Understanding and Improving Object-Oriented Software Through Static Software Analysis

---

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

in the

University of Canterbury

by

Warwick Irwin

University of Canterbury

2007

Contents

List of figures ..... **v**

Abstract ..... **viii**

1 Introduction **1**
1.1 Overview 1
1.2 Understanding software 2
1.3 Static analysis 4
1.4 Outline of thesis 7

2 Existing static analysis technology **8**
2.1 The phases of static analysis 8
2.1.1 An example of static analysis phases 11
2.2 Conventional parser development 15
2.3 Semantic modelling of software 22
2.3.1 Reflection 24
2.3.2 Alternative models 26
2.4 Informing software engineers 33
2.4.1 Software measurement 34
2.4.2 Software visualisation 38

3 Parsing background **41**
3.1 Parsing concepts and terminology 42
3.2 Grammar classes and parsing algorithms 46
3.2.1 Context free grammars and others 46
3.2.2 General and restricted context free parsers 48
3.2.3 LL parsers 50
3.2.4 LR parsers 52
3.3 LR parser classes—an escalating example 56
3.3.1 LL(1) parsing 57
3.3.2 LL(1) parsing using automata 59
3.3.3 LR(0) parsing 61
3.3.4 SLR(1) parsing 69
3.3.5 LALR(1) parsing 72
3.3.6 LR(1) parsing 74
3.3.7 GLR parsing 77
3.3.8 Dotted items 80
3.4 Chapter summary 81

4 Yakyacc: Yet another kind of yacc **82**
4.1 Yet another parser generator? 82
4.2 Yakyacc architecture 85
4.2.1 Input of grammars 87

4.2.2 Output of parsers 89
4.3 Yakyacc design 90
4.3.1 Runtime PDA 90
4.3.2 PDA construction 108
4.4 Discussion 122
4.4.1 GLR parsing 122
4.4.2 GLR-based sub-sentence parsing 124
4.4.3 Hybrid parsing algorithms 125
4.4.4 Heterogeneous k 126
4.5 Evaluation 126
4.6 Chapter summary 128

5 JST: Semantic modelling of Java code 129
5.1 Why Java? 131
5.2 The type system of Java 132
5.3 Development approach 133
5.4 JST architecture 135
5.5 JST model 136
5.5.1 Main classes 141
5.5.2 Types 144
5.5.3 Typed Declarations 145
5.5.4 Executable classes 146
5.6 Populating the model 147
5.7 Emitting the model 149
5.8 Discussion 150
5.8.1 Strengths 150
5.8.2 Weaknesses and limitations 152
5.8.3 Extensions 153

6 Measuring and visualising Java programs 154
6.1 The role of metrics and visualisations 156
6.2 Pipeline architecture 164
6.3 Metrics calculation 168
6.3.1 CodeRank 171
6.4 Generating virtual world visualisations 176
6.4.1 Class Clusters 181
6.4.2 Other applications 183

7 Conclusions and future work 184
7.1 Conclusions 184
7.2 Continuing and future work 186
7.3 Final words 188

Acknowledgments 190

References 191

List of figures

Figure **Page**

Figure 1: Static analysis phases 8
Figure 2: Combined semantic analysis 10
Figure 3: A simple Java program 11
Figure 4: Tokens produced by a scanner 11
Figure 5: Parse tree fragment produced by a parser 13
Figure 6: Semantic model fragment produced by a semantic analyser 14
Figure 7: Manual and automatic parser development 16
Figure 8: Java reflection semantic model classes 25
Figure 9: JDT core model interfaces (simplified) 28
Figure 10: JDT AST classes (simplified) 29
Figure 11: JDT binding classes (simplified) 31
Figure 12: Fragment of Java exposition grammar 42
Figure 13: Raw text grammar fragment 43
Figure 14: EBNF grammar rule expanded to BNF 44
Figure 15: Hierarchy of LR grammar classes 54
Figure 16: A grammar for a trivial language, and some sentences it generates 57
Figure 17: Example sentences and parse trees 57
Figure 18: A recursive grammar and example sentence 57
Figure 19: Example parse tree for recursive grammar 58
Figure 20: A recursive descent parser 58
Figure 21: Recursive descent deterministic finite automata 60
Figure 22: Grammar modified to defeat LL(1) parser 61
Figure 23: Grammar modified to defeat LL(k) parser 62
Figure 24: Combining LL automata into a nondeterministic PDA 62
Figure 25: LR(0) Pushdown Automaton 63
Figure 26: Execution of PDA 64
Figure 27: Partially constructed LR(0) parser for grammar with nested parentheses 66
Figure 28: LR(0) PDA with ε-transitions removed 67
Figure 29: Deterministic LR(0) PDA 68
Figure 30: Execution of PDA as it parses nested parentheses 69
Figure 31: Grammar modified to require an SLR(1) parser 69
Figure 32: Nondeterministic LR(0) PDA for grammar with methodCall 70
Figure 33: LR(0) state containing a reduce-reduce conflict 70
Figure 34: LR(1) state that eliminates the conflict 70
Figure 35: Deterministic SLR(1) PDA for grammar with methodCall 71
Figure 36: A grammar that produces an LR(0) shift-reduce conflict, and its resolution in an SLR(1) PDA fragment 72
Figure 37: Grammar modified to require an LALR(1) parser 72
Figure 38: SLR(1) PDA fragment with inadequate state 73
Figure 39: LALR(1) PDA 73
Figure 40: Grammar modified to require an LR(1)parser 74

Figure 41: Nondeterministic LALR(1) PDA (ε–transitions removed) 74
Figure 42: Reduce-reduce conflict reachable from two sources 75
Figure 43: Inadequate LALR(1) state 75
Figure 44: LR(1) PDA fragment 75
Figure 45: LR(1) PDA 76
Figure 46: Grammar modified to require a GLR parser 77
Figure 47: Inadequate LR(1) PDA 77
Figure 48: Execution of PDA with branching stack 79
Figure 49: Graph-structured stack configuration 80
Figure 50: Dotted items derived from super rule 80
Figure 51: Decoupled architecture of yakyacc 85
Figure 52: Generating a parser for use in a pipeline 86
Figure 53: Potential mechanism for custom parser actions 87
Figure 54: Alternative grammar specifications using BNF and EBNF 88
Figure 55: Using yakyacc to generate bnf2xml 89
Figure 56: Parser generation using a stylesheet 89
Figure 57: Runtime package structure 91
Figure 58: Grammar classes 92
Figure 59: Parse tree classes 94
Figure 60: Parse trees for recursive grammar 95
Figure 61: Token factory classes 96
Figure 62: Tree factory classes 96
Figure 63: Parse tree visitors 97
Figure 64: PDA base classes 97
Figure 65: The `parse()` method of `Parser` 98
Figure 66: Table-driven parsers 99
Figure 67: The `step()` method of `SimpleParser` 100
Figure 68: Rekers' algorithm 101
Figure 69: Graph-driven parsers 103
Figure 70: The `step()` method of `DeterministicParser` 103
Figure 71: The `step()` method of `NondeterministicParser` 103
Figure 72: The `shift()` and `goTo()` methods of `DeterministicParser` 104
Figure 73: The `shift()` and `goTo()` methods of `NondeterministicParser` 104
Figure 74: Graph-driven parser stacks 105
Figure 75: Classes for `Token` input 106
Figure 76: State machine classes 107
Figure 77: State machine builder 108
Figure 78: Parser generator package structure 109
Figure 79: Yakyacc grammar classes 109
Figure 80: `KParser` 110
Figure 81: `PopNode` construction 111
Figure 82: `Transmogrifier` classes 111
Figure 83: Major `Transmogrifier` methods 112
Figure 84: Priming `Transmogrifiers` with different `PopNodes` 112
Figure 85: `PDAMaker` 113
Figure 86: Escalating parser construction 113
Figure 87: The `split()` method of `LRTransmogrifier` 114
Figure 88: The `prune()` method of `Transmogrifier` 115
Figure 89: `TransmogrifierGSSNode` hierarchy 116

Figure 90: The `split()` method of `LRPopNode` 117
Figure 91: The `partition()` method of `LRPopNode` 117
Figure 92: Splitting constructor of `LRPopNode` 118
Figure 93: Parse trees that calculate lookahead 119
Figure 94: Dynamic state machine classes 121
Figure 95: JST overview class diagram 140
Figure 96: Main JST classes 141
Figure 97: `Scope` and `Decl` classes 142
Figure 98: Example name look-up rules 142
Figure 99: `PackageDecl` and `SourceFile` 143
Figure 100: `TypeDecl` and its subclasses 144
Figure 101: Method invocation resolution 145
Figure 102: `TypedDecl` and subclasses 146
Figure 103: `ExecutableCode` and its subclasses 147
Figure 104: `ModelVisitor` hierarchy 149
Figure 105: JST memory usage 151
Figure 106: Code age editor 160
Figure 107: `SeeSoftLike` 161
Figure 108: Pipeline input and output 164
Figure 109: Pipeline example 166
Figure 110: Pipeline XML files 167
Figure 111: Number of methods visitor 170
Figure 112: `CodeRanker` 173
Figure 113: ClassRank parallel coordinates graph 173
Figure 114: Aliens program metrics 174
Figure 115: Visualisation filters in the pipeline 177
Figure 116: Class cohesion variations 178
Figure 117: Separable class 179
Figure 118: Real class cohesion 179
Figure 119: VRML browser 180
Figure 120: Magic book 180
Figure 121: VT CAVE (console) 180
Figure 122: UC GeoWall 180
Figure 123: Class cluster 181
Figure 124: Class cluster with method cones 182
Figure 125: Class cluster with method spikes 182
Figure 126: NOC 3D TreeMap 183
Figure 127: Debugs 183

Abstract

Software engineers need to understand the structure of the programs they construct. This task is made difficult by the intangible nature of software, and its complexity, size and changeability. Static analysis tools can help by extracting information from source code and conveying it to software engineers. However, the information provided by typical tools is limited, and some potentially rich veins of information—particularly metrics and visualisations—are under-utilised because developers cannot easily acquire or make use of the data.

This thesis documents new tools and techniques for static analysis of software. It addresses the problem of generating parsers directly from standard grammars, thus avoiding the common practice of customising grammars to comply with the limitations of a given parsing algorithm, typically LALR(1). This is achieved by a new parser generator that applies a range of bottom-up parsing algorithms to produce a hybrid parsing automaton. Consequently, we can generate more powerful deterministic parsers—up to and including LR(*k*)—without incurring the combinatorial explosion that makes canonical LR(*k*) parsers impractical. The range of practical parsers is further extended to include GLR, which was originally developed for natural language parsing but is shown here to also have advantages for static analysis of programming languages. This emphasis on conformance to standard grammars improves the rigour of static analysis tools and allows clearer definition and communication of derived information, such as metrics.

Beneath the syntactic structure of software (exposed by parsing) lies the deeper semantic structure of declarations, scopes, classes, methods, inheritance, invocations, and so on. In this work, we present a new tool that performs semantic analysis on parse trees to produce a comprehensive semantic model suitable for processing by other static analysis tools.

An XML pipeline approach is used to expose the syntactic and semantic models of the software and to derive metrics and visualisations. The approach is demonstrated producing several types of metrics and visualisations for real software, and the value of static analysis for informing software engineering decisions is shown.

Keywords: Static analysis, parsing, parser generators, GLR, Tomita, semantic analysis, software metrics, software visualisation, XML pipeline, visualisation pipeline.

Chapter 1

Introduction

1.1 Overview

This thesis is concerned with tools and techniques for acquiring and delivering information about the static structure of software, in order to help software engineers understand and improve their products. It introduces a rigorous and comprehensive approach to collecting and modelling information about software structure, describes a working implementation, and presents some example applications.

Because software is written in a programming language 'code', a necessary task of a static analysis tool is to decode the source by parsing. The resulting parse trees expose the syntactic structure of the software, making this information available for further analysis. Parsing of programming languages is a thoroughly researched field of computer science, but the practical issues of applying parsing theory in software engineering applications have been less comprehensively addressed. We take steps to remedy this by presenting an improved parser generation approach that has significant advantages in power and flexibility over conventional parser development practice.

Using the syntactic information from parsing, semantic analysis tools can discover semantic concepts and relationships in the software, such as classes, methods, inheritance and invocations. We have developed a semantic modelling tool for Java programs. It provides a more

comprehensive and accessible model of software structure than is available from other tools. For example, the model resolves invocations of overloaded methods, describes the entire scope structure, and relates semantic features to syntactic ones.

By exposing all the features of a program in a model, the software's semantic structure is made available for processing by other static analysis tools, which may manipulate the data in a variety of ways in order to provide helpful information to software engineers. Such tools may calculate metrics, produce visualisations, test compliance with design heuristics, or even form the repository of an Integrated Development Environment (IDE), among other applications. We present several examples of such tools.

A pipeline architecture provides the means of integrating our static analysis tools. Source code enters the pipeline, is parsed, semantically analysed, and further processed by metrics, visualisation or other tools. At each stage, the data is represented in XML, so it may be viewed, stored, modified, or re-processed. The result is a flexible and transparent mechanism for developing and experimenting with techniques that enhance understanding of software.

1.2 Understanding software

Industrial software is commonly extremely large and complex, and subject to continuous evolution. These factors combine to make software hard to understand, and consequently hard to develop and maintain. A single program is too complex for a human to comprehend in its entirety.

Understanding complex software requires more than comprehending the behaviour of lines of code. It involves the construction of a mental model of the software structure—its components and their inter-relationships—and an appreciation of the *design forces* acting on this structure. Software designers must weigh multiple competing influences as they seek to optimise software attributes, such as simplicity, understandability, efficiency and generality. In doing so they apply a rich—although perhaps imprecise and conflicting—set of design principles, heuristics, strategies, patterns and idioms. Software engineers speak about how a de-

sign *feels* or how code *smells* [34], reflecting the inexact nature of how they judge a design's quality.

The task of understanding software and the task of designing software are linked inextricably: designers seek to make code understandable, and comprehension enables design. Both are facets of what is arguably the central theme of software engineering: managing the complexity of software. The fundamental design principles of software engineering—ideas such as clustering, encapsulation, abstraction and information hiding—address complexity by decomposing systems into more manageable units. This decomposition relieves the software engineer of the impossible task of understanding a program in its entirety, by providing localised *neighbourhoods* for specific development tasks.

The success of decomposition techniques inevitably depends on the degree of independence of the resulting pieces. Effective decomposition enables software engineers to understand and change some region of interest, without unexpectedly disrupting other parts. Despite the efforts of software engineers, in practice software neighbourhoods often lack clearly defined boundaries; the possible impact of a code change can be diffuse and difficult to predict. Neighbourhoods span multiple levels of abstraction; typically, a software engineer must know the system architecture at a high level of abstraction, a number of source code sections precisely, and between these extremes related software features at various intermediate levels of detail, depending on their proximity to the changing code. Determining proximity is itself not straightforward, as the *closeness* of software components has many dimensions. For example, a statement might be lexically proximate, such as when code resides in the same scope, it could be in a direct or indirect client relationship through method calls, or it could have a temporal relationship, such as when a state change influences program behaviour.

Unsurprisingly, principles that encourage decomposition are at the core of software engineering theory. These include, for example, *cohesion* and *coupling* [108], *information hiding* [85], and object orientation with its associated body of design principles [93], for example. These concepts are now so entrenched in software design culture that it might sometimes be forgotten that they are not ends in themselves. They are valuable because they help software engineers to manage complexity by decomposition, making a system more understandable and amenable to change.

Our interest in addressing the task of understanding code is to improve the process and tools by which software engineers acquire the information needed to build relevant, accurate mental models of their programs and use them to inform software design and implementation.

1.3 Static analysis

Software engineers need to understand the structure of software in order to change and extend it. In current software development practice, this process relies almost entirely on the expertise of the software engineers, who work directly with the raw material of software—the source code—and perhaps with supplements such as design documents and UML diagrams [33].

Reading of source code is most efficient for tasks that require understanding of a small portion of code at the highest level of detail, such as when editing that code. For larger or more remote portions of a software neighbourhood a degree of abstraction is warranted and only certain features of the code will be relevant to a developer's decision making. UML diagrams and other system models can be very valuable for this purpose, but are limited in the types of information they represent, and like source code, they show only a prescribed view of the software. Further, source code and diagrams offer no guidance on the application of sound design principles, other than by enforcing constraints built into the programming language or formal notation.

Static analysis tools can help. Static analysis involves examining software artefacts—usually, but not exclusively, source code—to glean relevant information. For example, source code checkers such as *lint* or enhanced compilers (e.g. *gcc –wall*) provide programmers with feedback on programming constructs; for instance by checking for code reachability and uninitialised variables. Source code editors typically colour syntax and complete method invocations as they are typed, and code browsers locate declarations on demand. Some development environments go further, deriving UML diagrams from code, calculating metrics and auditing code against constraints. More experimentally, software visualisation tools seek to present aspects of software structure through visual metaphors.

Despite the prevalence of such tools, we suggest that much of the potential of software analysis to aid software engineers remains unexploited. Software metrics, for instance, are little used in mainstream software development practice, with some exceptions such as counting Lines of Code (LOC). It might be argued that software metrics and other static analysis measures are not more widely used because they have little to contribute to engineers' decision-making. While this argument reflects current practice, it is not a convincing limitation. Although most of the decision-making process of a software designer cannot be automated, static analysis tools can aid the designer by providing relevant information that would otherwise have to be gleaned manually from the program, or might even have gone unnoticed. Metrics and visualisations can illuminate software neighbourhoods at appropriate levels of detail. Design characteristics, such as levels of coupling, can be automatically calculated and violations of design constraints can be detected.

Object-oriented software employs a significantly richer semantic model than procedural software. Consequently it involves the designer in a broader range of design considerations. Over time, the object-oriented software development community has produced an assortment of principles, heuristics and patterns to guide design. Many of these rules are expressed as maxims in the vocabulary of software engineers, such as the *Acyclic Dependencies Principle* [69] [84], the *Law of Demeter* [65], and *Separation of Concerns* [26]. Some, such as the *Liskov Substitution Principle* [67], have formal definitions, while others, such as *Do the Simplest Thing That Could Possibly Work* and *Model the Real World*, defy precise definition and are necessarily fuzzy and subjective.

In many cases static analysis can support targeted investigations of the software to help software engineers make design choices in the presence of competing influences. Objective rules, such as the *Acyclic Dependencies Principle* (which warns against cyclic package dependencies) can be checked, with transgressions automatically flagged for the designer's consideration. Investigation of subjective rules, such as *Separation of Concerns*, can be supported with metrics such as *Lack of Cohesion in Methods* (LCOM) [9] and by enhancing the engineer's view of software structures and neighbourhoods. This highlights features that exert design forces, violate heuristics or emit code smells.

Information from static analysis tools can be used to augment and complement traditional perspectives of software. For example, a 3D visualisation might appear alongside a UML

diagram, or metrics might be used to decorate conventional source code and diagram views by colouring tokens.

We suggest that the case for wider use of static analysis information is strong, but that tools for acquiring high quality data have been lacking. In this work, we have addressed this obstacle by developing new static analysis tools that expose the structure of Java programs by automatically building a model of their syntactic and semantic structure. The information in the resulting model is available to software development tools for processing and presentation. This can then help engineers to better understand the relevant software and improve their designs.

Improved parser generation is a central theme of this thesis. We have found that the choice of parsing algorithm has a profound influence on the complexity of the entire source code analysis—not just syntax analysis but also lexical and semantic analysis—and also influences the quality of the metrics and visualisations that may be produced from the resulting model.

Finally, we note that the term *static analysis* encompasses a great variety of approaches, and consequently has different connotations across research communities. The *IEEE Standard Glossary of Software Engineering Terminology* [47] defines static analysis as "The process of evaluating a system or component based on its form, structure, content, or documentation." Our work falls entirely within this definition, but we do not mean to imply we address all types of static analysis. On the contrary, our main focus is on tasks at the technical kernel of conventional static analysis, particularly parsing and semantic analysis of source code. *Semantic* analysis is itself an overloaded term; we use it in the conventional sense encountered in parsing and compiler textbooks[1].

[1] For example, Aho et al. [1], p.8 say: "The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase."

1.4 Outline of thesis

The main contribution of this thesis is an improved approach to the use of static software analysis for informing software engineering tools and ultimately software engineers' decisions. This is achieved by new parser generation technology better suited to the requirements of static analysis, by building a comprehensive semantic modeller of Java programs (including resolution of overloaded method calls), and by using an XML pipeline to support transparent, unconstrained manipulation of the resulting models in order to derive and communicate information that is relevant to many software engineering tasks. The rest of this document is structured as follows.

- Chapter 2 provides additional background on existing static analysis technology and motivates the improvements made in this work. In particular, it highlights the problems of conventional parsing, explains the need for semantic models, and discusses the use of the resulting data to derive metrics, visualisations, and other feedback for software engineers.

- Chapter 3 addresses parsing in more detail and provides an incremental example of the LR parser classes. This sets the context for the improvements described in the following chapter.

- Chapter 4 introduces a more flexible parser generation approach and describes yakyacc; an implementation of this approach.

- Chapter 5 addresses semantic modelling of Java, as implemented in JST.

- Chapter 6 applies yakyacc and JST to the goal of measuring and visualising Java programs.

- Chapter 7 draws conclusions and discusses some expectations for future work.

Chapter 2

Existing static analysis technology

This chapter describes conventional static analysis, and places it in the context of helping software engineers to understand software. The usual decomposition of static analysis—into scanning, parsing and semantic analysis—is described, and the limitations of existing tools are noted. The goals of this research are defined in the light of these limitations.

2.1 The phases of static analysis

Source code represents software as a linear sequence of symbols. Its syntactic and semantic structure is implicit in this linear sequence. *Static analysis* is the process by which we extract models from source code, make its features explicit, and then investigate those features. This process is shown in Figure 1.

