Declaration Abstract Syntax Classes

In Figure A.2 the class hierarchy for declarations is shown. Subclasses are pictured with doubly-lined arrows.

![Class Hierarchy Diagram]

Code Example A.4 contains the classes for variable declarations and initialisers. Field declarations can also appear as either class or interface declarations.

```java
// classes for providing variable declarations
class VariableDeclaration implements Typeable;
class FormalParameter extends VariableDeclaration;
class LocalVariableDeclaration extends VariableDeclaration;
class FieldDeclaration implements ClassMemberDeclaration, InterfaceMemberDeclaration;
interface VariableInitializer extends AbstractSyntax;
class SimpleInitializer implements VariableInitializer;
class ArrayInitializer implements VariableInitializer;
class VariableDeclarator implements AbstractSyntax;
class VariableDeclaratorId extends VariableDeclarator;
// list declarations
class FormalParameters extends List;
class VariableDeclarators extends List;
class VariableInitializers extends List;
```

Code Example A.4: Variable Declaration Classes
Code Example A.5 contains the classes for method and type declarations. Type declarations can be either class or interface declarations and method declarations also cover constructors, abstract methods, and macros.

```java
// classes for providing method and type declarations
// interfaces for class and interface member declarations
interface ClassMemberDeclaration extends AbstractSyntax;
interface InterfaceMemberDeclaration extends AbstractSyntax;

// all method declarations can appear within a class and are typeable
class MethodDeclaration implements ClassMemberDeclaration, Typeable;
class MacroDeclaration extends MethodDeclaration;
class ConstructorDeclaration extends MethodDeclaration
  implements InterfaceMemberDeclaration;

// type declarations
class TypeDeclaration implements ClassMemberDeclaration, BlockStatement;
class ClassTypeDeclaration extends TypeDeclaration
  implements InterfaceMemberDeclaration {
  class InterfaceTypeDeclaration extends TypeDeclaration
    implements InterfaceMemberDeclaration;
class EmptyTypeDeclaration extends TypeDeclaration;
}

// modifiers for methods, classes, etc.
class Modifiers implements AbstractSyntax;

// list declarations
class ClassMemberDeclarations extends List;
class InterfaceMemberDeclarations extends List;
class MacroParameters extends List;
class TypeDeclarations extends List;
class Throws extends List;
```

**Code Example A.5: Method and Type Declaration Classes**

### A.1.3 Expression Abstract Syntax Classes

Figure A.3 contains the class hierarchy for expressions (and also shows where identifiers and variable declarations fit). The subclasses of `StatementExpression` and `Literal` are abbreviated in the hierarchy but expanded in the following Code Examples.

Code Example A.6 details the classes used for expressions and contains interfaces to differentiate between normal expressions and those that can appear as statements and those that can appear on the left-hand side of an assignment.

Code Example A.7 and Code Example A.8 contain classes for identifiers and literals respectively.
Figure A.3: Expression Abstract Syntax Class Hierarchy

Code Example A.6: Expression Classes

```
// classes for expressions
interface Typeable extends AbstractSyntax;
// non-side effect expressions
interface Expression extends Typeable;
class Brackets implements Expression;
class Infix implements Expression;
class Cast implements Expression;
class InstanceOf implements Expression;
class IfThenElseExpression implements Expression;
// side-effect expressions
interface StatementExpression extends Expression;
class Assignment implements StatementExpression;
class Prefix implements StatementExpression;
class Postfix implements StatementExpression;
class MethodCall implements StatementExpression;
class Creation implements StatementExpression;
class ArrayCreation implements StatementExpression;
// expressions that can appear on an assignment's left-hand side
interface LeftHandSide extends Expression;
class Simple implements LeftHandSide;
class ArrayAccess implements LeftHandSide;
class FieldAccess implements LeftHandSide;
// operators
class Operator implements AbstractSyntax;
// list declarations
class StatementExpressions extends List;
class Expressions extends List;
class Arguments extends Expressions;
class ArrayCreationExpressions extends Expressions;
```
CHAPTER 10: CONCLUSION AND FUTURE WORK