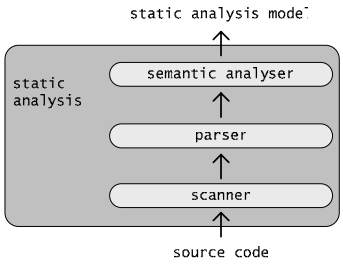

Figure 1: Static analysis phases

By convention, static analysis is decomposed into the simpler tasks of:

- *Scanning* (or *lexical analysis*), which makes lexical structure explicit by breaking source code into a stream of *tokens*.

- *Parsing* (or *syntactic analysis*), which makes syntactic structure explicit by grouping tokens to form parse trees, according to a *grammar*.
- *Semantic analysis*, which makes semantic structure explicit by examining parse trees to discover semantic entities and their relationships.

We describe these three tasks as the *phases* of static analysis, to indicate their logically sequential nature; although, as explained below, they usually run concurrently in conventional static analysis applications [1]. Examples of the inputs and outputs of phases are presented in Section 2.1.1.

The archetypal static analysis tool is a compiler front-end, which scans, parses and semantically analyses a source file before generating intermediate code [1]. With the exception of code generation, the requirements of a compiler front-end largely coincide with our goal of extracting structural information. This is unsurprising, as a compiler must acquire comprehensive information about software structure in order to derive an executable program from source code.

The work presented in this thesis follows the conventional decomposition of static analysis into scanning, parsing and semantic analysis. However, it differs from existing compiler technology in its intent, with significant consequences for the implementation. Compilers perform static analysis for the specific purpose of generating executable programs; i.e. the static analysis model remains internal to the compiler and typically consists of a *parse tree* (or an *abstract syntax tree* that approximates a parse tree) and an associated *symbol table* (represented by data structures tailored to the needs of the compiler). There is no requirement that the entire model for a program, or even for a source file, be present at any one time. In contrast, our static analysis tools produce a general-purpose model of the whole program and expose it for use by software engineering tools.

More specifically, our work differs from conventional compilers' static analysis in the following ways:

- A particular emphasis is placed on producing a model that is defined in terms of the standard description of the programming language; i.e. a standard (or otherwise definitive) grammar and its associated semantic description, (The Java Language Specification [36], for example). This allows the model, or information derived from it, to

be interpreted by any software engineer familiar with the language definition. This contrasts with conventional compiler construction practice, in which parsers are developed by modifying standard grammars until they conform to the constraints of a particular parsing algorithm, often LL(k) or LALR(1) [39]. Grammar modification has the unfortunate side-effect of producing parse trees that describe syntactic structure in some non-standard way. This difference is explored further in Section 2.2.

- The semantic analyser in a compiler front-end is concerned not only with finding semantic concepts and connections, but also with checking that all semantic rules of the language are followed. For example, a compiler must check that inheritance relationships are acyclic and that abstract classes are not directly instantiated. Our objective is to model software already known to be compilable, so some semantic checks are unnecessary; we can simply assume that such checks have already been made by a normal compiler. Although this simplifies the semantic analysis task in comparison to that of a compiler, the difference is less than might be expected since we aim to expose full structural information to downstream tools. Our semantic model needs to be sufficiently rich that, in principle, such checks *could* be performed. We note also that the task of static analysis itself requires a comprehensive model of the type, scope and naming system of the relevant language in order to support correct lookup of names and, in particular, to resolve calls to overloaded methods.

- A conventional compiler processes one compilation unit at a time, and so performs semantic analysis on individual compilation units. A linker or dynamic loader later makes connections, such as method invocations, between the separately compiled units. In our approach we create a single semantic model that spans an entire program, including all semantic connections. Figure 2 depicts the combination of inputs from multiple parse trees, resulting in a single semantic model.

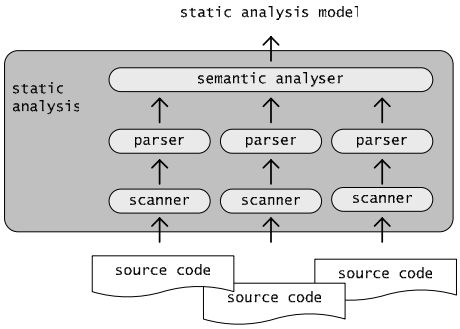

Figure 2: Combined semantic analysis

Compilers are not the subject of this research; they are mentioned here to provide a familiar architectural model for static analysis. We conclude our discussion of compilers by noting

that our improvements to parser generation can be beneficial for compiler construction, in the same ways they benefit other static analysis tasks; see Chapter 4 for details. Similarly, a compiler could make use of our semantic model.

Finally, it is worth noting that the separation of source code analysis into the simpler tasks of scanning, parsing, and semantic analysis is not fundamental; it is a convenience based on the capabilities of the technologies typically used for these tasks. Scanners perform a relatively easy job, and typically use a Deterministic Finite Automaton (DFA) derived from a set of regular expressions. Parsing is more difficult than scanning and usually requires a more powerful technology, such as a Push-Down Automaton (PDA) derived from a grammar. No context free parsing technology, however, is sufficiently powerful to recognise all the structural features found in modern programming languages. For example, no parser (derived from a context free grammar) can ensure that identifiers are declared before their use, or that actual parameters match formal parameters. Consequently, these checks are delegated to a semantic analyser.

2.1.1 An example of static analysis phases

This section gives a very simple example of the inputs and outputs of the three static analysis phases for the tiny Java example in Figure 3. As is usual in source code, indentation informally (and redundantly) displays one aspect of program structure—syntactic nesting—to a human reader. However, the program could instead easily have been collapsed into a single line.

```
class HelloWorld {
    public static void main(String args[]) {
        System.out.println("hello world");
    }
}
```

Figure 3: A simple Java program

A scanner breaks source code into a stream of tokens; as shown in Figure 4. Each token represents a lexical unit found in the input. The token type (in **bold** in the figure) is a symbolic identifier rather than a string. For most tokens, including keywords and punctuation symbols, the original string in the source code can be inferred from the token type alone. In this example the `class` token indicates that the string

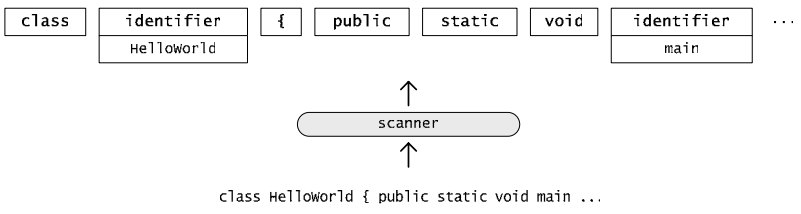

Figure 4: Tokens produced by a scanner

"`class`" was found in the input, and that those characters did not form part of a longer token. In contrast, `identifier` tokens do not correspond uniquely to source code strings, and therefore must carry their original source strings (e.g. "`HelloWorld`", "`main`") as additional information, to be used later by the semantic analyser.

For syntactic purposes white space and comments serve only to separate tokens, and are filtered out before parsing. However, these lexical features influence some metrics such as LOC and comment frequency. As explained in Section 4.3.1.2, in our scanners we can chain tokens together to form a sequence that includes white space tokens (including newlines) and comment tokens. In this way the full lexical structure of a source file is preserved in the token list, even though white space and comment tokens will not participate directly in a parse tree.

In this research, the lexical analysis approach is conventional; we use widely available tools such as Flex [79], Java's regular expression library, or simple hand-coded scanners. We do not address lexical analysis further, other than to expand on the implications for scanners caused by our parsing approach. In conventional compiler architecture, lexical analysis includes the need for the scanner to populate a symbol table, in order to simplify the subsequent task of parsing. Ideally, this is done in a way that keeps the scanner independent of the parser. The parser, on the other hand, is inherently dependent on the scanner. In some cases, however, information discovered during parsing must be fed back to the scanner in order to modify the scanner's behaviour to prevent parsing ambiguities. This creates a cyclic dependency between scanner and parser. For example, this feedback loop is used to avoid ambiguities in parsing the `typedef` syntax of C programs, where it is known as the *lexer feedback hack*. Our use of stronger parsing algorithms eliminates the need for such a close coupling between scanner and parser, as will be discussed in Chapter 4.

A parser exposes the syntactic structure of a program by mapping a stream of tokens into a parse tree. Figure 5 shows a fragment of a parse tree constructed from the tokens of the example in Figure 3. A parse tree groups together adjacent words or phrases to form longer phrases, until a sentence that spans the entire input is found. The legal groupings, and consequently the possible structures of parse trees, are defined by a grammar. See Figure 12 on page 42 for an example.

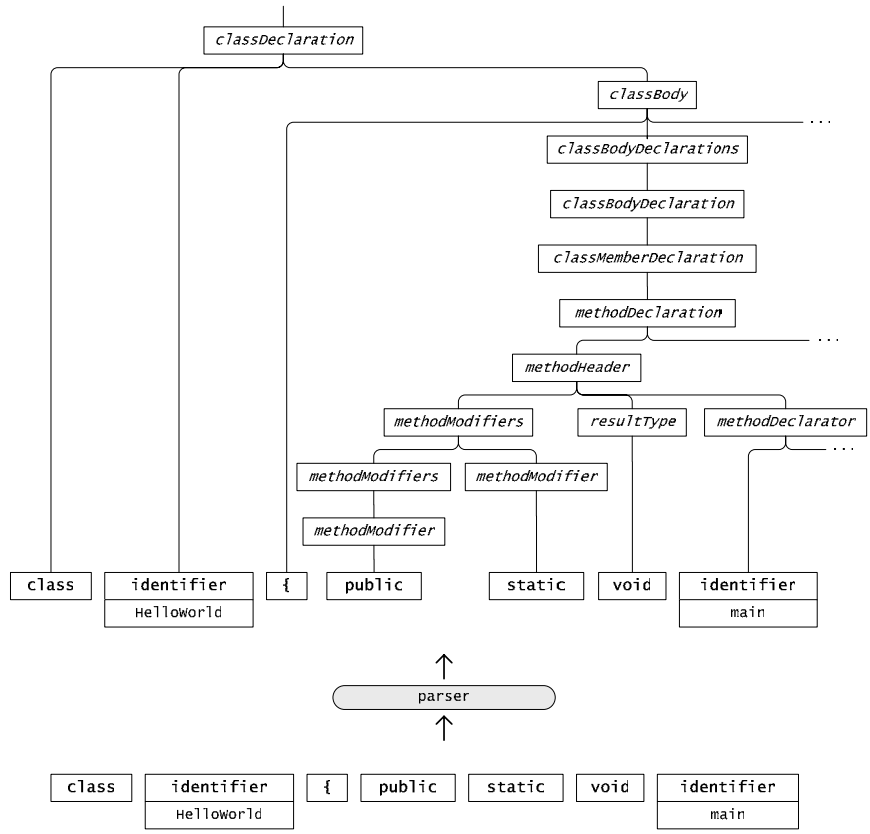

Figure 5: Parse tree fragment produced by a parser

Parsers can of course be developed manually, but this is a time-consuming process for grammars with the complexity of typical programming languages. Instead, such parsers are commonly created by using an automated *parser generator* (yacc [54], for example). As explained in Section 2.2, however, current parser generation practice is not without problems, for which this thesis proposes some solutions.

Semantic analysis identifies the semantic entities in parse trees and determines the relationships between them. These semantic entities are instances of concepts defined by the programming language, such as classes, methods and variables. Relationships between them include, for example, inheritance, invocation and containment. In Figure 6 a simplified semantic model produced by a semantic analysis of the example parse tree is shown. The main structural elements of the program are represented as objects in the semantic model. In this case, the model shows a class called `HelloWorld`, that contains a `main` method that accepts a parameter called `args`. The type of this parameter is represented by an object (not shown in the diagram) that describes an array of `String`s, which in turn references an object that describes a `String`.

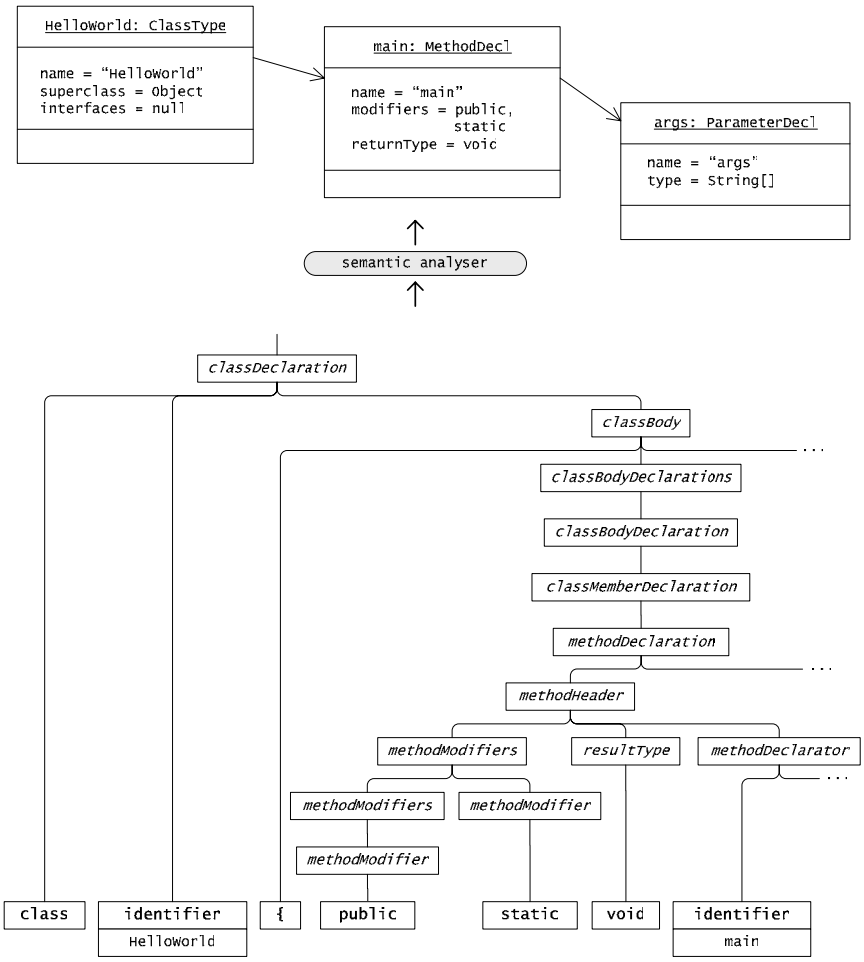

Figure 6: Semantic model fragment produced by a semantic analyser

Unlike a parse tree, which represents the syntactic structure of a single source code translation unit (file) in a strictly hierarchical fashion, the semantic model represents an entire program as a directed graph of connections between semantic entities. If our example program were more elaborate, the model would show additional semantic connections that are not evident in the parse tree. For example, if the `main` method invoked another method, perhaps in another source file, the invocation relationship would be explicit in the model, whereas it is not explicit in the parse tree. In a similar way the usages of variables are connected to their declarations and the declarations are connected to their types. Semantic analysis of Java is addressed in Chapter 5.

Once lexical, syntactic and semantic analysis have all been performed, the complete static structure of software has been exposed in a model and is available to software engineering tools, including those that calculate metrics or construct visualisations.

This separation of static analysis into three phases is a powerful convention for controlling the complexity of the task. Each exposes more information by emitting a more elaborate data structure: scanning produces a linear data structure, parsing produces a tree, and semantic analysis produces a graph. In conventional approaches this clean separation of concerns is often compromised to some degree by the need for all static analysis phases to access a symbol table. It breaks down when a language is too complex for the parsing technology employed. More powerful parsers avoid this coupling. This will be discussed in Chapter 4.

2.2 Conventional parser development

Parsing is a thoroughly investigated area of computer science, and parser development is a commonplace software engineering activity. Consequently it might be assumed that little room for improvement to established parser development practice remains. We suggest, however, that despite its firm academic foundations, the practical issues of parser development for software engineering purposes have been less well addressed. This section gives a simplified introduction to common parser development practices and the parsing problems addressed in our research. A more thorough discussion of parsing technology is presented in Chapter 3. In this section we refer to several parser classes—LL(k), LALR(1), LR—with minimal explanation; the reader requiring more parsing background is referred to the next chapter.

A language is defined by a grammar. The grammar also implies the structure of parse trees for sentences in that language. A grammar therefore contains all information needed to develop a parser for that language.

A parser may be developed manually, by writing a program that builds parse trees in a fashion consistent with the grammar. For example, a parser hand-coded in C appears in Figure 7 (top). Hand-coded parsers often use a *recursive descent* parsing algorithm (see Chapter 3), because the resulting code directly reflects the grammar and is simple enough to be understood by humans, even for complex languages. Unfortunately, the inherent weakness of this parsing algorithm means that relatively few grammars can be mechanically implemented as a recursive descent parser. For more complex grammars—including those of many actual programming languages—a substantial amount of inventiveness is required on the part of the parser's authors in order to convert the given grammar into recursive descent function calls. As a result the parser may bear little resemblance to the original grammar.

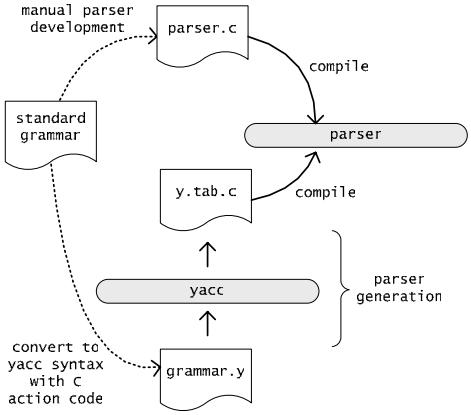

Figure 7: Manual and automatic parser development

The alternative to manual parser development is to use a parser generator, such as yacc (Figure 7, bottom) [54], bison [27], or ANTLR [87]. Yacc, as is typical of parser generators, requires an input file (`grammar.y` in the figure) that:

- Specifies the grammar in its own dialect of Backus Naur Form (BNF; see Chapter 3).
- Contains C action code (embedded in the grammar) that will be executed when the generated parser runs and recognises grammar productions. This user-supplied code typically builds a parse tree.

The output of yacc is a C source file (`y.tab.c`) that implements the parser, including the supplied C action code.

Automatic parser generation enables the use of more elaborate parsing algorithms, unconstrained by human cognitive capabilities. Even so, the set of grammars that can be accepted by conventional parser generators such as yacc is limited, and unmodified standard gram-

mars are likely to fall outside it. The size of the set of grammars accepted is a function of the power of the parsing algorithm (LALR(1), in the case of yacc) used by the parser generator.

A great variety of parsing algorithms of varying powers have been developed (Chapter 3 gives an overview), but only a few of these are routinely applied to the task of parsing programming languages. Unsurprisingly there is a trade-off between speed and generality. The measure of speed for a parser is its asymptotic performance against the number of tokens in the input. To be practical programming language parsers must exhibit linear performance, because of the very large numbers of tokens in source files. Consequently mainstream parsing practice has largely avoided slow (polynomial or exponential time) general parsing algorithms in favour of fast (linear time) but more restrictive algorithms. These linear-time parsers have the common characteristic of operating deterministically; i.e. they are both fast and restricted because they can pursue only one line of investigation. Grammars that allow ambiguous phrases are intractable to deterministic parsers. This remains true even if ambiguity is merely an artefact of the limited context used by the parsing algorithm, rather than a fundamental characteristic of the grammar.

A parser that reads tokens in sequence ("left to right") and constructs a parse tree starting at the root and working toward the leaves ("top down") is known as LL (see Chapter 3). Recursive descent parsers are a common implementation. LL parsers are widely used because their simplicity makes them comprehensible to human parser developers. However, this simplicity also severely restricts the set of grammars they can handle deterministically. In other words, the likelihood of an LL parser generator failing to generate a parser for a particular grammar is relatively high.

The most powerful linear parsing algorithms belong to the LR class (see Chapter 3). LR parsers differ from LL parsers in that they build parse trees from the leaves toward the root ("bottom up"). This makes LR parsers fundamentally more powerful than LL, because they have more information available when recognising productions. The child nodes of a parse tree branch are known before the branch itself. Put another way, the parser does not attempt to recognise a production until it has seen all the RHS symbols of that production. This additional power comes at the cost of increased complexity, making the use of a parser generator mandatory. As Grune and Jacobs [39] put it: "the control mechanism of such a parser is so complicated that it is not humanly possible to generate it by hand" (p. 78).

The most powerful deterministic variant of LR parsers is known as LR(*k*), where *k* defines the depth of lookahead used to decide between alternative parser actions [61]. Unfortunately LR(*k*) parsers are widely thought to be impractical, because conventional implementations of these algorithms produce a combinatorial explosion in the number of states of the parsing automaton. The term *canonical* LR(*k*) is used to denote this explosive approach to building LR parsers. In Chapter 4, we show how combinatorial explosion can be avoided and substantiate our claim that LR(*k*) parsers can be made practical. (Other authors have reached the same conclusion by different paths, as we explain in Section 4.4.)

When *k* is reduced to zero, the combinatorial explosion does not occur and the resulting LR(0) parsers are easily constructed, having the same order of complexity as the grammar itself, because the number of states is directly related to the combined size of all RHSes in the grammar. Unfortunately the ability of such parsers to choose between possible parser actions is so much diminished that LR(0) parsers are very weak.

In order to avoid the problems of canonical LR(*k*), simpler and weaker LR variants were devised that nevertheless still surpass the power of LL approaches. Foremost among these are SLR(*k*) [23] and LALR(*k*) [22], which avoid state explosions while still offering a substantial portion of the power of canonical LR. LALR is the more powerful of the two. Even so, values of *k* higher than one are still commonly considered problematic in practice; again because of combinatorial explosions. Each increment in *k* adds a dimension to a table-driven parser. Once again we find that this problem can be avoided, and that SLR(*k*), LALR(*k*) and LR(k) parsers for higher values of *k* can be made practical. Chapter 4 will provide details.

With canonical LR and lookaheads deeper than one ruled out of consideration by programming language parser developers, LALR(1) has long been viewed as the best practical approach. This view is articulated by Aho et al. in their influential textbook [1], p.236: “This method is often used in practice because the tables obtained by it are considerably smaller than the canonical LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar.” As Gough [37], p.255 puts it, “The LALR parsers are the most widely used bottom-up parsers at the present time. They appear to strike an ideal balance between the power of the underlying method, and the complexity of its implementation.”

The dominance of LALR(1) was cemented not just by the prohibitive increase in complexity needed to attain the more powerful canonical LR, but also by the view that the resulting gains in power would be of relatively little practical value anyway. This attitude is evident in the quote above: "most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar" [1]. Gough [37] (p. 262) more directly states:

> It appears that in practice the number of grammar constructs which lead to LR(1) but not LALR properties is very small. The LALR method is thus likely to remain fashionable […].

Elsewhere, Aho et al. [1] state that "LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written" (p. 215). Such sweeping claims for LALR and LR might create false expectations that grammars for real programming languages can readily be expressed within the limits of these parsing classes, and that there is consequently little need for more powerful parsers. This belief, however, does not match our experience in developing parsers for modern programming languages, which may be syntactically ambiguous (and hence intractable to all deterministic approaches) or just very difficult to parse. We suggest that claims for the sufficiency of limited parser classes underestimate the problems of adapting grammars to fit these limits. Perhaps the claimants also underestimate the complexity of programming languages for which parsers would be required; C++ is a telling example. See [48] for an exploration of this topic.

In practice programming language grammars commonly defeat even the most powerful of the linear time parser classes, unless they are specifically designed to comply with the algorithm. Consequently a grammar designed primarily to describe a programming language to human readers is likely to be unsuitable for parsing by the dominant parser generators; LL(k) and LALR(1). As noted by Grune and Jacobs [39], "a grammar that is designed without regard for a parsing method and just describes the intended language in the most natural way has a small chance of allowing linear parsing automatically".

Standard grammars are no exception, as their main purpose is to define and communicate the syntax of a programming language to human readers. The alternative approach of crafting a standard grammar to accommodate a particular parsing algorithm sacrifices simplicity and

comprehensibility for human readers. The solution sometimes adopted (in the Java Language Specification [36], for example) is to provide two grammars, one for human comprehension and another for parsing. This approach, however, does not address the problem that parsing will then not conform to the grammar as understood by humans.

When a given grammar falls outside the capabilities of conventional parser generators, the usual parser development practice is to manually adapt the grammar until it meets the constraints of the chosen parser class, while still continuing to describe the same language as the original grammar (or so it is hoped). This practice of grammar modification has several disadvantages:

- Grammar design is a labour intensive task requiring specialist skills and good understanding of the chosen parsing technology.

- It is difficult or, for some languages, impossible to modify a grammar to meet the limitations of a parsing algorithm without changing the underlying language.

- Parse trees conform to the modified grammar, rather than the original. In other words, the syntax of a sentence is represented in some non-standard way. This complicates the task of any downstream program that relies on the language standard. Semantic analysis in particular is made more difficult, because the semantic rules of the language are normally defined in terms of the standard grammar.

- Manual adaptation of grammars leads to a loss of rigour. Apart from the obvious potential for human error in modifying the grammar, the use of a custom grammar introduces a confounding translation step when describing syntactic features. This is of particular concern in static analysis applications, in which we seek to rigorously calculate and describe metrics, visualisations and other representations of program features. The natural way to communicate syntactic features is in terms of the standard grammar. So, for example, we might describe a metric as the average number of *expressions* per *statement*, where *expression* and *statement* are defined by the standard grammar.

- Reference grammars change as languages evolve and grow. Even small changes can require a disproportionate amount of reworking of a custom grammar derived from previous versions of the original grammar.

For some languages no amount of modification will make a grammar acceptable to a deterministic parser generator. This is always true when the language is inherently ambiguous. In C++, for example, a single sequence of tokens such as

```
s(*b)[5];
```

has three syntactically valid meanings: (1) a declaration of `b` as a pointer to an array of five `s`'s, or (2) a dereference of `b`, cast to `s` and indexed to its fifth element, or (3) an invocation of function `s` with parameter `*b` and the returned value indexed to its fifth element. The ambiguity must be resolved semantically.

More subtly, the language may be unambiguous, yet still fall outside the discriminatory power of a deterministic parsing algorithm. This occurs, for instance, when an arbitrary amount of lookahead is needed to resolve an ambiguity. For example, in a language that allows any depth of nested parentheses, a fixed amount of lookahead is not adequate to differentiate all possible inputs.

For such languages it becomes necessary to work around the weakness of a deterministic parser in more intrusive ways:

- One strategy is to construct a parser from a grammar that approximates the language with some superset of the legal syntax, and requires the semantic analyser to transform the parse tree to the expected form (while catching any invalid syntax), or in some other way to accommodate the modified language. The gcc compiler [98], for example, uses this strategy.

- Another approach is to protect the parser from ambiguity by feeding syntax information from the parser back to the scanner, so that the scanner may change the types of tokens it produces and thus avoid ambiguity. We earlier noted an example of this approach known as the *lexer feedback hack*. This workaround is viable only when the disambiguating information is available in the parser.

- In cases where semantic disambiguation is necessary, feedback is required from the semantic analyser to the parser. For example, a Java parser (from a grammar such as the exposition grammar of [36]) cannot tell from syntactic information alone whether an identifier should be reduced as a `className` or an `interfaceName`. This information requires a semantic analyser that can model and interpret the scope structure, inheritance hierarchy and lookup rules of the language.

In the case of a complex language such as C++, it may be necessary to combine all these strategies together. The result is a severe breakdown of the decomposition of static analysis: scanning, parsing and semantic analysis become heavily coupled and very complex. We have found that these difficulties can be entirely avoided by the use of more powerful parsing techniques, including GLR—a nondeterministic version of LR—which was originally developed for parsing natural languages. We explore the application of GLR to the problem of parsing C++ in [48].

This thesis presents a solution to these parsing problems, in the form of a more adaptable parser generator. Rather than adapting a grammar to a chosen parsing algorithm, our parser generator adapts the parsing algorithm to the given grammar, generating a parser that exhibits linear or near-linear performance on actual source code. As a consequence we can parse source code according to its definitive grammar, so that the resulting parse trees support further static analysis—including semantic analysis, metrics calculation, and visualisation—in terms that conform to the language standard. Chapter 4 explains how this is achieved.

2.3 Semantic modelling of software

A programming language definition ascribes semantics to the syntactic constructs of a programming language. Classes, inheritance, methods, method invocations, types, scopes and access protection are examples of the semantic concepts of OO programming languages. These concepts are, in effect, the material with which software designers work.

Parsing is concerned only with the way in which tokens are grouped together, and not with the meaning of those tokens. A parse tree therefore does not explicitly show the semantic structure of a program, although semantic features will be evident to some degree in the lan-

guage syntax. A parse tree, however, cannot show the network of connections between semantically related parts of a program, such as the relationship between a variable declaration and its type, defined elsewhere in the program.

A semantic analyser examines parse trees in order to identify the semantic entities in the software and to find the relationships between them. A representation of this structure is a semantic model of the program. Such a model captures the program's underlying structure, as opposed to the superficial structure of syntax. This gives a far richer source of information than is available from parse trees, and this information also corresponds closely to the structural concepts manipulated by software designers.

In order to build a complete semantic model, a semantic analyser must be able to represent the full type system of the language. It must record all declarations, and use scope rules to resolve names. For example, in order to resolve a method invocation in a Java program, the semantic analyser must first determine the scope in which the invocation is made, and then find all visible methods of the given name, correctly traversing the inheritance hierarchy, imported packages, etc. The types of the actual parameters must be determined, a process that may involve analysing arbitrary expression syntax, and these types must be matched against the types of the candidate methods' parameters. Type promotions must be applied, with closer matches favoured over more distant ones. Access rules must be checked to eliminate inaccessible methods. At the end of this process a relationship is added to the semantic model, connecting the invocation expression with the declaration of the method to be invoked, according to the static type structure of the program.

The essential data structure of a semantic analyser is a symbol table. The term *symbol table* evokes the need to look up named symbols (such as types and variables) whenever their names are used in the source code. In modern programming languages, and OO languages in particular, the scope structure and name resolution rules are sufficiently complex that the term *table* is misleading; in fact, name look-up requires a complex graph of data structures that reflect the scope rules of the language. Scopes must correctly reflect inheritance, nested classes, namespaces, access control, overloading, etc. The requirements of such a data structure coincide with those of a comprehensive semantic model; it must fully capture the declarations and scopes of the program. Rather than viewing a symbol table as a useful data struc-

ture for constructing a semantic model, our approach is to recognise them as the same concept. The symbol table *is* the semantic model (and vice versa).

Unlike syntax, programming language semantics are usually described in natural language, and semantic analysers are typically developed manually rather than generated automatically. For this research we manually developed a semantic analyser for Java, following the language specification. As already noted, this process was greatly aided by the use of a parser consistent with the standard grammar. The semantic analysis program works from the same constructs that are used in the language specification.

In Chapter 5, we present the design of a semantic model for Java. In essence, this is primarily a data modelling exercise; we produced an OO model of the concepts represented by the Java type system. Objects in this model describe the packages, classes, fields, methods, blocks and other entities from which the program is made. Relationships between these objects represent inheritance, containment, accesses and invocation, among others. The resulting model provides a richer and more complete representation of the static structure of Java programs than is available elsewhere.

2.3.1 Reflection

Alternatives to our semantic modelling approach include Java's reflection facility. Java's reflection classes, in fact, comprise a semantic model, with the added feature of integrating with the running process. This allows, for example, access to runtime values of fields and execution of methods. Figure 8 shows the main features of the reflection API, omitting runtime-specific aspects. The relationships depicted are logical rather than an exact model of the implementation. Clearly, the major semantic structures of Java programs are represented in this model, and the reflection API therefore goes some way toward achieving the goals of this project.

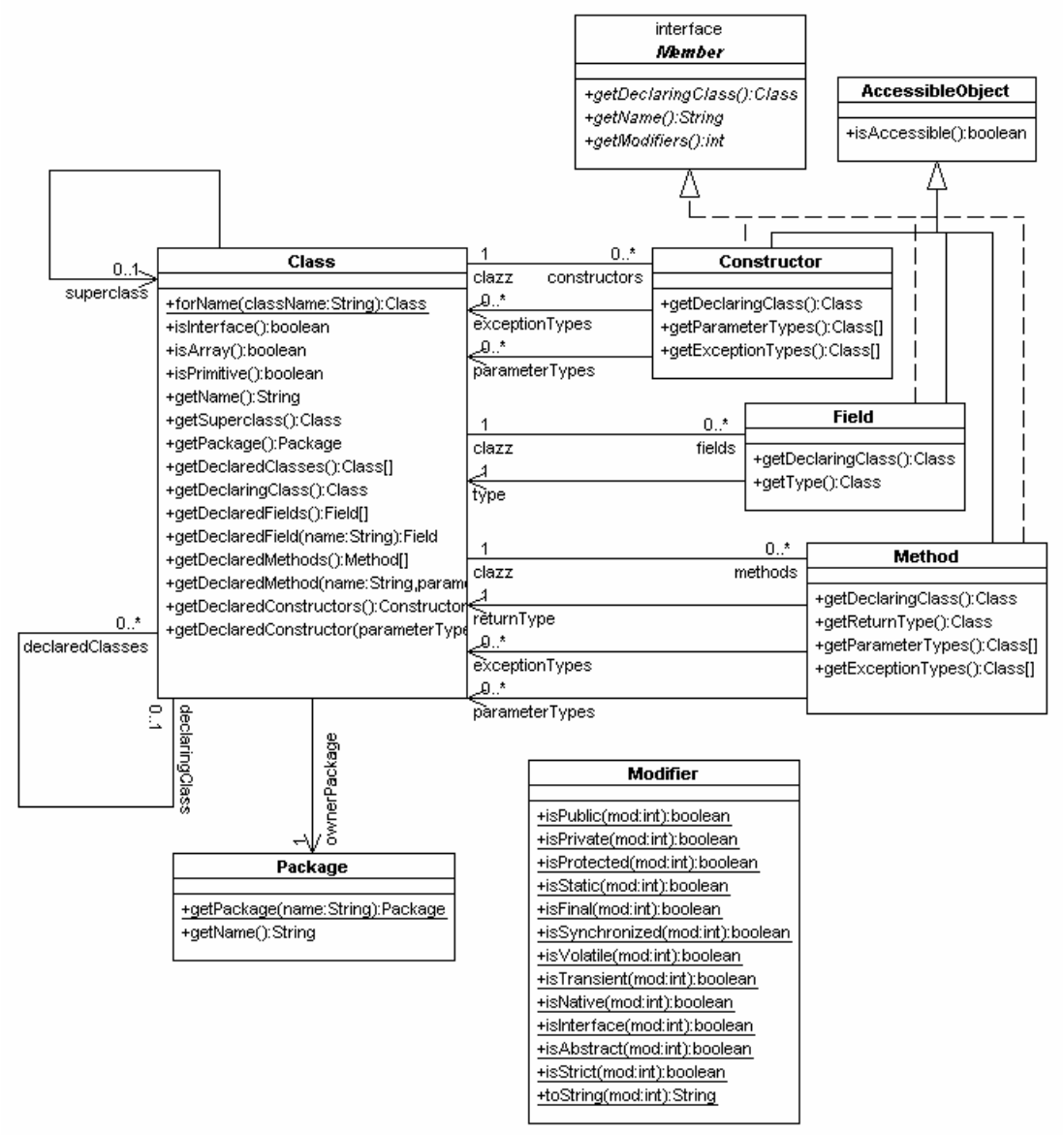

Figure 8: Java reflection semantic model classes

Reflection, however, was not designed as a comprehensive, general-purpose semantic model for Java. Most notably, the reflection model describes method interfaces but does not represent the contents of methods. The concepts of block scopes, local variables and method invocations, for example, are not modelled at all. These internal structures are very important for our purposes, including determining software neighbourhoods, measuring reuse, detecting dependencies, etc. A further limitation of reflection in our context is the degree of model abstraction. Reflection maps the actual semantic elements of a Java program into a simplified, generalised form. For example, ordinary classes, inner classes, interfaces, primitive types and arrays are all described by instances of `Class`, despite the significant variations in

their semantics. This degree of approximation is unhelpful for calculating precise metrics and deriving other information, because a portion of the semantics is suppressed by the model and must be deduced (if possible) by whatever tool needs to use it.

Despite its limitations, reflection provides a valuable supplement to our source code based approach. We use reflection when source code is unavailable. This is often the case for library utilities in typical projects. The semantic analyser builds a full model of classes for which parse trees are available and reflects on `.class` files to build a skeleton model of library software. This is sufficient to show how the software under development relates to existing libraries.

Other tools that expose the contents of Java `.class` files include decompilers and interfaces to support debuggers. Such tools offer more detail than reflection; e.g. by showing the contents of methods. Nevertheless, they are not ideally suited to general semantic modelling and inevitably provide a less faithful portrayal of the source code than a parse tree. More fundamentally, any approach that relies on an intermediate representation of a program (such as `.class` files) will be restricted to languages that use that intermediate format, and to programs that compile without errors. Source code analysis does not share these limitations. In this work, we have restricted semantic analysis to Java, but the approach will also work for other languages. Likewise, this thesis addresses only buildable programs, but a related project extends the approach to work with code that contains errors [51].

2.3.2 Alternative models

We have already noted the parallels between our approach and that of compilers. An even stronger parallel exists with the models employed by some Integrated Development environments (IDEs) to represent the structure of software under development. Eclipse [29] [3] is a well-known example. In fact, our goal of exposing software structure to software engineering tools matches exactly the needs of an IDE, which provides a collection of such tools. An example of our approach being used for this purpose is described in [21].

The open source Eclipse IDE offers a useful basis for comparison with our syntactic and semantic modelling approach. Much of the Eclipse functionality described here was developed

in parallel with this research. Today Eclipse remains under very active development, with a large community of users.

Eclipse provides a framework into which development tools can be plugged. One such plug-in is the Java Development Tool API (JDT), which models Java programs in order to support arbitrary Java development plug-ins, including editors, UML diagram tools, refactoring tools, and others. JDT essentially occupies the same niche as our static analysis model, but also offers additional features to support IDE tasks.

JDT is too complex to describe fully here, but we can outline some relevant features. Figure 9 shows a simplified version of several interfaces in `org.eclipse.jdt.core`, the core Java modelling package. It is difficult to characterise this model as either syntactic or semantic; it shows the main declared features of a program in a semantic style, but only relates them syntactically. By exposing classes (as types), fields, methods and so on, it resembles the model presented by the reflection API (Figure 8), although the model is derived from source code rather than class files. Also, as with reflection, the internal structures of methods such as statements and method invocations are not modelled by `jdt.core`, with the exception of local variables.

Unlike reflection, `jdt.core` does not model semantic relationships between model entities. So, for example, an `IType` object representing a class does not know the `IType` of its superclass, it knows only the superclass name (a `String`). Similarly an `IField` object can provide the name of its declared type, but not the model object. However, some syntactic nesting is modelled, e.g. a class can return all of its `IField` objects. An exception to the rule of modelling only syntactic relationships is made for inheritance. The `ITypeHierarchy` interface indirectly exposes the inheritance relationships between `IType` objects via methods such as `getAllsubTypes(IType)`.

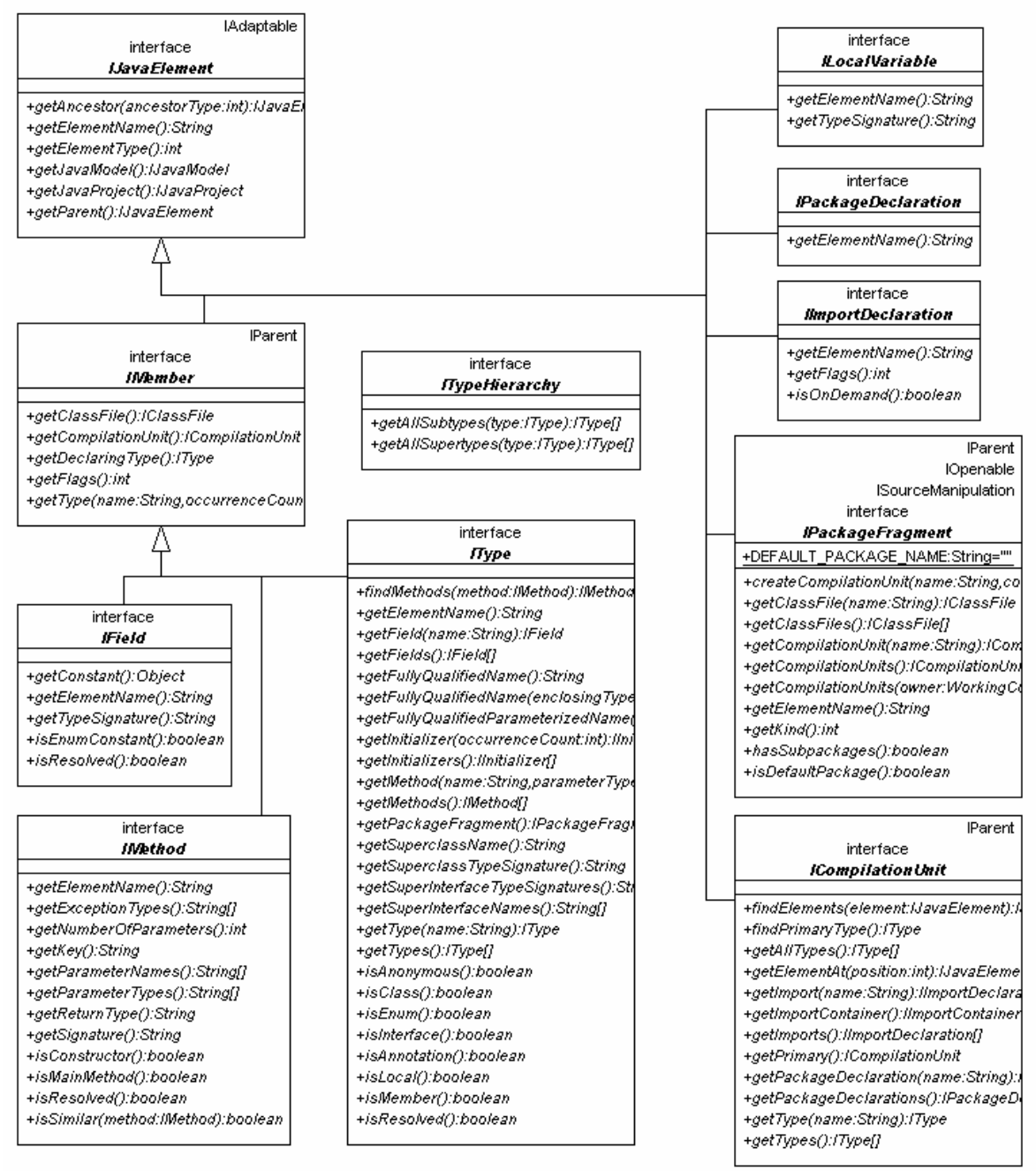

Figure 9: JDT core model interfaces (simplified)

Another package of JDT, `org.eclipse.jdt.core.dom`, defines classes for more extensively modelling the syntactic structure of Java programs. Figure 10 is a much-simplified class diagram that illustrates the main features of the syntactic model (many classes are omitted for clarity). The model is an Abstract Syntax Tree (AST), a data structure that approximates a parse tree without corresponding exactly to any grammar; although it is influenced by the LALR(1) grammar of the underlying parser. Every node in the tree is an instance of