FUTURE WORK

Code Example A.7: Identifier Classes

```java
// classes for identifiers, dot separated names, and symbols

class Identifier implements Typeable;
class Name implements AbstractSyntax;
class Symbol implements AbstractSyntax;
```

Code Example A.8: Literal Classes

```java
// classes for literals

interface Literal extends Expression;

class LiteralString implements Literal;
class LiteralInteger implements Literal;
class LiteralChar implements Literal;
class LiteralBoolean implements Literal;
class LiteralFloat implements Literal;
class LiteralNull implements Literal;
```
B.1 Assertions

Code Example B.1 and Code Example B.2 contain the Genesis and Maya definitions of `assert` as used in subsection 9.4.1.1.

```
class Assert {
    macro Statement (assert, Expression e) {
        return {{
            if (!e) {
                System.err.println("Assertion Failed: " + (new StringLiteral(e.toString())));
                throw new AssertionError("Assertion Failed");
            }
        }};
    }
}
```

Code Example B.1: Genesis Assertion Definition

```
import maya.tree.*;
import maya.grammar.*;

use Syntax;
use ForEach;

abstract Statement syntax (assert(Expression));

public class Assert implements MetaProgram {
    public Environment run(Environment env)
    {
        Statement syntax A(assert(Expression e));
        {
            return new Statement {
                if (!e) throw new Error("Assertion failed");
            };
        }
        return new A().run(env);
    }
}
```

Code Example B.2: Maya Assertion Definition
B.2 Iteration

Code Example B.3 and Code Example B.4 contain the Genesis and Maya definitions of `forall` as used in subsection 9.4.1.1.

```
import java.util.Iterator;
public class TestForall {
    delayed macro (forall, (, FormalParameter p, ), in,
        Expression:Iterator e, Statement b) throws TypeMismatch {
        return {{
            for(Iterator i = (`e).iterator(); i.hasNext(); ) {
                `(p.type()) `(p.getIdentifier()) = (`(p.type())) i.next();
                b
            }
        }};
    }
}
```

Code Example B.3: Genesis Iterator Definition

```
import java.util.*;
import maya.tree.*;
import maya.grammar.*;
abstract Statement syntax(MethodName(Formal) lazy(BraceTree, BlockStmts));
Statement syntax ForEach(Expression:Iterator enumExp \ . foreach(Formal var) lazy(BraceTree, BlockStmts) body) {
    final StrictTypeName castType = StrictTypeName.make(var.getType());
    return new Statement {
        for(Iterator enumVar = $enumExp; enumVar.hasNext(); ) {
            $(DeclStmt.make(var)) $(Reference.makeExpr(var.getLocation())) = ($castType) enumVar.next();
            $body
        }
    }
    public defineMayanContainer(ForEach) { ForEach }
```

Code Example B.4: Maya Iterator Definition
B.3 Type-safe Formatted Output

Code Example B.5 and Code Example B.6 contain the Genesis and Maya definitions of `printf` as used in subsection 9.4.1.1.

```java
class PrintF {
    delayed
    macro ExpressionStatement (printf, (, LiteralString s, ,, Arguments list, ))
    throws TypeMismatch, TooManyActualParameters, TooManyPlaceHolders {
       Expression exp = /* call to external function to do generation */;
        return {{ System.out.println(`exp); }};
    }
}
```

Code Example B.5: Genesis Type-safe Formatted Output Definition

```java
import maya.tree.*;
import maya.grammar.*;
use maya.util.Syntax;
use maya.util.RunMayans;
Expression syntax
PSprintf(Expression:PrintStream p.printf(list list list list(Expression, ',',') args))
{
    final FormatState state = new FormatState(args);
    return new Expression {{
        PrintStream writer = $p;
        $({
            StmtList ret = new StmtList{};
            for (Expression e = state.parse(); e != null; e = state.parse())
            {
                ret = new StmtList { $(as Statement ret) writer.print($e); }
            }
        (Statement) ret;
    };
        writer;
    }};
}
public defineMayanContainer(Printf) { PSprintf }
```

Code Example B.6: Maya Type-safe Formatted Output Definition