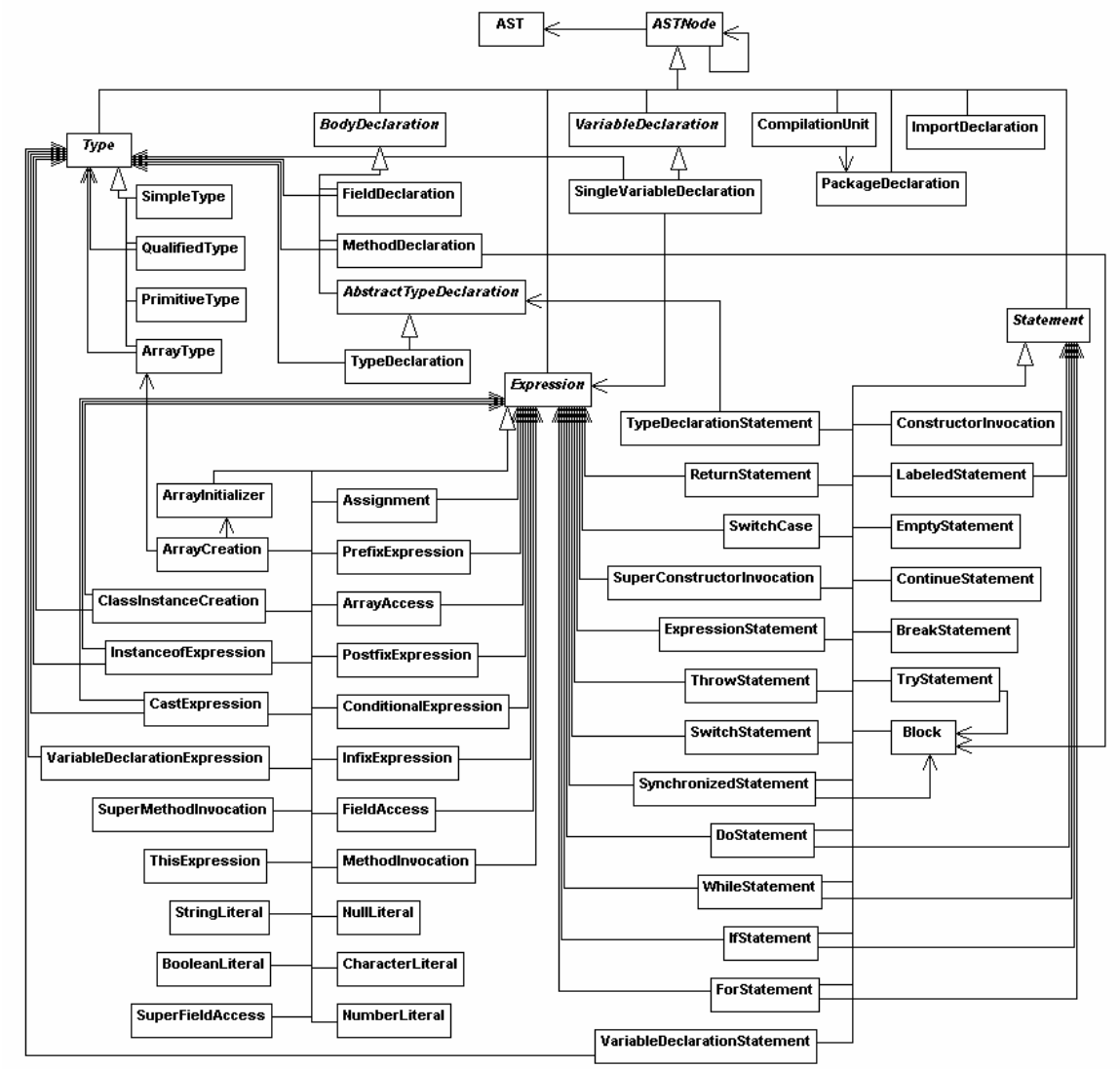

Figure 10: JDT AST classes (simplified)

the abstract `ASTNode` class. Concrete subclasses describe specific syntax, such as a `FieldDeclaration`, a `WhileStatement` or a `MethodInvocation`. Each object in the tree contains subtree nodes of the specific types required. For instance, a `WhileStatement` contains an `Expression` representing the test expression and a `Statement` representing the body of the loop. As can be seen from the diagram, `Expression` and `Statement` are heavily used as abstract nodes, making up the bulk of the tree. The proliferation of subclasses reflects the variety of statements and expressions in Java.

Another heavily used class is `Type`, which records the use of a type, as opposed to modelling the type itself. For example, a `SimpleType` contains (indirectly) a `String` that names a class or interface. The type itself is described by a `TypeDeclaration`.

Semantic connections, such as between a `Type` and the `TypeDeclaration` it implicitly references, are not represented in AST nodes. However, the same package (`jdt.core.dom`) also provides a number of classes and interfaces that model *bindings*, i.e. objects that are described in JDT documentation as containing "resolved information". These appear in Figure 11. Binding objects are, in effect, a semantic model that describes packages, variables, methods and types, and the relationships between them. For example, an instance of `VariableBinding` models a declared variable, including a relationship to a `TypeBinding` object that models the variable's type. In turn, a `TypeBinding` that models a class provides access to the `TypeBinding` of its superclass, the `MethodBindings` of its methods, and so on. In this way binding objects form a self-contained network of semantic entities.

The binding model in Figure 11 is similar to the JDT core model in Figure 9, and (in the latest version of JDT) is in fact connected to it. A binding object can provide the related `IJavaElement` via the `getJavaElement()` method. This allows a translation from the more richly connected semantic model of bindings to the sparser model of software features provided by `jdt.core`.

The binding classes are even more similar to the model provided by Java reflection (Figure 8). Like reflection, JDT's binding model approximates the semantics of Java by abstracting and generalising concepts. JDT's `TypeBinding` corresponds to reflection's `Class`, `PackageBinding` corresponds to `Package`, and `VariableBinding` corresponds loosely to `Field`. `MethodBinding` serves as both `Constructor` and `Method`.

As noted earlier, reflection does not expose internal or syntactic features of semantic objects. JDT improves on reflection by modelling connections between binding objects and the AST nodes that declared them. Any (syntactic) AST node that represents a declaration can provide a (semantic) binding object for that declaration. For example, a `MethodDeclaration` node in the syntax tree has a `resolveBinding()` method that returns an `IMethodBinding`. Similarly, any AST node that references a declared feature can provide a binding object for the referenced entity. `MethodInvocation`, for example, has a `resolveMethodBinding()`

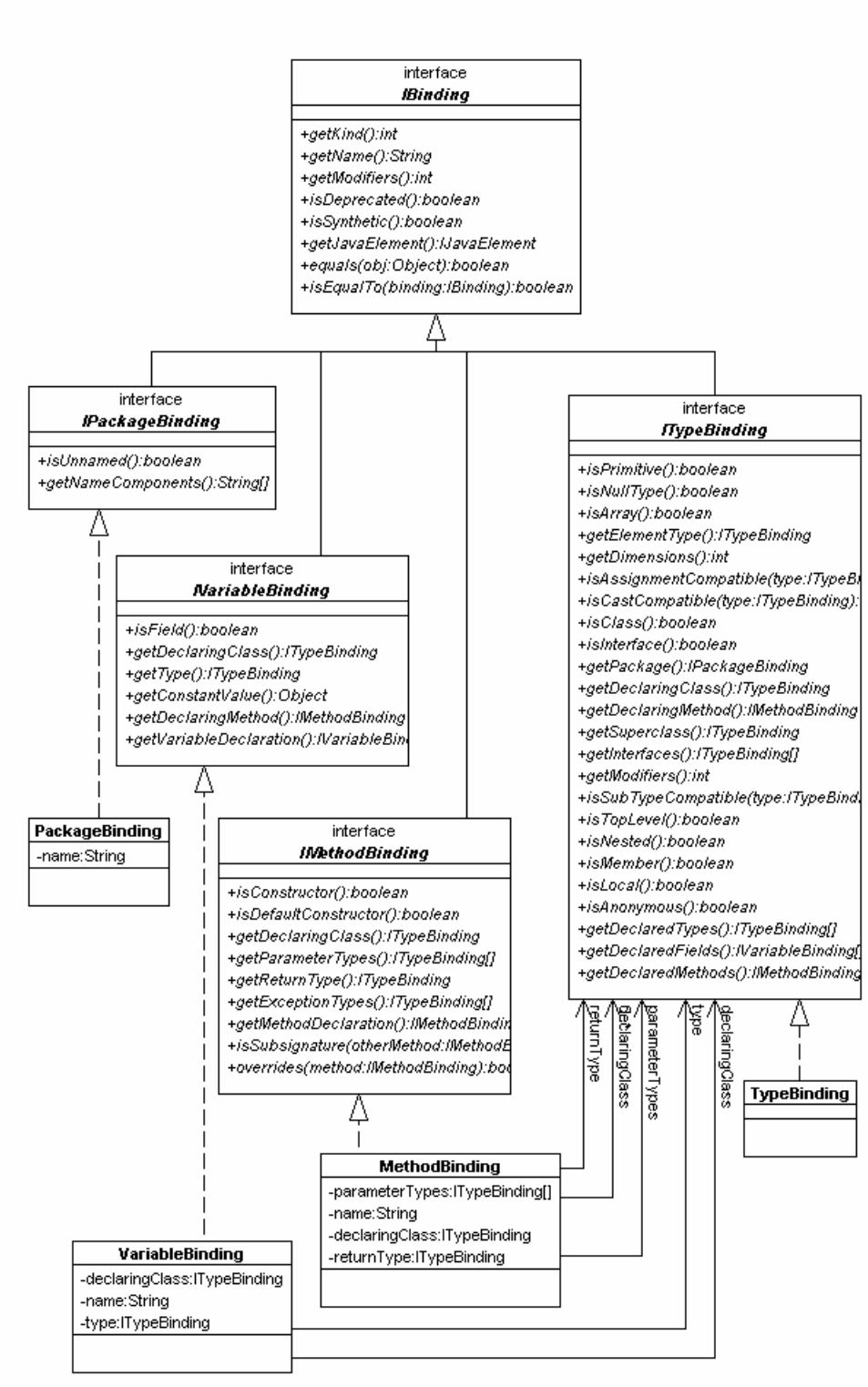

Figure 11: JDT binding classes (simplified)

method that returns a `MethodDeclaration`. Unlike many static analysis tools, overloaded method calls are fully resolved.

Relationships in the reverse direction (from the binding to the syntax tree node) are also available, although less directly. Given a binding object, the AST node containing the declaration can be located by the `findDeclaringNode()` method of a `CompilationUnit`. `CompilationUnit` is the root node of a syntax tree. Of course, only the tree containing that declaration can find it, which means that a caller must first know in which tree to look.

This last point illustrates a significant difference in the semantic modelling approaches of JDT and this research. We build a monolithic semantic model that spans an entire program. Objects in this model are identity objects, so that one semantic concept is represented by one object. In JDT a semantic model is provided as a supplement to a syntax tree. Different trees create different binding objects. In other words, if two syntax trees reference the same semantic entity, different binding objects will be provided for that entity by each tree. The `equals()` method of these objects can be used to determine if they model the same thing. For example, if two classes in separate source files each extend the same superclass, different (but equal) `ITypeBinding` objects will be provided for the superclass by each AST.

We conclude this description of JDT by noting that, although it shares with this research the goal of presenting a static model of Java software, it differs in significant ways in its emphasis and implementation. Generally speaking, JDT is more complex and less focussed on the task of rigorously modelling software structure. Specific differences are listed below.

- JDT is inherently part of an IDE framework, and must participate in an interactive environment of diverse (unknown) tools working concurrently on a changing code base. This constrains the design and use of the model in some ways, for instance by discouraging the construction of a monolithic model for an entire program, in order to keep resource usage in check. It also requires JDT to provide features that are outside the scope of our work, such as the ability to modify syntax trees and to respond to changes made by other tools. In contrast, our model is a stand-alone snapshot of a program, focussed solely on representing the program structure.

- JDT seeks to approximate the syntactic and semantic structure of programs by abstracting and generalising concepts. For example, classes, interfaces, primitive

types, arrays and other types are all represented as instances of the same class. Our approach is to maximise fidelity to the language standard, so that semantic and syntactic structures closely correspond to concepts in the language definition. This is made possible for syntactic modelling by the use of more powerful parsing that uses the standard grammar. In turn, faithful syntax representation enables a semantic model tied closely to the language definition; with one concrete class per concept.

- Clients of the JDT model are tools specifically written to use the JDT API. This API prescribes not only the information available, but the ways in which the model may be manipulated. Our model is exported as an XML file that may be read and processed independently of our modelling tools.

- The JDT model is (somewhat redundantly) decomposed into three facets: the core model, ASTs and bindings. The overall design of the model is syntax-centric; the semantic structure exposed by bindings is supplementary to ASTs. The boundary between syntactic and semantic concerns is blurred, effectively resulting in a hybrid syntactic-semantic model. Our approach has a more precise architecture of two levels: syntax (modelled by parse trees) and semantics (modelled by a semantic model), with a definite separation of concerns. By keeping the levels separate but related, each can be designed to more precisely represent the structure it models. The semantic model is the primary structure, from which parse trees may be accessed. Our semantic model is also more elaborate and comprehensive than that of bindings, so that it is possible to discern more of a program's structure without referring back to its parse trees. For example, our semantic model directly represents all the scopes defined in a program and shows what declarations they contain.

2.4 Informing software engineers

The focus of this research is on the construction of comprehensive syntactic and semantic models of programs, as described in earlier sections. These models are not an end in themselves. Our primary motivation is the desire to better inform software engineers about software structure, helping them to recognise software characteristics, detect and evaluate design forces, and discern software neighbourhoods. In order to perform this role, the software

models must be amenable to further processing, so that other software tools may filter and transform the data to provide specific information pertinent to software engineer's decision-making processes.

In this work we applied our software model to the task of informing software engineers by using model data to derive metrics, visualisations, and other feedback for developers. Our goal in addressing metrics and visualisations is not to propose specific new metrics or visual metaphors (although we do make use of some). Instead, we provide a basis for defining arbitrary software structure metrics and visualisations. We show how to do so in a way that reduces the difficulty of the task and improves the rigour and transparency of the results. The approach is robust and suitable for application to industrial-scale software development.

2.4.1 Software measurement

Engineering disciplines require, by definition, the use of quantifiable approaches. Software engineering theoreticians have defined many metrics, yet quantifiable approaches still play a relatively minor role in mainstream software development practice. Some simple measures are customary for software development—development time and costs, bug counts, for instance—but characteristics of the software products themselves are less commonly quantified. A notable exception is the Lines of Code (LOC) metric. It is arguably the only software metric to be used pervasively, despite its serious flaws as a useful yardstick for development effort, software size or complexity.

Many attempts at defining software metrics with stronger theoretical underpinnings have been made. Halstead's *software science* [40] and McCabe's *cyclomatic complexity* and *essential complexity* [70] are well-known early examples. However, the validity of these theories and the metrics themselves have been repeatedly questioned. The concepts of *cohesion* and *coupling* [108], dating from around the same time, have been received more favourably and have profoundly influenced the field of software design; yet no standard cohesion and coupling metrics have emerged. A great many other product metrics—Purao and Vaishnavi [90] identified over 350—have been proposed, without making the transition into orthodox practice.

One reason for this is the evolution of software technology, and particularly the advent of object-oriented software, which has outmoded some metrics and made applications of underlying theories more troublesome. The most widely-known OO metrics were proposed by Chidamber and Kemerer [9], but these have been criticised by Churcher and Shepperd [16] and others. As Fenton & Pfleeger note, "there is as yet no widespread agreement on what should be measured in object-oriented systems and which metrics are appropriate" [31], p.319.

Despite this unsatisfactory state, the potential benefits of metrics are undiminished, and we argue that there are in fact many valuable metrics that can be of immediate use to OO software engineers. However, we do not attempt to identify the ideal suite of OO metrics in this work. Instead, we make progress towards better metrics, in part by addressing a problem described by Fenton and Pfleeger [31] in the following words:

> When measuring, there is always a danger that we focus too much on the formal, mathematical system, and not enough on the empirical one. We rush to create mappings and then manipulate numbers, without giving careful thought to the relationships among entities and their attributes in the real world.

The entities, attributes and relationships of interest for software structure metrics are exactly those of our static analysis model. We suggest that the under-emphasis of such models in software metrics research is not due to neglect, but largely to the difficulty of building the models properly. In a research setting, some metrics tools make use of heuristic techniques, such as fuzzy parsing, to extract approximate data from for deriving metrics. For example, statements might be counted by scanning for semicolons, or classes might be found by isolating class declaration syntax without parsing surrounding code. We suggest that a lack of rigour in calculating metrics is generally unhelpful, and that such tools are error prone and of limited use in industrial software settings. Measurement of software reuse, for example, is sure to require detection of method calls. To be accurate this requires resolution of overloaded methods, which in turn requires full understanding of the scope and type system of the language. Such involved and comprehensive information is beyond the reach of heuristic tools. In contrast, our approach provides a full and accurate model of the software structure,

described in terms native to the language. This provides a more robust base for deriving metrics.

It might be suggested that the use of metrics is premature in the absence of any cogent theory of software. As Kyburg [63] puts it "If you have no viable theory into which *X* enters, you have very little motivation to generate a measure of *X*". Some metrics researchers have attempted to address this by developing models that map measurable characteristics of software to higher-level concepts of software quality, [71] for example. Our data is a suitable resource for these approaches, but the value of such metrics is likely to remain questionable because of the subjective, multi-dimensional nature of quality. We argue, however, that sufficient theory is already established in the field of OO design to make metrics valuable. Object-orientation itself provides a model of how to structure software, backed by a rich collection of design principles and heuristics. Many of the software characteristics with which these principles and heuristics are concerned are measurable.

In many cases even simple measurements of software characteristics are of relevance to a software designer. For example, consider a programmer editing a method `bar()` of some class `Foo`. Our model could directly supply the programmer (perhaps via some sort of dashboard metaphor) with information such as:

- `Foo` is 3 levels deep in the inheritance hierarchy (Depth In Tree).
- `Foo` has 6 immediate subclasses (Number Of Children)
- `Foo` has 8 subclasses total.
- Variables of type `Foo` are declared in 0 other classes.
- Variables of some ancestor type of `Foo` are declared in 2 other classes.
- `bar()` overrides a method, which in turn implements an interface method.
- `bar()` is overridden in 1 subclass.
- `bar()` is called by 12 other methods, 10 of them in other classes.

These simple measures serve to characterise the software feature under investigation, helping the engineer form an impression of the importance of the method, how heavily connected it is and in what ways, how much of the software neighbourhood must be understood, and how widespread the impact of changes is likely to be.

Many design principles and heuristics suggest more targeted metrics. Minimising coupling, for example, can be encouraged by measuring the number of methods called by `bar()`, while the related *Law of Demeter* [65] (which mandates a single level of dereferencing only) can be backed by counts of violations, or measuring the number of levels of dereferencing. The more fundamental idea of *information hiding* [85] can be supported by counting the number of variables and methods in scope. Many other heuristics, such as those of Arthur Riel [93] can be quantified. Some, such as *minimise the number of messages in the protocol of a class* are straightforward, while others such as *keep related data and behaviour in one place* require elaboration; Chidamber and Kemerer's *Lack of COhesion in Methods* (LCOM) being one possibility [9].

The opportunity for developing more sophisticated metrics is also improved by our model. For example, we have defined ClassRank, PackageRank and MethodRank as a suite of metrics for ranking software entities based on their semantic connections, in much the same way as the Google search engine calculates pagerank of web pages from hyperlinks [77]. These metrics provide a measure of the relative structural importance of software entities in a program, and indicate the topology of software reuse. Without the semantic relationships captured by our model, in this case method invocations and variable accesses, these metrics could not be calculated.

As noted earlier, another research project [21] uses our static analysis model as the core of a repository for a collaborative IDE. One of the tasks of the central server is to automatically detect software neighbourhoods of individual developers and to infer the proximity of other developers. In this setting metrics that describe a software neighbourhood, perhaps by a *Degree Of Interest* (DOI) metric, and quantify proximity are useful.

Finally we note than many tools that support metrics calculation constrain the developer to using a particular API, programming language or custom file format. While it is possible to use our Java model API directly (as the collaborative IDE does), metrics clients need not do so. The use of an XML pipeline architecture provides independence from the way our tools are implemented; clients are dependent only on an XML schema.

2.4.2 Software visualisation

A fundamental difficulty faced by software engineers is the overwhelming volume of information about software that must be assimilated. Source code itself is much too detailed and expansive to efficiently answer the questions of a reader interested in a diverse range of software forces at various levels of abstraction, and in particular software neighbourhoods.

Metrics alone can go some way toward addressing this problem by selecting and condensing relevant information about the software. A coupling metric, for example, might convey an important characteristic of a complex class in a single value. Even so, the sheer scale of industrial software and the large number of dimensions for which pertinent measurements might be obtained means that raw metric data is more likely to contribute to the problem of information overload than its solution. Large tables of values, as produced by some software development tools, are an inadequate means of communicating information to humans. Data volume is not the only problem; such tables also detach the calculated metrics from the underlying software structure. A class complexity measure, for example, is set apart from the source code or UML diagram representing the class itself.

Software visualisation techniques aim to more efficiently communicate information about software to human observers. A visualisation might directly portray aspects such as an inheritance hierarchy, or derived information, including metrics such as code size or complexity. These direct and indirect approaches are complementary and may be combined in a single visualisation. For example, classes in a hierarchy visualisation might also depict class size or complexity, perhaps by varying the colour, size or shape of the class representations. This combined approach allows metrics to be presented as embellishments of the underlying structure being measured, thereby reducing the need to mentally map metrics data onto a suitable conceptual model.

Conventional two dimensional graphs such as histograms, scatter plots and line graphs are very effective general information visualisation techniques, and may readily be produced from our models. For some purposes, two dimensional graphs are well suited to the task of visualising software metrics. Plotting method size against method complexity, for example, can expose over-complicated methods that are candidates for refactoring. In many cases,

however, software metrics present a number of challenges to conventional graphing techniques, including:

- The sheer volume of data. Programs range in size up to tens of millions of lines of source code. Even when clumped into more manageable units such as methods and classes, many thousands of components may need to be visualised.

- The highly multi-dimensional nature of software. Software has many facets of interest to software engineers, including lexical structure, inheritance hierarchies, composition hierarchies, call graphs, dynamic behaviour, control flow, data flow and many others. This multi-dimensionality is reflected in UML, for example, which provides over a dozen diagram types, each emphasising a different aspect of the software. No one model of software can be considered sufficient for full understanding. In order to resolve design forces and apply design heuristics, software engineers must consider many dimensions in combination.

- The extremely non-linear distributions of many metrics. Often metrics data is strongly skewed or clustered, and may contain outliers with extreme values. These effects mean that no single graph is suitable for viewing relationships at all scales.

- The diverse and transient focal points within software that arise as a software engineer investigates design or programming issues. These foci form the centres of semantic software neighbourhoods, in which certain details of the software assume a level of importance greater than similar features more removed from the area of attention. This creates a need for highly dynamic visualisations, in which the level of detail can be adjusted quickly and in a non-uniform way.

In order to address problems such as these, software visualisation researchers are investigating unconventional ways of depicting software. Three dimensional virtual worlds are one such avenue of exploration. In this work we show how our semantic modelling approach may be used to construct experimental 3D visualisations of software structure. Software metrics derived from the semantic model are presented as adornments of the underlying visual representation, in order to convey a rich set of information in a single coherent form.

As noted earlier, our static analysis tools are designed to work in an XML pipeline architecture. We use the term *the visualisation pipeline* to describe our use of parsing and semantic modelling tools in conjunction with metrics filters and visualisation software [50], [13]. This pipeline offers significant advantages for software visualisation research. The benefits of rigorous and comprehensive modelling over more *ad hoc* data acquisition have already been noted. They apply equally to the task of visualisation. The XML pipeline allows use of the model in a very flexible and open environment. The data is saved in a readable form at each stage of its transformation. Any tool that can manipulate XML may participate in the pipeline, and any part of the pipeline can be modified and re-executed as new information is discovered.

Finally, we note the limitations of our approach for some visualisation applications. Our model captures static software structure, and is therefore suitable for software visualisation research that seeks to depict *static* aspects of software. Another main branch of software visualisation portrays the *dynamic* behaviour of software, for which we do not collect data. Static structure models do, however, provide a useful base for dynamic visualisations. Our model is also limited to representing a program at some point in time; it does not attempt to model changes as software structure is developed. A related project [20] extends the model to include time.

Chapter 3

Parsing background

This chapter provides a more thorough background for the subsequent discussion of parsing and advances our main arguments regarding practical parsing in the context of static analysis. Parsing is one of the most venerable subjects of computer science, with a correspondingly vast literature. It is impossible (and unnecessary) to describe the entire field here. Rather, we concentrate on establishing a framework for our contribution. The reader seeking a more general overview is directed to a textbook such as that of Grune and Jacobs [39], which includes an extensive annotated bibliography. A more formal treatment of parsing, covering LL(k), several LR classes (but not LALR) and a number of general parsing methods (but not GLR), is given by Aho and Ullman [2]. The pre-eminent text on LR parsing as it is conventionally applied in programming languages compilers is by Aho et al. [1].

The reader might be forgiven for questioning the need to revisit parsing theory and practice in the context of understanding, measuring and visualising software. We suggest, however, that the practical needs of the software engineering community, and static analysis tool developers in particular, are less well served by the parsing literature and existing tools than they might be. Some of the parsing field's received wisdom is inappropriate for the task of developing parsers for static analysis, and in the next chapter we offer an alternative better suited to this task.

Our main goal in this chapter is to provide an intuitive and accessible exposition of the relevant ideas, rather than detailing the formalisms that underpin parsing theory. The existing

literature is replete with lemmas and proofs, but is generally more concerned with establishing the legitimacy of theories than with articulating practical and applied considerations. Our aim, on the other hand, is to present the concepts in a manner that leads naturally into the objected-oriented implementation described in the following chapter. The remainder of this chapter is structured like this:

- Section 3.1 reviews relevant parsing concepts and terminology.
- Section 3.2 places LR parsing in context within the wider parsing field, and outlines the LR parsing classes.
- Section 3.3 describes a series of example grammars that require progressively more powerful parser classes and shows how these parsers are generated.
- Section 3.4 summarises the chapter.

3.1 Parsing concepts and terminology

This section briefly reprises the relevant concepts and vocabulary of parsing, using examples. Terms being introduced are italicised.

Figure 12 presents a fragment of the Java Language Specification[2] [36] that corresponds with the parse tree fragment visible in Figure 5 (page 13). The complete grammar defines the entire syntax of Java.

```
classDeclaration ::= classModifiers? class identifier super? interfaces? classBody

classModifiers ::= classModifier
                   classModifiers classModifier

classModifier ::= public
                  protected
                  private
                  abstract
                  static
                  final
                  strictfp

super ::= extends classType

interfaces ::= implements interfaceTypeList

interfaceTypeList ::= interfaceType
                      interfaceTypeList , interfaceType

classBody ::= { classBodyDeclarations? }

classBodyDeclarations ::= classBodyDeclaration
                          classBodyDeclarations classBodyDeclaration
```

Figure 12: Fragment of Java exposition grammar

[2] Actually, the Java Language Specification provides two grammars for Java: one is used for exposition purposes, while the other defines the reference implementation of the Java compiler. We use the exposition grammar, as it is fully explained and designed to be comprehensible. Moreover, the language semantics are defined in terms of the exposition grammar.

This grammar—like most programming language grammars—belongs to a class known as *context free grammars* (CFGs). CFGs define *context free languages*. CFGs consist of *productions* (also called *production rules*) that each have a *left-hand side* (LHS) that consists of a single *nonterminal*, and a *right-hand side* (RHS) that consists of any number of *symbols*, where a symbol is either a nonterminal or a *terminal*. We distinguish between terminals, which are found in a grammar, and *tokens*, which are the lexical units that make up a string to be parsed. Each token in the input string corresponds to one terminal in the grammar.

The set of all terminals used in a language is known as the language's *alphabet*. The alphabet of Java, for example, includes the terminals `public`, `class`, `identifier`, `{`, etc. The productions of the grammar define the complete (infinite) set of ways in which terminals may be composed to produce legal *sentences* of the language. So, for example, every compilable Java program is a sentence in the Java language.

The notation commonly used to describe context free grammars is *Backus Naur Form* (BNF) [75]. In parsing literature, many variations of BNF can be found. In this work, we use two:

- In figures in this document, we distinguish nonterminals and terminals typographically, using *italics* and **bold**, respectively, and we group alternative RHSs for any one LHS, using a | sign to separate the alternative RHSs. Figure 12 is an example.

- Raw text documents, such as those supplied to our parser generation tools, adhere to the original style of Naur, in which nonterminals are delimited by angle brackets. A LHS nonterminal introduces each production, followed by `::=` and the RHS. Figure 13 shows some of the productions from the previous figure translated into plain text BNF.

```
<classModifiers> ::= <classModifier>
<classModifiers> ::= <classModifiers> <classModifier>

<classModifier> ::= public
<classModifier> ::= protected
<classModifier> ::= private
<classModifier> ::= abstract
<classModifier> ::= static
<classModifier> ::= final
<classModifier> ::= strictfp
```

Figure 13: Raw text grammar fragment

We use the term *grammar rule* to denote the set of productions that have the same LHS nonterminal, and refer to a particular grammar rule by the name of the LHS nonterminal; for example, we can talk of the *classModifier* rule of the grammar in Figure 12, which describes seven productions. We refer to a particular production within a grammar rule by appending

a subscript with the number of the alternative RHS, for example, the *classModifier*$_3$ production is *classmodifier* ::= **private**.

Extended BNF (EBNF) syntax has even more variability than BNF. Extensions typically allow optional clauses and repeated (zero-or-more or one-or-more) clauses on the RHS. Although EBNF enables more concise grammar descriptions, it does not improve on the fundamental power of BNF and the extensions must be translated (perhaps automatically) to standard BNF before parser generation. The Java exposition grammar fragment in Figure 12 makes use of one feature of EBNF syntax (the ? character) to specify optional symbols. The *classDeclaration* rule translated into non-extended BNF is shown in Figure 14. Section 4.2.1 addresses the use of EBNF input to yakyacc.

```
<classDeclaration> ::= <classModifiers> class identifier <super> <interfaces> <classBody>
<classDeclaration> ::= class identifier <super> <interfaces> <classBody>
<classDeclaration> ::= <classModifiers> class identifier <interfaces> <classBody>
<classDeclaration> ::= <classModifiers> class identifier <super> <classBody>
<classDeclaration> ::= class identifier <interfaces> <classBody>
<classDeclaration> ::= class identifier <super> <classBody>
<classDeclaration> ::= <classModifiers> class identifier <classBody>
```

Figure 14: EBNF grammar rule expanded to BNF

One nonterminal in a grammar is designated as the *start symbol*. In this work, we follow the convention that the first nonterminal (that is, the symbol on the left of the first grammar rule) is the start symbol. The start symbol provides the type of the root of all parse trees for that grammar.

We restrict our attention to CFGs that we define as well-formed for programming language specification:

- Every nonterminal used on the right-hand-side of a production must be defined. That is, it must appear as a left-hand-side of a production.
- Every nonterminal that is defined must be used. That is, it must appear at least once on the right-hand-side of a production, or it must be the start symbol.
- The grammar is not *cyclic*. In other words, it contains no *useless productions*, which directly or indirectly allow a left-hand side to be the same as a right-hand side. Such productions make a grammar infinitely ambiguous: they allow an infinite number of parse trees for a given input.

- All nonterminals must be (eventually) resolvable to terminals. An example of a violation of this constraint is a complete grammar rule *nt* `::=` `t` *nt*. Such a rule can never participate in a finite sentence because then it never resolves to terminals.

A *recognizer* is a program that can determine whether a given sentence belongs to a language; it outputs a Boolean result. A *parser* is a recognizer that constructs a parse tree (or trees) that spans the entire sequence of input tokens. The leaf nodes of a parse tree contain the tokens of the sentence, and the branch nodes contain subtrees that match the RHS of a production. Thus each branch node corresponds to one LHS nonterminal that has been recognised. The root node is a branch containing the start symbol.

A single language may be described by any number of grammars. Typically, a *standard grammar* is used to provide the official definition of a language, but modified versions of this grammar are used for parsing. The C++ grammar used in the gcc compiler [38], for example, differs markedly from the C++ standard grammar [52]. As we have remarked, the need for modified versions of grammars arises because of limitations in the power of the parsing method; the grammar structure best suited to definition and explanation of the language is rarely suitable for the widely used parsing algorithms. Grammars intended to communicate syntax clearly to a human reader often defeat a parsing automaton because of the automaton's restricted context and deterministic behaviour. In general, the weaker the parsing algorithm employed, the more extensive the grammar modification required. The impact of modification of grammars to meet parser limitations was discussed in Section 2.2.

Grammars may be *ambiguous*: a single sentence might have more than one valid parse tree. If all possible grammars for a language are ambiguous, the language itself is ambiguous. Ambiguous grammars are intractable to deterministic parsing approaches, while more powerful parsers tolerate ambiguity and can produce a *parse forest* when multiple valid parses exist. The imagery of a parse forest is somewhat misleading. Ambiguous syntax arises when two or more RHSs with the same LHS match a segment of the input stream. For this particular sequence of tokens, multiple parse trees exist. These trees share the same leaves, and perhaps some branches, but not their topmost branches. However, because the topmost branches match the same LHS and span the same tokens, they are equivalent from the perspective of higher branches in the tree, and can be encapsulated as one ambiguous parse tree node. This means that ambiguities can always be localised within a parse tree branch, and

the tree as a whole will always have one root. The resulting data structure is known as a *packed parse forest*. If it also allows distinct parse tree nodes to share sub-trees when those sub-trees would otherwise be identical copies, it is known as a *packed shared parse forest* [100].

3.2 Grammar classes and parsing algorithms

This section outlines the landscape of parsing classes, in order to put LR parsing in its wider context and explain why LR parsing is of particular interest. It then outlines the main parsing classes internal to LR. Section 3.2.1 begins by briefly portraying the wide view, distinguishing context free grammars from other major grammar classes. Section 3.2.2 returns to our topic of CFGs, and divides context free parsers into those that can handle all CFGs (slowly) and those that can handle a subset of CFGs (quickly). Restricting our attention further to the fast parsers, Sections 3.2.3 and 3.2.4 reprise the two main approaches used for programming language parsing: LL and LR, respectively. The latter section presents the main subclasses of LR.

3.2.1 Context free grammars and others

Like nearly all research into parsing of programming languages, this research addresses only context free grammars because they have adequate descriptive power while remaining tractable to fast parsing algorithms. Here we briefly mention Chomsky's seminal work on grammar classification, in order to place CFGs in their wider linguistic context [10]. Chomsky defines a hierarchy of grammar classes according to their descriptive power; the classes form a series of proper subsets. From most to least powerful, they are named Type 0 – Type 3:

- Type 0 grammars are known as *phase structured*, and allow arbitrary sequences of symbols on the LHS and RHS of productions (although left-hand sides cannot be empty). These grammars are extremely powerful—capable of generating all sets that can be generated—but at the cost of making automatic parser generation intractable in the general case [45], pp. 182-183.

- Type 1 grammars are known as *context sensitive*. They allow multiple symbols on the LHS, but require all but one of them to appear again on the RHS. The other (changed) symbol must be a nonterminal on the LHS, and may be replaced by any number of symbols on the right. In this way the production provides an unchanged context for the substitution. (Chomsky actually defines this class of grammars in a different but equivalent way; we use this definition as it provides an intuitive meaning for the "context" in "context sensitive".) This class of grammars is somewhat smaller than Type 0, and somewhat easier to parse. Nevertheless, they remain too difficult for practical use. In the words of Grune and Jacobs, "Type 0 and Type 1 grammars are well-known to be human unfriendly and will never see wide application." [39], pg. 70. They also note that "all known parsing algorithms for Type 0 and Type 1 grammars have exponential time dependency" (p.72).

- Type 2 grammars are the context free grammars that we employ in this work. They allow only one nonterminal on the LHS; in other words, the context provided by Type 1 grammars is missing. Type 2 grammars are much less powerful than Type 1 or 0 grammars, yet remain adequately powerful for practical application to most programming languages, and allow automatic parser generation. (This reduction in parsing power, however, is one reason subsequent semantic analysis is necessary.) Parsers for CFGs require polynomial time in the general case, but linear time parsers exist for some subclasses of CFGs. This distinction is emphasised in Section 3.2.2. Sections 3.2.3 and 3.2.4 attend to several subclasses of CFGs for which linear time parsers are possible.

- Type 3 grammars are known as *regular grammars*. They further restrict productions so that a RHS contains exactly one terminal, optionally followed by one nonterminal. This restriction makes them equivalent to regular expressions. They are not powerful enough to describe most programming languages, as they cannot describe nested constructs such as nested parentheses. They are well suited to the simpler task of lexical analysis. They can easily be parsed in linear time.

Hereafter, we return our attention to Type 2 grammars (CFGs).

3.2.2 General and restricted context free parsers

Automatic parser generation requires an algorithm that can convert a given grammar into a parser. For context free grammars, many such parsing algorithms have been devised. They vary in the speed with which the generated parser can process sentences and in their *power*. The power of a parsing algorithm is an indication of its generality: a more powerful parsing algorithm accepts a larger set of grammars than a weaker algorithm.

Several context free parsing algorithms are maximally powerful: they are capable of parsing all context free languages. In recognition of this power, they are known as *general* context free parsers. Examples include the Cocke, Younger, Kasami (CYK) parser [107], Unger's parser [103], and the more widely used Earley's parser [28]. The power of general parsers comes at the cost of requiring polynomial or even exponential time (relative to the number of tokens in the sentence) to parse some or all sentences of those grammars. Parsers based on Earley's approach have the fastest worst-case performance, requiring $O(n^2)$ time for unambiguous grammars and $O(n^3)$ for ambiguous grammars. No linear time general parsing algorithms are known.

The property of generality is extremely valuable. It allows a parser to accept any CFG, without imposing the need for the grammar to be transformed. For rigorous static analysis, this property is particularly important, as explained in Section 2.2. However, computer programs typically consist of very long sequences of tokens—possibly millions—for which parsers with polynomial time requirements are impractical. Consequently, programming language parsers have made infrequent use of general context free parsing algorithms. Instead, parser developers have used linear approaches, effectively trading generality for speed. We argue that, for a broad set of software engineering applications, this trade-off is not necessary with the use of GLR parsing.

GLR parsing is a special case of general context free parser. Chapter 4 explains the construction of a GLR parser. GLR, which stands for *General LR*[3], is a non-deterministic, directional, bottom-up parsing approach, also known as Tomita's parser [100], although the idea was described earlier by Lang [64] and subsequent improvements have been made by other

[3] More fully, General Left to right parser, producing a Rightmost derivation.

authors. Most importantly for our purposes, Nozohoor-Farshi [80] corrected a problem that caused the parser to loop infinitely on ε-transitions, although he described only a recognizer (rather than a parser). Rekers [91] describes a parser incorporating the fix.

GLR parsers have very poor—$O(n^3)$ or worse, depending on implementation—worst case performance [60]. Tomita, however, recognised the value of the approach for parsing natural languages, noting that the worst case behaviour is not produced by grammars and sentences of real natural languages. Tomita demonstrated near-linear performance for short sentences (some tens of tokens) of natural languages. Nevertheless, GLR parsing has been relatively little used for parsing of programming languages, in contrast to the linear methods. Tomita's emphasis on natural languages and the parser's inadequate worst-case behaviour have perhaps contributed to the parsing community passing over GLR parsing for programming language applications. In this research, we demonstrate the effectiveness and considerable advantages of using GLR parsing for the static analysis of programming languages; see Chapter 4.

By sacrificing power, parsers can be made faster. Parsers that are guaranteed to work in linear time can be obtained by restricting the set of grammars they accept. In order to parse in linear time, a parser must be deterministic: its behaviour must be tightly prescribed at every step because each input token must be processed in constant time. In contrast, more powerful parsers such as the general parsers discussed above are non-deterministic: they explore alternative parses until they can discover which are viable, and so their performance can degrade with input length.

Two fundamental distinctions can be made regarding the way parsing algorithms tackle the problem of parsing: whether they are directional or non-directional, and whether they construct parse trees from the top down or the bottom up:

- Directional parsers process tokens in sequence, either from left to right or right to left. Non-directional parsers process tokens in some more arbitrary order, and so require the entire sentence to be available in memory. Unsurprisingly, deterministic behaviour is more readily achievable with directional parsers.

- Top-down parsers identify the root of a parse tree first, and then proceed to construct branches, then leaves. Conversely, a bottom-up parser assembles leaves into branches, and branches into higher-level branches, until the root is ultimately recognised.

The next two sections look at the dominant parsing approaches today; both are directional, in order to attain linear time performance. They differ in the order in which they recognise parse trees: top-down and bottom-up.

3.2.3 LL parsers

This section briefly introduces directional, top-down parsing. Deterministic parsers in this class are very widely used because they are simple and intuitive. Unfortunately, this class is relatively weak, and consequently prone to requiring extensive grammar transformation.

Any directional parser that reads the input from left to right and assembles the parse tree from the top down is known as an LL parser. The first L stands for Left to right (hence directional) and the second L stands for Leftmost derivation. A leftmost derivation, when produced by a left to right parser, is simply a top-down construction of the parse tree. The terminology makes more sense when a grammar is viewed as a production mechanism for generating sentences, rather than as a recognition mechanism for parsing. In effect, a top-down parser reverses the series of transformations that would occur if the sentence were derived from the start symbol by repeatedly substituting the leftmost nonterminal in the sentence with an appropriate RHS.

An equally powerful (and consequently rarely used) class of grammar is RR: a Right to left parser producing a Rightmost derivation. This is a directional top-down parser that processes tokens in the reverse order from LL. It is worth noting that although LL and RR are equally powerful, they are not identical. A grammar that is LL is not necessarily RR and vice versa. The grammar classes' power is described as equivalent.

A non-deterministic LL parser is general; it can parse any context free grammar, but not in linear time. This kind of parser is commonly implemented as a *backtracking* recursive descent parser. Such a parser explores the space of possible parse trees (given the input seen so far) using recursive function calls, one function per nonterminal. This is simply a depth-first

search strategy through the space of possible parse trees for the input seen so far. Backtracking occurs when a local search fails to match the actual input tokens and an alternative must be tried from some higher point in the parse tree.

LL parsers can be made deterministic if they can successfully predict the input whenever the parser is faced with choosing between alternative RHSs; this eliminates the need to backtrack. If, for a given number *k* of tokens, an LL parser can always predict the correct RHS by examining only the next *k* lookahead tokens, then the parser is said to be LL(*k*), and has linear time performance.

In practice *k* is usually set to one, making LL(1) the dominant top-down parsing approach. Although higher values of *k* yield greater parsing power, parser developers often choose to modify grammars rather than calculate deeper lookaheads because the number of possible lookaheads grows exponentially with *k*. Nevertheless, some practical LL(*k*) parser generators exist;ANTLR [87], for example.

The work of Parr [86] should be noted here: any parsing algorithm that produces homogeneous table entries (or states) containing the same depth of lookahead is necessarily limited to small values of *k* because of the combinatorial explosion of lookaheads. Parr shows that practical LL (and other) parsers with larger values of *k* are attainable by using different values of *k* for different parts of the parser. For realistic grammars, long lookahead is required in very few parts of the grammar, so parsers with heterogeneous table entries can mitigate the effects of lookahead explosion by using high values of *k* only where it is essential. Parr's work elevates LL(*k*) parsing for $k > 1$ to a practical parsing approach. (Parr obtains the same conclusion for LR parsing classes, except for canonical LR(k), which relies on full lookahead for state generation. We revisit this issue below and obtain a better result in Chapter 4.)

Nevertheless, the power of LL(*k*) remains fundamentally limited by its need to make predictions based on lookahead. Users of LL(*k*) parsers typically have to make substantial modifications to grammars in order to avoid grammar constructs that defeat the algorithm. A frequent grammar transformation is known as *left factoring*, in which ambiguities caused by common prefixes in two or more RHSes are removed by forcing the prefix into a common production. Likewise, *left recursive* productions always confound deterministic LL algorithms and must be eliminated from the grammar. A left recursive production is one in

which the first nonterminal on a RHS is, after zero or more productions are applied, the same as the LHS; see *classModifiers*$_2$, in Figure 12 for an example.

While directional, top-down parsing is perennially popular because of its accessibility, it is not well suited to the task of parsing without having to modify the grammar, and hence not well suited to our purposes. The next alternative, directional bottom-up parsing, presents much better characteristics.

3.2.4 LR parsers

A directional parser that reads the input string from left to right and constructs the parse tree from the bottom up is called LR. As before, the L, indicates direction: Left to right. The R denotes a Rightmost derivation, implying bottom-up construction. An RL parser is the right to left equivalent of LR (and, like RR, is rarely used). Like most parsing literature, we ignore RL parsers.

Directional, bottom-up parsers are more complex to construct and harder for humans to understand, but they offer a significant advantage over top-down: prediction is unnecessary. An LL parser makes decisions (predictions) as soon as possible, while an LR parser defers decisions for a long as possible. In LR, recognition of a nonterminal is deferred until all its RHS symbols have been seen. This means an LR parser has a richer context to work with than an LL parser. At the same point in the input, an LR parser will have constructed less of the tree because it has deferred decisions until it sees more.

As with LL parsers, LR parsers are general if they are non-deterministic. This is exactly the approach taken by Tomita's parser, hence the name General LR (GLR). As previously noted, however, GLR is not linear in the worst case, and the programming language parsing community has concentrated on deterministic LR approaches, despiteTomita's [97] finding that in practice , near-linear performance is exhibited by GLR parsers for real sentences of natural languages.

The capabilities of LR parsers are derived directly from the characteristics of Push-Down Automata; this class of parser is the result of using PDAs to parse. The various subclasses of LR such as SLR and LALR arise from different strategies for configuring the PDA. Remarkably, the most powerful approach, LR(*k*), was also the first [61].

A PDA is a finite state machine augmented by a stack that records states traversed. The use of a stack improves the power of the state automaton, allowing it to track nested constructs, such as parentheses. Intuitively, a PDA can be viewed as using states to track the progression of a parse through the RHS of a production, while using the stack to track nesting of productions. In Section 3.3 we provide several examples of PDA construction and execution. For now, this intuitive explanation is sufficient to introduce the two basic actions of a PDA:

- *Shift* actions change the automaton's state, by consuming a token and taking a transition to another state. In effect, these actions simply record the presence of the token and move on.

- *Reduce* actions occur when a complete production has been recognised. They pop the PDA to an earlier state—one expecting to see the recognised nonterminal. A reduce action is always followed by a *goto* action that consumes the newly reduced non-terminal and moves the machine to a new state. A goto action is just like a shift, except it consumes a nonterminal instead of a terminal. The result of a reduction is to take the PDA back to the state it was in before seeing the recognised nonterminal, and then the goto consumes that nonterminal and moves to a new state.

A deterministic PDA must be able to choose its next action—a shift or reduce—by examining only the current state and the current lookahead, i.e. the next *k* tokens in the input stream. It is this property of allowing only one action that makes the PDA deterministic. Shift actions are straightforward: a shift always matches the next single token, which uniquely identifies the next state. Consequently, PDA states never contain conflicting shift actions. A state may, however, contain any number of reductions and only one may be chosen if the automaton is to remain deterministic. Each reduction can potentially lead to a different state, and hence may be followed by a different sequence of tokens. If these lookahead sequences are disjoint, the state can always choose the right reduction. If not, the state contains a *reduce-reduce conflict*. Similarly, if a shift action and the sequence of tokens that follow it coincides with the lookahead of a reduce action, a *shift-reduce conflict* occurs. A state containing either type of conflict is described as *inadequate*. If any state is inadequate, the parsing algorithm fails for the grammar as a whole and a deterministic parser cannot be generated using this algorithm.

Inadequate states can sometimes be avoided by using deeper lookahead, or by using more states in the automaton in such a way that fewer lookaheads apply in individual states. The latter strategy is used in canonical LR(*k*), which minimises the potential for ambiguities by differentiating the parser's context into as many states as possible. This is the source of the approach's power, but also its downfall for practical application: it increases the number of states by orders of magnitude, because the possible future inputs to the parser must be anticipated by the states.

SLR(*k*) and LALR(*k*) parsers do not proliferate states like LR(*k*). In fact, they use exactly the same set of states as LR(0), differing only in their use of lookahead to choose parser actions. SLR stands for *Simple LR*, so named because it uses a simplistic approach to calculating lookahead sets: it derives from the grammar a covering approximation of the actual lookahead sets, giving a parser that is much more powerful than LR(0). Because the lookahead is a superset of the actual possibilities, SLR(k) may still fail to resolve some ambiguities that cannot occur in reality. LALR(*k*) improves the lookahead calculation until it is optimal given the restricted number of states available, hence its name: *Look Ahead LR*.

As with LL parsing, LR lookahead depth is usually restricted to 1 to avoid combinatorial explosions. As we noted earlier, Parr [86] adopts heterogeneous depths of k to make larger values of k practical. Even so, he uses a covering approximation of the real lookahead (in a manner akin to SLR lookahead calculation) to keep lookahead calculation time linear. His approach does not extend to canonical LR(k) parsing. He states "The LR(*k*) parsing method has little to gain from the linear approximation analysis as the number of parser states is exponential and full *k*-lookahead info must be moved along during state construction in case it is needed." In Chapter 4 we present a practical approach for calculating heterogeneous depths of *k* using exact (not approximate) lookahead that works for LR parsers up to and including full LR(*k*).

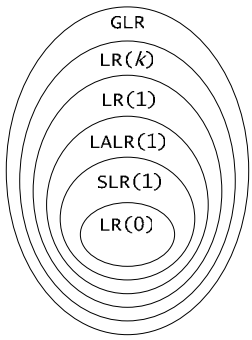

Figure 15: Hierarchy of LR grammar classes

Figure 15 depicts the relative power of those LR grammar classes that are proper subsets. The situation is less clear cut when values of *k* greater than 1 are used for SLR(*k*) and

LALR(*k*). The outermost class, GLR, is equivalent to the class of context free grammars.

As we explained in Section 2.2, LALR(1) is today the dominant bottom-up parsing approach; higher values of *k* or more powerful algorithms are relatively rare in practice. Many LALR(1) parser generators are available, including the widely used yacc and bison. Newer versions of bison also include a GLR option, but a recent paper indicates the implementation is seriously flawed: "… the present Bison 'GLR' implementation merely splits stacks when conflicts are encountered, so it displays exponential growth in memory requirements which makes it impractical." [56].

For tasks such as compiler construction, the limits of LL(*k*) or LALR(1) are generally tolerated (as indicated by the number of compilers using deterministic parsing technology), although they can be a substantial burden on developers, and complex languages such as C++ push the limits of what is possible with these technologies.

For the purposes of static analysis of software, however, grammar modifications are even less desirable. In the interests of rigour and communicability, identification and measurement of syntactic features should be defined in terms of a language's standard grammar. For example, a metric that counts the number of declarations in a program should define *declaration* in terms of the language standard, since this is what the language's community understands. If a parse tree reflects a non-standard grammar, its usefulness for deriving syntactic metrics is compromised. A metric such as the number of *expressions* per *statement*, for example, is sensitive to perturbations of expression and statement syntax. Similarly, semantic constructs are predicated upon, and described in terms of, syntactic constructs, so conformance to a standard grammar offers significant advantages for semantic modelling and metrics of semantic features.

Much metrics literature treats lightly the problem of rigorously defining features of software to be measured, assuming that casual concepts of features such as classes, inheritance, methods, declarations, expressions, etc are adequate. We argue, however, that these entities should be precisely defined, that doing so is often less straightforward than might be expected, and that some metrics are very sensitive to variations in definitions. For example, a metric that counts the methods of a class should define precisely what is meant by the term *method*. Does it include constructors, compiler-generated methods, overloaded operators

and instantiated template methods? How does inheritance and method overloading affect the count? The use of standard grammars (and semantics) provides a definitive basis for describing which language features are included.

For completeness, we note that some grammar transformations can be automated. For example, [92] provides techniques for modifying grammars to conform to various LR subclasses. Techniques such as these guarantee accuracy, but detract from the original grammar writer's communication efforts and interfere with software models and metrics in the same way as manual interventions.

These considerations provide motivation for questioning the dominance of LALR(1). We (and other authors) find that stronger parsing classes are in fact practical. Consequently we suggest that static analysis tool developers forego the usual approach of transforming grammars to accommodate the limits of LL(*k*) or LALR(1), and instead use a parser generator that applies algorithms sufficient for given grammars. For real programming languages, the full power of LALR(*k*) and LR(*k*) can be obtained while avoiding the proliferation of states and lookaheads that has historically made these parser classes unusable. If even these classes prove inadequate, GLR parsers can be used. Chapter 4 describes a tool that delivers the full spectrum of LR parser powers.

3.3 LR parser classes—an escalating example

The landscape of LR parser classes is complex (and some LR classes have been omitted here, NQLALR(*k*), for example [24]), and the situation is not helped by the convoluted evolution of ideas in the field, or the often theoretically strong but less practical treatment in the literature. In this section we present a series of examples showing the construction of LR parsers of increasing power: LL(1), LR(0), SLR(1), LALR(1), LR(1) and GLR. This escalating approach demonstrates how each parsing class improves upon the previous one. More significantly, it is consistent with our approach for building hybrid LR parsers by applying progressively more powerful algorithms to a subset of the state machine, resulting, in this case, in an LR(1) parser being built without proliferating states. This progression shows also how GLR naturally extends the linear LR parsers by introducing nondeterminism in a con-

trolled way, thus enabling it to deal with programming languages too complex for lesser approaches.

In these examples, we ignore parse tree construction, in order to concentrate on the parsing process itself.

```
statement ::= declaration
            | assignment

declaration ::= type variable ;

type ::= int
       | bool

variable ::= id

assignment ::= variable = value ;

value ::= id
```

```
int i;
```

```
bool b;
```

```
i = j;
```

Figure 16: A grammar for a trivial language, and some sentences it generates

3.3.1 LL(1) parsing

Figure 16 shows a grammar that generates only three possible sentences, which are shown in the same figure. Parse trees for these sentences are in Figure 17.

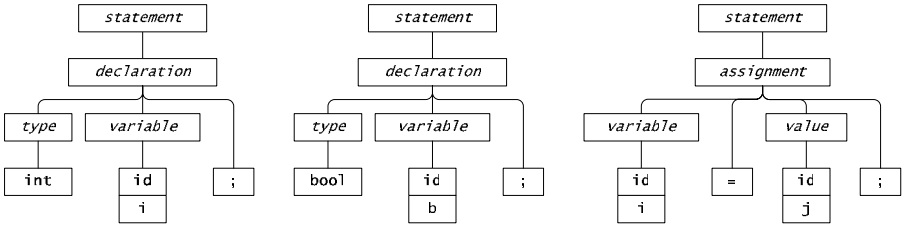

Figure 17: Example sentences and parse trees

In order to parse a sentence, we must read the raw tokens of the sentence and derive a valid parse tree, if one exists. For the language of Figure 16, the number of possible sentences is finite (exactly 3). Consequently, constructing a parser for this language is trivial; the parser need only enumerate the possibilities and compare the input with them.

A recursive grammar, however, can generate an infinite number of sentences. In Figure 18, the original grammar is extended with a recursive production, and an example sentence shown. (Changes from the previous grammar are highlighted.) Figure 19 gives an example parse tree.

```
statements ::= statement
             | statement statements

statement ::= declaration
            | assignment

declaration ::= type variable ;

type ::= int
       | bool

variable ::= id

assignment ::= variable = value ;

value ::= id
```

```
int i;
bool b;
i = j;
```

Figure 18: A recursive grammar and example sentence

Although the sentences of this language are innumerable, parsing is still simple: we could add a loop to our earlier parser in order to recognise the series of statements. But if we were to continue to add productions—including recursive productions—to the

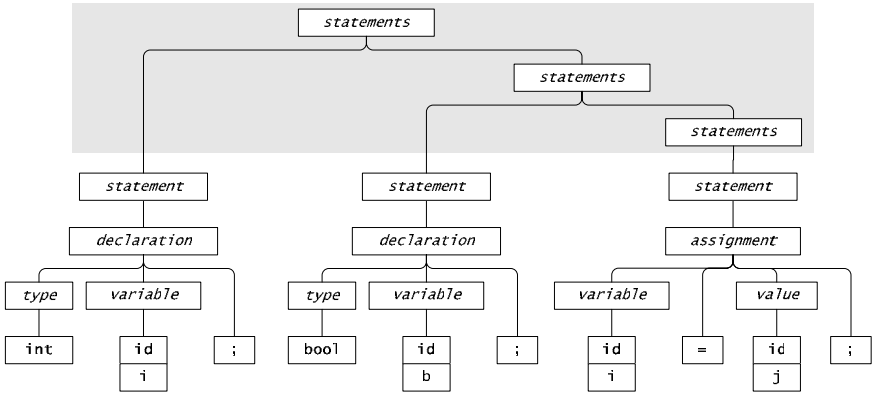

Figure 19: Example parse tree for recursive grammar

grammar, a generated parser would require a systematic way of tracking the state of the parse in order to know what choices were possible at any point in the input stream. An intuitive and widely used solution to this problem is a *recursive descent* parser. Figure 20 provides an example recursive descent parser for this grammar. For simplicity, this program recognises the language but does not build a parse tree. An appropriate scanner is assumed.

A recursive descent parser typically has one method for each nonterminal of the grammar. The runtime stack (of method invocations) is used implicitly to keep track of the nesting of productions as they are recognised. This leaves individual methods with two simple responsibilities:

- To keep track of the position of the parse as it steps along the sequence of symbols in the right-hand-side of a production. For example the `parseAssignment()` method sequentially matches the symbols `id = id ;` on the right-

```
class RecursiveDescentParser {

    Scanner scanner = new Scanner(System.in);
    Token lookahead = scanner.next();

    public static void main(String args[]) {
        new RecursiveDescentParser().parseStatements();
    }

    private void parseStatements() {
        parseStatement();
        if (lookahead != Tokens.EOF)
            parseStatements();
    }

    private void parseStatement() {
        if (lookahead == Token.INT || lookahead == Token.BOOL)
            parseDeclaration()
        else if (lookahead == Token.ID)
            parseAssignment()
        else
            error(Token.INT | Token.BOOL | Token.ID);
    }

    private void parseDeclaration() {
        parseType();
        parseVariable();
        advance(Token.SEMICOLON);
    }

    private void parseType() {
        advance(Token.INT | Token.BOOL);
    }

    private void parseVariable() {
        advance(Token.ID);
    }

    private void parseAssignment() {
        parseVariable();
        advance(Token.EQUALS);
        parseValue();
        advance(Token.SEMICOLON);
    }

    private void parseValue() {
        advance(Token.ID);
    }

    private void advance(int tokenMask) {
        if (lookahead | tokenMask != 0)
            lookahead = scanner.next();
        else
            error(tokenMask);
    }

    private void error(int tokenMask) {
        System.out.println("Syntax error on " + lookahead );
        System.out.println("  Expected: " + Token.getName(tokenMask));
        System.exit(-1);
    }
}
```

Figure 20: A recursive descent parser

hand-side of an assignment production.

- To choose which productions of a nonterminal match the input. For example the `parseStatement()` method uses the current lookahead token to choose between a declaration and an assignment.

The example parser of Figure 20 examines the next token in the input stream in order to choose between alternative productions. In doing so, it attempts to predict which tokens it will see in the near future. Consequently, this type of recursive descent parser is known as a *predictive parser*. For the given grammar, it is possible to distinguish between alternative productions using one token of lookahead, because the set of tokens that may begin a declaration (`int, bool`) is disjoint from the set of tokens that may begin an assignment (`id`).

Predictive parsers that use *k* tokens of lookahead can handle the class of grammars known as LL(*k*): a *left-to-right* parse producing a *leftmost derivation* using *k* tokens of lookahead. Their behaviour is deterministic because they always choose one option, as opposed to exploring all options.

The need to accurately predict right-hand sides is the limiting factor on the power of this parsing approach. A recursive descent parser assembles a parse tree from the top down. That is, it recognises the root node and descending branches before recognising a leaf node. As it descends, it must choose between alternative productions (such as *`declaration`* and *`assignment`*) in order to construct the correct branch—and to know which method to call). At the time it makes these decisions, it has not yet seen the tokens that will eventually make up the production being chosen.

3.3.2 LL(1) parsing using automata

As noted above, each method in a predictive parser must keep track of its position in the sequence of symbols on the right hand side of the production it is attempting to recognise, and when faced with alternative productions must use lookahead tokens to choose which is applicable. In the program of Figure 20, this is done using raw programming language constructs: sequential statements track progress through a production and if-statements choose between productions. The same behaviour can be achieved using a set of simple Deterministic Finite Automata (DFAs) that track the state of the parse of a production.

Figure 21 provides a set of DFAs based on the grammar of Figure 16. (For simplicity, the original grammar is used here instead of the recursive grammar of Figure 18.) The guard conditions on the transitions from state 1 indicate the need for lookahead to choose between the transitions. Each DFA can be mechanically produced from the grammar's productions like this:

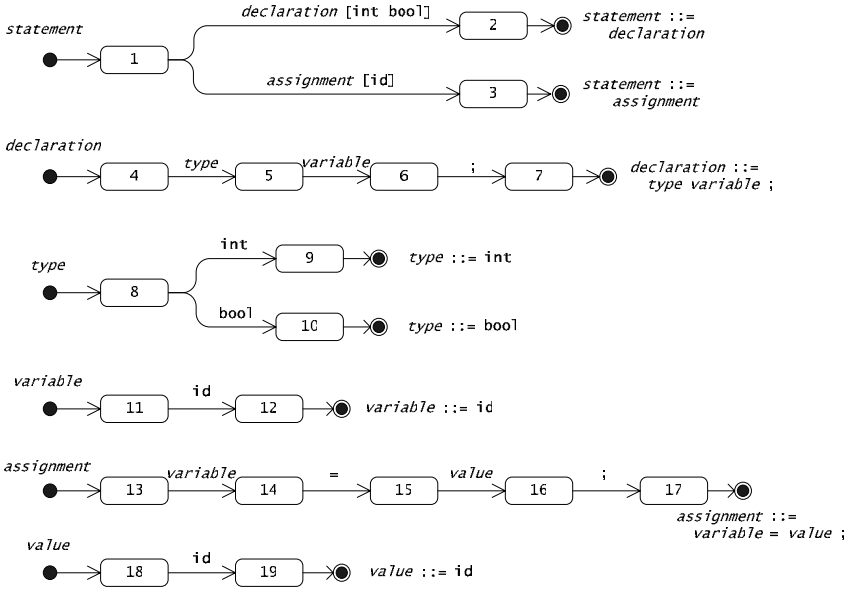

Figure 21: Recursive descent deterministic finite automata

```
for each nonterminal, N
    make a start state N0

for each production, N ::= s1 s2 s3…
    end ← N0
    for each si
        make a new state, ei
        add a transition from end to ei
        end ← ei
    add a reduction of N ::= s1 s2 s3… to end

if any state has two or more transitions on the same symbol
    merge the destination states

add guard conditions to choose between transitions on nonterminals
```

The guard condition on a transition is the set of tokens that can begin that nonterminal. (This is known as that nonterminal's *first set.*)

Given this set of DFAs, it is straightforward to implement a predictive parser similar to that of Figure 20 in which each method explicitly implements one of the above automata. Each

transition on a nonterminal involves invoking another parser method (which runs another automaton), and each transition on a terminal requires checking that the terminal exists in the input. Choices between transitions are made by comparing the lookahead token to the tokens in the first sets of the nonterminals.

It is not strictly necessary to implement the automata as methods. The methods—and the runtime stack that implicitly tracks their nesting—could be replaced with an explicit stack that records which automata are currently active, and which state each automaton currently occupies. In this way, a nesting set of DFAs can parse LL(1) languages.

3.3.3 LR(0) parsing

The weak point of LL parsing is its need to make predictions based on limited context. If a grammar allows two productions that begin with the same token(s) to occur in the same context, the approach fails. For example, Figure 22 extends the base grammar of Figure 16 with the production *type* `::= id`. This allows a *declaration* to begin with an `id`, so that an LL parser can no longer tell from one token of lookahead whether it is faced with a *declaration* or an *assignment*.

```
statement ::= declaration
            | assignment

declaration ::= type variable ;

type ::= int
       | bool
       | id

variable ::= id

assignment ::= variable = value ;

value ::= id
```

```
i = j;
```

```
String s;
```

Figure 22: Grammar modified to defeat LL(1) parser

Increasing the lookahead to two would solve this problem for the given example. This is not a general solution, however. The example of Figure 23 allows an arbitrary number of identical tokens to begin *variableValue* and *literalValue*. Either production allows nested parentheses to an arbitrary depth. No fixed number of lookahead tokens will be adequate for all sentences of this grammar. This problem is a compelling reason to avoid making predictions by deferring decisions until more input has been seen.

An intuitively appealing (but ultimately inadequate) approach is to combine the separate DFAs of an LL parser (such as that of Figure 21) into a single DFA that tracks parse state until the complete RHS of a production has been seen. For the grammars introduced prior to Figure 23, this approach would work. However, Context-Free Grammars have greater expressive power than DFAs (which are equivalent to regular expressions). For the grammar of Figure 23, it is not possible to construct a DFA that can recognise this language, because a DFA has no ability to track the nesting levels of parentheses.

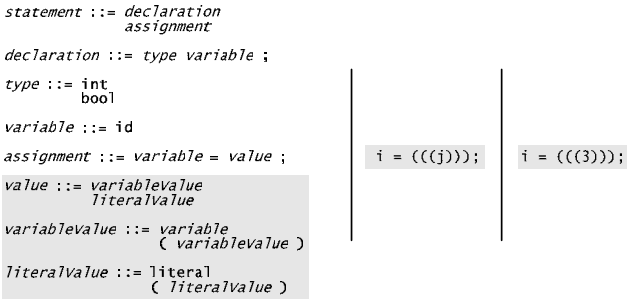

Figure 23: Grammar modified to defeat LL(k) parser

Fortunately, a simple improvement to the power of a DFA adds this ability. A *Push-Down Automaton* (PDA) is a DFA that uses a stack to track its progress through states, and can pop the stack to jump back to earlier states. In Section 3.3.2, we noted that a set of DFAs plus a stack is adequate for LL(1) parsing. The stack provides the ability to track nested productions in order to ensure that symbols—such as parentheses—match. If we use a single PDA in place of the nesting LL(1) automata, we will retain the ability to track nested productions (via the PDA stack), and add the ability to defer recognising productions until they are complete.

We now explain how a PDA can be derived from a set of LL DFAs, which in turn were derived from the original productions.

Figure 24 depicts a nondeterministic automaton derived from the LL auto-

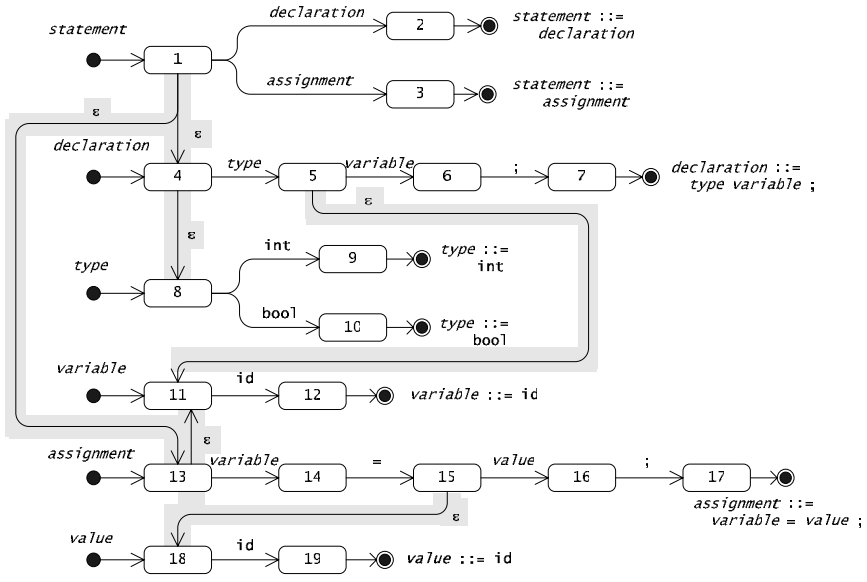

Figure 24: Combining LL automata into a nondeterministic PDA

mata of Figure 21. A parser at the start of the token stream is in State 1, expecting to see an assignment or a declaration. In an LL parser, we must predict which production is to be expanded, by using lookahead. If, on the other hand, we choose to defer this decision, then we must simultaneously occupy the start states for *declaration* (state 4) and *assignment* (state 13). We must also continue to occupy state 1, because we have not yet found out which transition to take in order to depart. We can introduce this behaviour by adding *epsilon transitions* from state 1 to states 4 and 13. Epsilon (ε) represents empty input; that is, a transition on epsilon can be taken without consuming input.

State 4 begins a declaration. It expects to see a type. We can defer recognising the type until we have actually seen it, by adding an ε–transition to the start state of the type automaton, state 8. Likewise, we add an ε–transition from every state with an outgoing transition on a nonterminal, to the start state of the automaton for that nonterminal.

Epsilon transitions introduce nondeterminism into the state machine; it can occupy more than one state at a time. We can eliminate nondeterminism by merging states, like this:

- For each state, find the set of states reachable on ε.
- Copy all outgoing transitions from those states into the original state.
- If the resulting state is still nondeterministic (because it now has multiple transitions on the same input), merge the set of states reachable on that input.

Figure 25 shows the resulting PDA, after ε–transitions are removed. Note that as the automata have

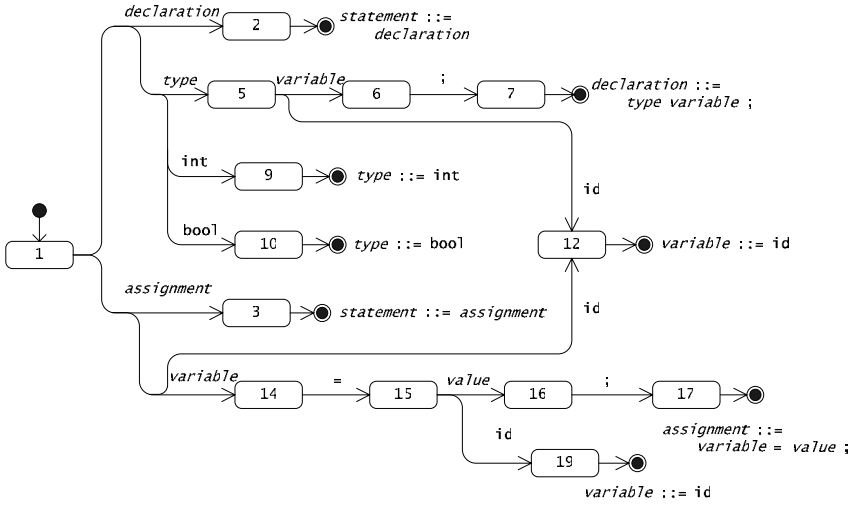

Figure 25: LR(0) Pushdown Automaton

been combined into a single automaton, the start states of all nonterminals except state 1 have been merged into other states. State 1 is the start state of the start nonterminal *statement*, and so has become the start state of the PDA. The start states of all other productions are now unreachable, and so have been eliminated from the machine.

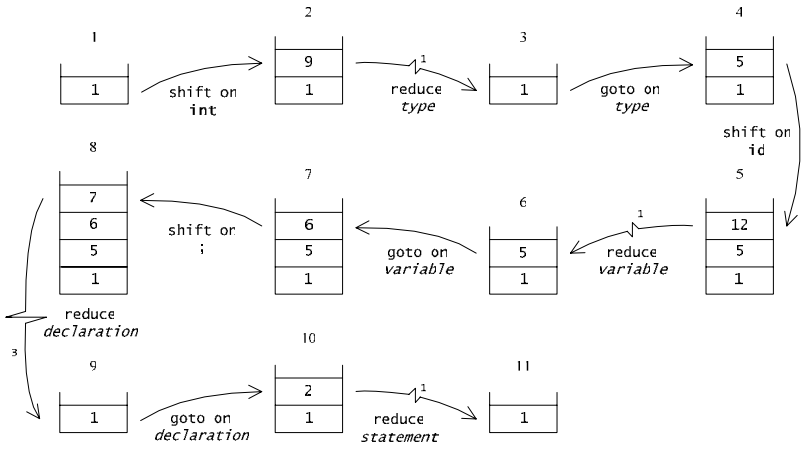

Figure 26: Execution of PDA

Figure 26 shows an example of the operation of the PDA as it parses the sentence:

```
int i;
```

The stacks (numbered 1 – 11) in the figure show the sequential configurations of the PDA. The topmost state of each stack is the current state of the PDA. This figure relies on the PDA terminology:

- A *shift* occurs when the PDA takes a transition on a terminal. Shifts always push one more state onto the stack. It becomes the current state.

- A *goto* occurs when the PDA takes a transition on a nonterminal. Like shifts, gotos push the new current state onto the stack.

- A *reduction* pops a number of states off the stack. The newly exposed topmost state becomes the current state of the automaton. Reductions occur when a state recognises a production; the number of states popped equals the number of symbols on the RHS of the production. In our notation, reductions are denoted by a zigzag arrow showing the number of states popped. A reduction is always followed by a goto, because the reduction always yields a nonterminal (from the LHS of the recognised production).

- Shifts, gotos and reductions are known as parser *actions*.

The behaviour of the parser shown in Figure 26 is:

1. The stack is initialised by pushing the start state: State 1.

2. The first token is `int`, so the PDA shifts to State 9. State 9 is pushed onto the top of the stack. State 1 remains in the stack, so that the PDA may return to it later, when it has recognised a complete production.

3. State 9 recognises that the token `int` corresponds to the RHS of a type production (*type* ::= int). This causes the PDA to reduce the production: it pops the states traversed in recognising the production and returns to the state that it occupied before those symbols were encountered. In this case the RHS contains only one symbol, the token int, so the stack is popped once and the PDA jumps back to the last uncovered state, State 1.

4. Now that it has recognised a nonterminal symbol (*type*), the PDA takes a goto transition on that nonterminal. This leads to state 5, which is pushed.

5. The next input is the token id. The parser shifts to state 12.

6. State 12 reduces *variable* ::= id, which pops the PDA back to state 5.

7. A goto on *variable* leads to state 6.

8. A shift on ; leads to state 7.

9. State 7 recognises a declaration (*declaration* ::= *type* id ;), so pops the three symbols of the RHS off the stack and returns to the uncovered state, state 1.

10. A goto on declaration leads to state 2.

11. Finally, state 2 recognises the entire statement and pops back to state 1.

At this point, the PDA has returned to the initial state. No other states appear on the stack and no unprocessed input remains. The start symbol of the grammar, statement, has just been recognised, so the parse is successful.

Push-down automata that use no lookahead, such as the example of Figure 25, can parse the class of grammars known as *LR(0)*. This stands for a *left-to-right* parse producing a *right-most derivation* using *zero* tokens of lookahead. In the context of a left-to-right parse, the term *rightmost derivation* describes a bottom-up parse; that is, leaves are recognised before branches, and the root is recognised last.

Earlier in this section, we noted that the grammar of Figure 23 defeats an LL(1) parser. We now construct an LR(0) parser for that grammar to show that LR(0) is equal to the task. Figure 27 represents an intermediate stage in the construction of the PDA, as separate LL(1) automata are combined into one nondeterministic PDA. Differences from the automaton of Figure 25 are highlighted. Epsilon transitions have been added from every state expecting a non-terminal to the start state of that non-terminal. For example, state 23 (which expects a *variableValue*) has an ε–transition to state 21 (which

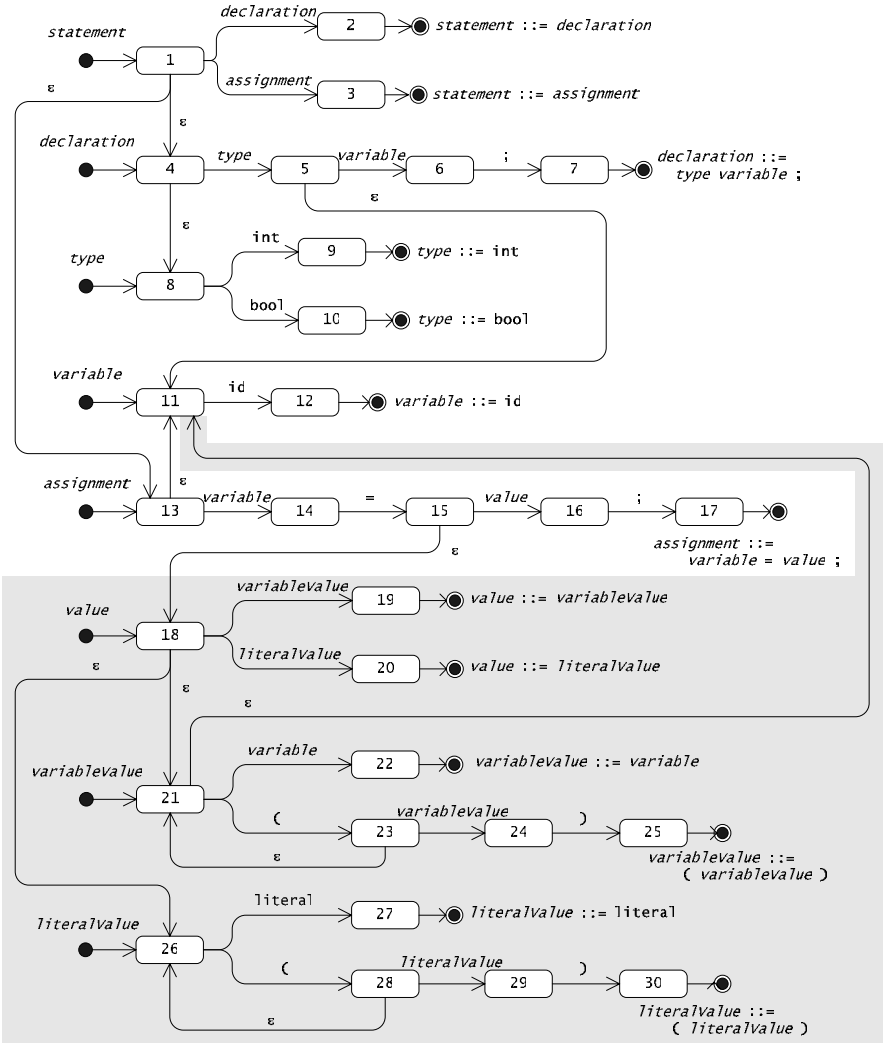

Figure 27: Partially constructed LR(0) parser for grammar with nested parentheses

begins a *variableValue*).

Removal of ε–transitions leads to the PDA shown in Figure 28. Note that some states (23 and 28) now have transitions to themselves. This is the result of removing ε–transitions within (left) recursive productions. For example, state 23 is reached while trying to recognise a *variableValue*, and it also expects an inner *variableValue*, which in the previous figure caused it to have an ε–transition to state 21. When the ε–transition is removed, the transitions out of state 21 are copied to state 23, giving it a self-transition (and a transition to state 22).

Although ε–transit-ions have been removed, the PDA of Figure 28 is still nondeterministic. State 15 has two transitions on an open parenthesis, reaching states 23 and 28. The final step in producing a deterministic PDA is to merge these two states into a new state. This appears

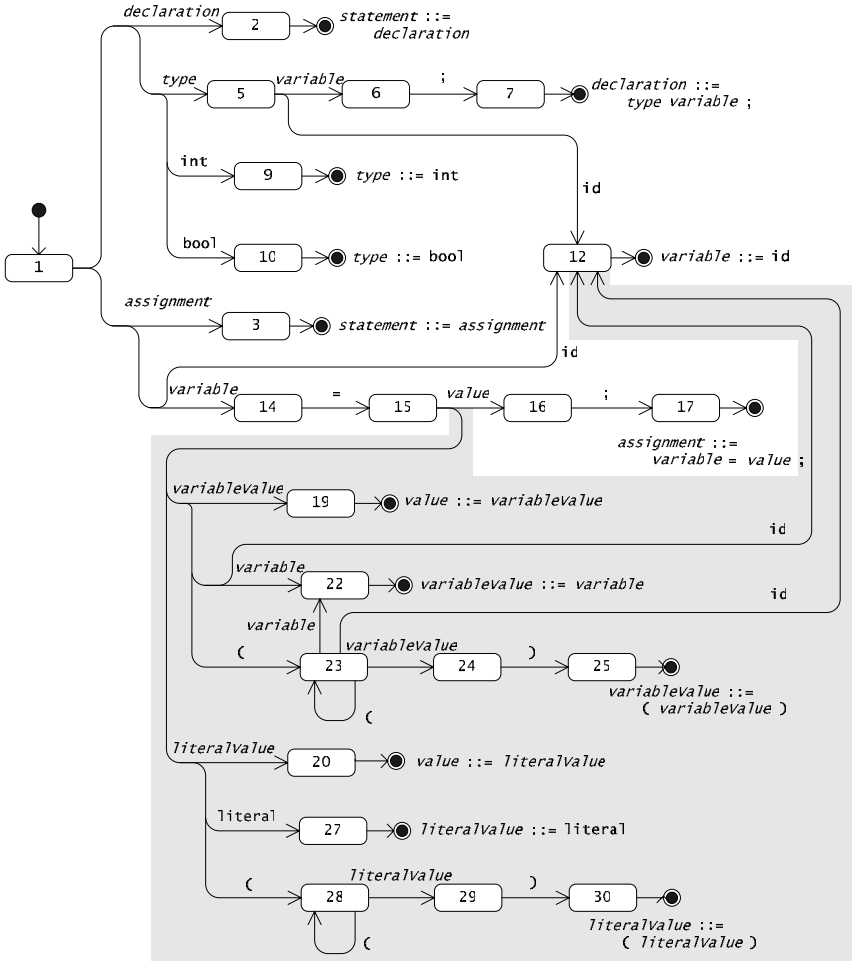

Figure 28: LR(0) PDA with ε-transitions removed

as state 31 in Figure 29.

The resulting PDA can differentiate between the sentences shown in Figure 23. For example, the sentence:

```
i = (((3)));
```

produces the sequence of actions shown in Figure 30, which captures the stack configurations up to the point that the nested parentheses have been recognised. (The example stops at the point where the parser is in state 20. It will subsequently reduce to state 15, goto 16, shift to 17, and reduce to 1.)

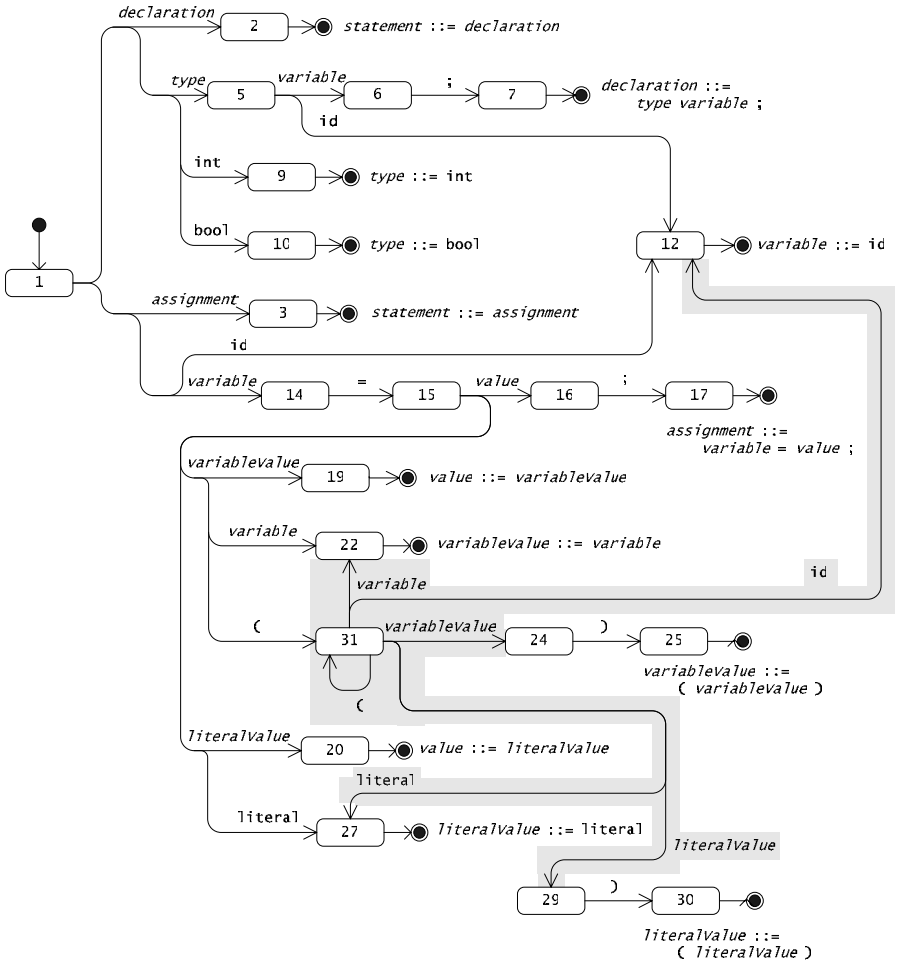

Figure 29: Deterministic LR(0) PDA

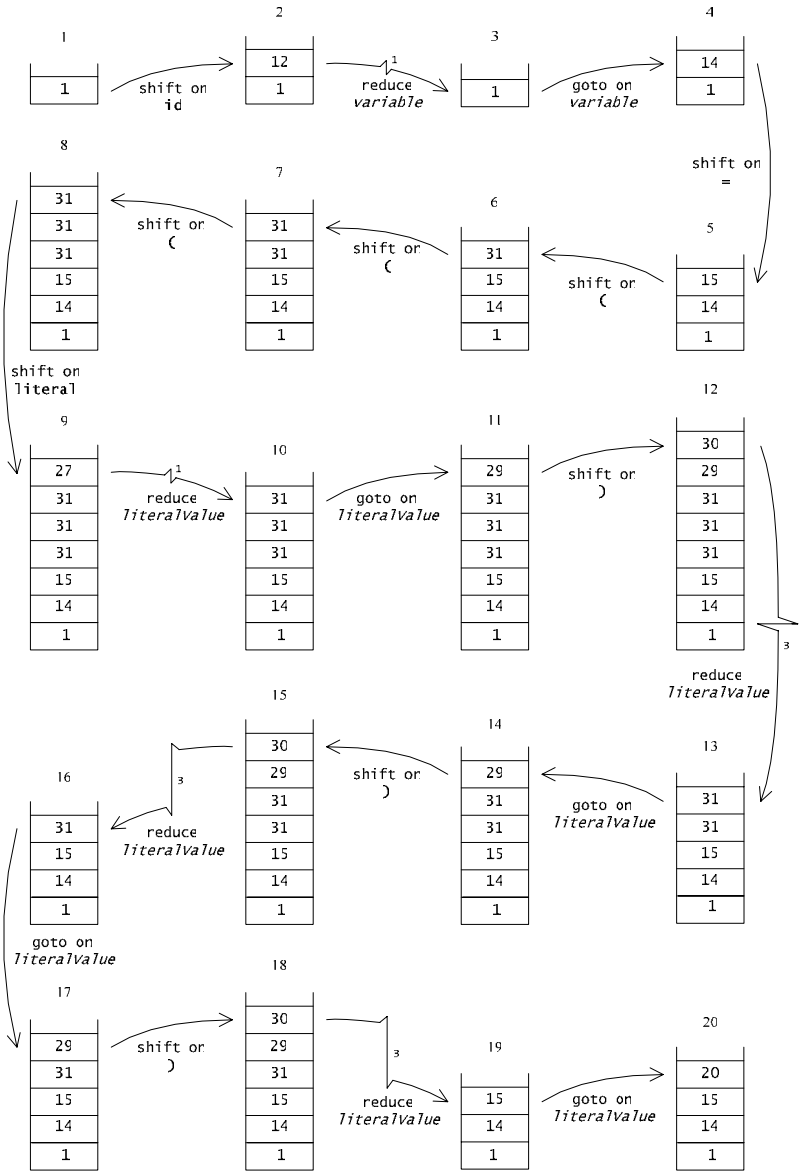

Figure 30: Execution of PDA as it parses nested parentheses

3.3.4 SLR(1) parsing

At this point, we have seen how to construct an LR(0) parser, which can handle some grammars that defeat an LL(1) parser. Many common grammar constructs, however, also defeat an LR(0) parser. Figure 31 presents a modified version of our example grammar,

```
statement ::= declaration
            | assignment

declaration ::= type variable ;

type ::= int
       | bool

variable ::= id

assignment ::= variable = value ;

value ::= variable
        | methodCall

methodCall ::= methodName ( )

methodName ::= id
```

i = foo;

i = foo();

Figure 31: Grammar modified to require an SLR(1) parser

with changes from the previous grammar highlighted.

This new grammar eliminates nested parentheses, but introduces a new problem. Figure 32 shows the construction of an LR(0) PDA, at the point ε-transitions have just been removed. State 15 is still nondeterministic because it has two transitions on **id**, to states 12 and 27. In the previous example, we resolved a similar problem by merging the two destination states to form a new state. In this case, however, merging of states 12 and 27 produces a new state, as shown in Figure 33, which is *inadequate*. State 28 is faced with a dilemma: it cannot tell whether to reduce the **id** as a *variable* or as a *methodName*. This is known as a *reduce-reduce conflict*.

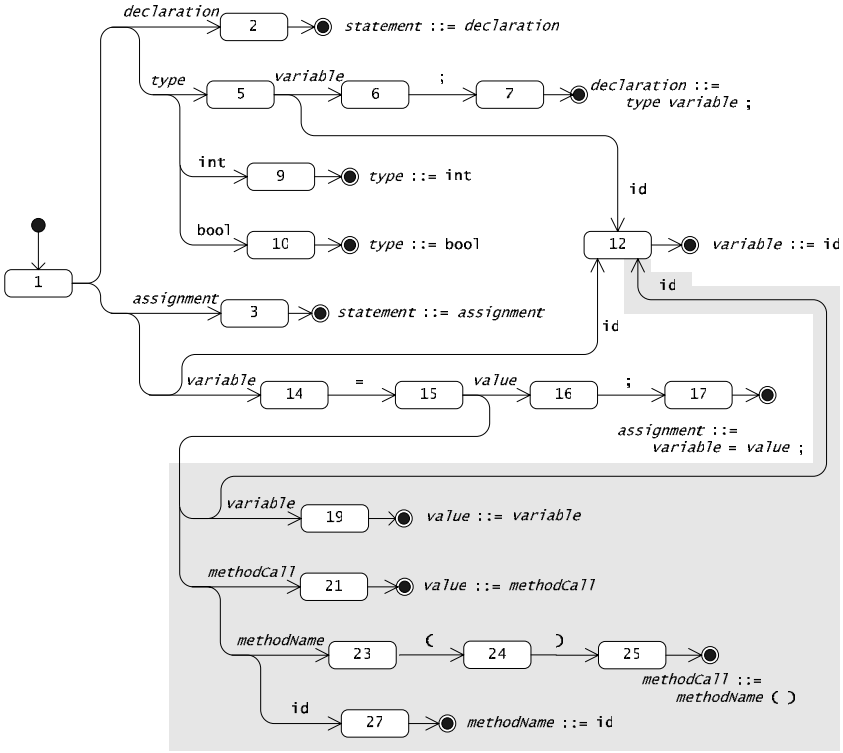

Figure 32: Nondeterministic LR(0) PDA for grammar with methodCall

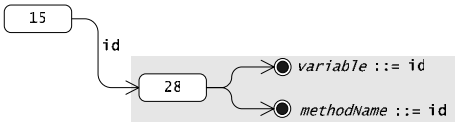

Figure 33: LR(0) state containing a reduce-reduce conflict

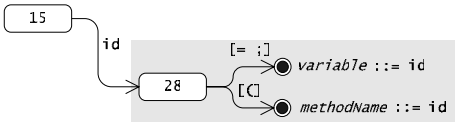

Figure 34: LR(1) state that eliminates the conflict

The solution is straightforward: lookahead is used to choose between the alternatives. Figure 34 shows the same state with lookaheads represented as guard conditions on the state machine. If the next token in the input is **=** or **;** then the **id** should be reduced as a *variable*. If the lookahead is an open parenthesis then the **id** is a *methodName*. Anything else indicates that the sentence contains a syntax error.

A simple way of calculating allowable lookaheads for nonterminals is by analysing the grammar to find the set of terminals that may follow the nonterminal. This is known as the nonterminal's *follow set*. The algorithm is straightforward, and may be found in any parsing textbook; we don't reproduce it here. It is important to note that follow sets include all tokens that can ever follow a nonterminal, regardless of where the nonterminal is used in the grammar. (This limits the power of this approach, as we discuss below.) In our example, a *methodName* is always part of a *methodCall*, so is always followed by an open parenthesis. A variable may be part of an assignment, in which case it is followed by an equals (**=**), or part of a declaration, in which case it is followed by a semicolon (**;**).

The class of grammars that can be parsed by a PDA using one token of lookahead derived from follow sets is SLR(1). The term *simple* evokes the relative ease with which the lookahead may be calculated. Figure 35 presents the complete SLR(1) PDA for our example. We show lookahead only on the state that needs it (State 28), but a conventional SLR(1) PDA would employ lookahead on all reductions, regardless of whether it removes a conflict.

The problem resolved by lookahead in this example was a reduce-reduce conflict. A similar problem occurs when an LR(0) state contains a reduction and one or more shifts. This is, unsurprisingly, known as a *shift-reduce conflict*. Figure 36 shows a grammar similar to that of Figure 31 that leads instead to a shift-reduce conflict: the LR(0) PDA cannot choose between reducing *variable* ::= id or shifting the

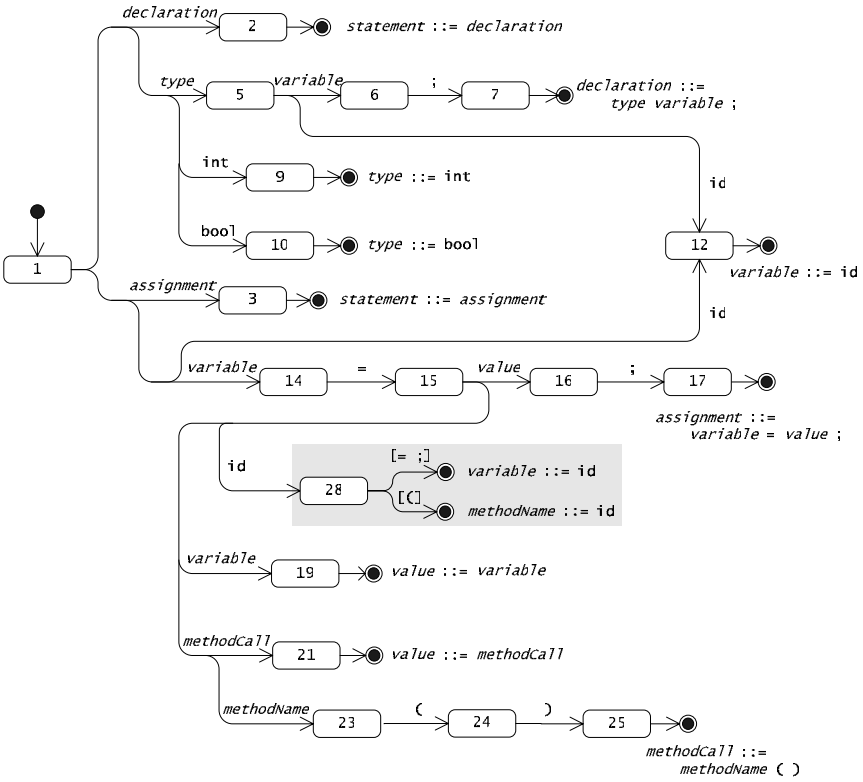

Figure 35: Deterministic SLR(1) PDA for grammar with methodCall

open parenthesis. An SLR(1) PDA resolves this problem in the same way it resolved the reduce-reduce conflict: lookahead is added to the reduction and shift actions, so the PDA can decide whether to reduce (on **=** or **;**) or shift (on **(**). State 27 in Figure 36 demonstrates the relevant fragment of the SLR(1) solution.

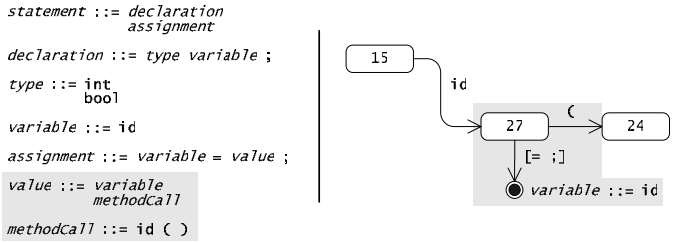

Figure 36: A grammar that produces an LR(0) shift-reduce conflict, and its resolution in an SLR(1) PDA fragment

3.3.5 LALR(1) parsing

We have seen how to add lookahead to an LR(0) parser in order to construct a more powerful SLR(1) parser. This additional power, however, is still insufficient to handle many grammars. Figure 37 presents a modified version of our example grammar, with changes from the grammar of Figure 31 highlighted.

We have introduced an *initializer* that follows a `variable` in a `declaration`. Consequently, the follow set of `variable` now includes an open parenthesis. Figure 39 shows the result of including this new lookahead in the SLR(1) PDA. When an open parenthesis is encountered in state 28, the PDA cannot choose between the two possible reductions; the state is now inadequate.

This problem is an artefact of using follow sets to determine lookahead. While it is true that an open parenthesis can follow a *variable*, it cannot do so in this context. State 28 is encountered only when parsing an *assignment*, a context in which an open parenthesis cannot occur. An open parenthesis follows a *variable* within a *declaration*.

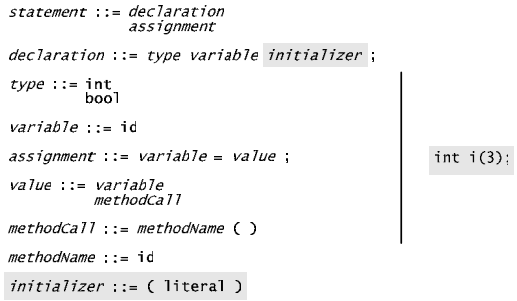

Figure 37: Grammar modified to require an LALR(1) parser

The solution is, of course, to calculate lookahead by finding which tokens can actually follow a

given reduction, taking context into account. We can do this by analysing the PDA to determine which states are reachable on a given reduction. All shifts from those reachable states provide the 1-lookahead for that reduction. From Figure 35, it can be seen that when state 28 reduces *variable* ::= **id**, it will pop to state 15, then goto state 19. State 19 will reduce *value* ::= *variable*, pop to state 15 again and goto state 16. The only way to proceed further is to shift on a semicolon, so the only possible lookahead is a semicolon. We can remove the equals and open parenthesis lookaheads from the guard condition in Figure 39.

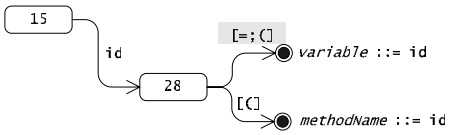

Figure 39: SLR(1) PDA fragment with inadequate state

The class of grammars that can be parsed by a PDA constructed in this way, using only lookahead possible in the context of individual states, is LALR(1). The term *lookahead* evokes the approach's emphasis on exact lookahead. The complete LALR(1) PDA for the given grammar appears in Figure 38. State 28 shows the resolved conflict. As with our earlier examples, we show lookaheads only where needed (on state 28), whereas a conventional LALR(1) PDA would use lookaheads on all reductions.

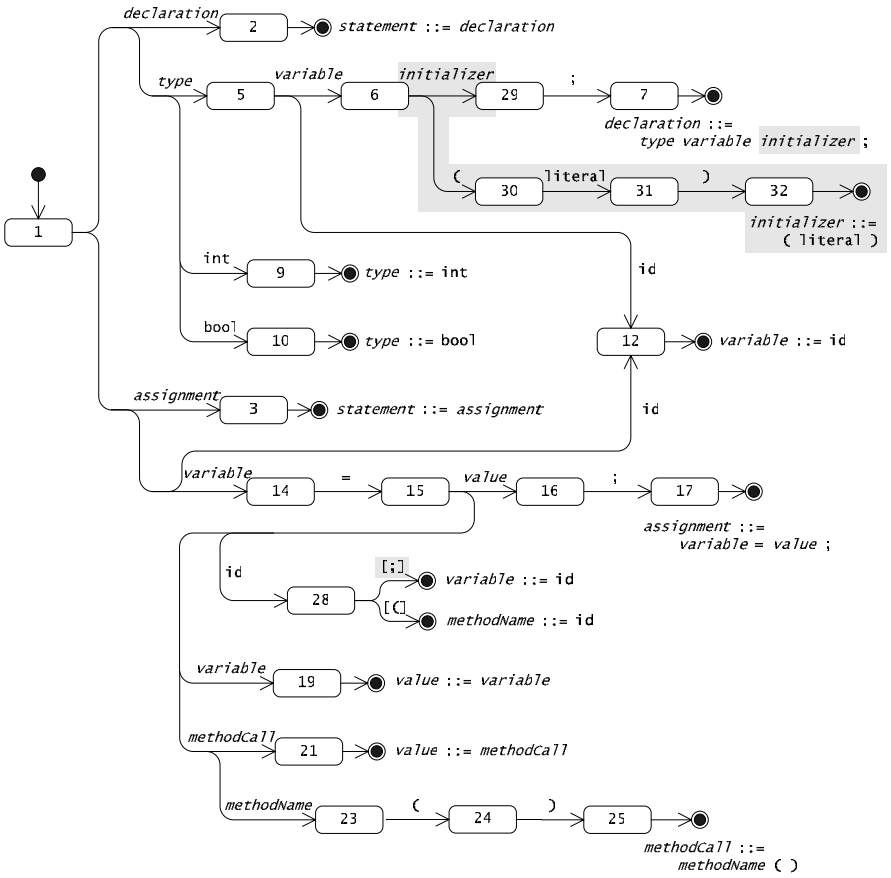

Figure 38: LALR(1) PDA

3.3.6 LR(1) parsing

For a given grammar, LR(0), SLR(1) and LALR(1) automata all share the same set of states. They differ only in their use of lookaheads on reductions. No further gains in power can be made by merely refining lookaheads. It is, however, not difficult to contrive a grammar that defeats an LALR(1) parser yet is still tractable using a PDA; such a grammar is presented in Figure 40.

```
statement ::= declaration
              assignment

declaration ::= type variable initializer ;
                type methodName ;

type ::= int
         bool

variable ::= id

assignment ::= variable = value ;

value ::= variable
          methodCall

methodCall ::= methodName ( )

methodName ::= id

initializer ::= ( literal )
```

int foo;

Figure 40: Grammar modified to require an LR(1)parser

This grammar is very similar to the previous example, in which the PDA had to choose between reducing an id as a *variable* or as a *methodName*. In this grammar, however, there are two places in which this choice must be made: while looking for a *value* inside an *assignment* (as before), and after a *type* in a *declaration*.

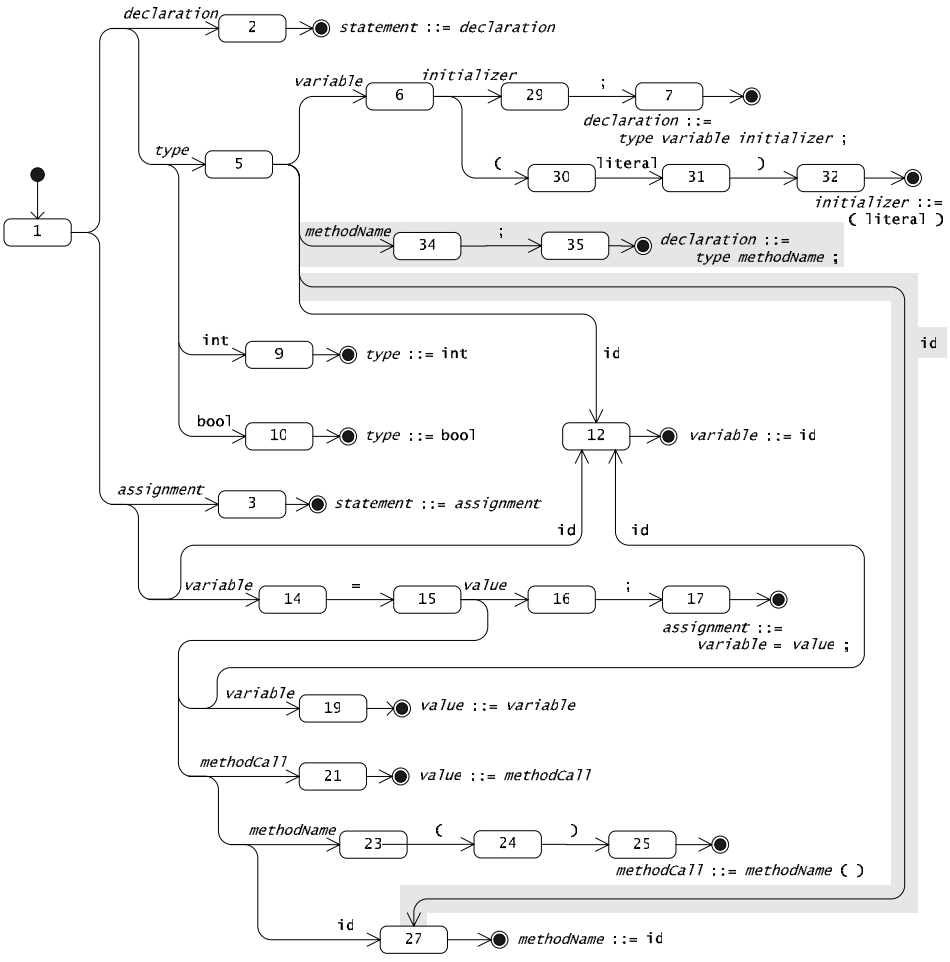

Figure 41: Nondeterministic LALR(1) PDA (ε–transitions removed)

Figure 41 shows the partially constructed PDA, at the point ε–transitions have just been removed. As

in our previous example, state 15 has two transitions on id, reaching states 12 and 27. The same situation now occurs in state 5: it has transitions on id to states 12 and 27. In order to make the PDA deterministic, we must merge states 12 and 27.

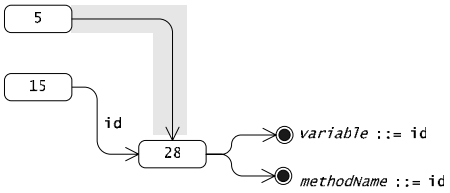

Figure 42: Reduce-reduce conflict reachable from two sources

In our previous examples, we did not encounter a situation in which the need to merge the same set of states occurred in more than one place. Had we done so, we would have applied the rule that sets of states should be merged uniquely. So, for example, we would merge states 12 and 27 to create a new state, and transition to that state from both places where the nondeterminism occurs (states 5 and 15), as shown in Figure 42.

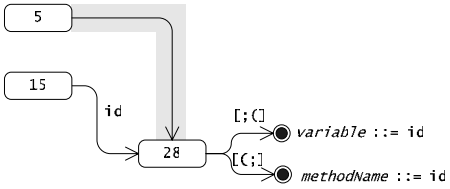

Figure 43: Inadequate LALR(1) state

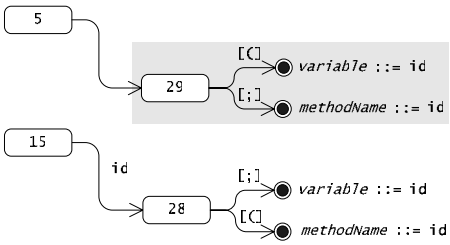

Figure 44: LR(1) PDA fragment

As expected, state 28 in Figure 42 has a reduce-reduce conflict. If we attempt to resolve this conflict by using LALR(1) lookahead, we find that, as in the previous example, from state 15 a *variable* will always be followed by a semicolon, while a *methodname* will always be followed by an open parenthesis. However, from state 5 the lookaheads are reversed: a *variable* will always be followed by an open parenthesis, and a *methodName* by a semicolon. When the lookaheads are combined, state 28 is once again inadequate, as shown in Figure 43. Thus, this grammar is not LALR(1).

This problem arose because we found two situations in which states had to be merged, and attempted to use a single state to handle both conflicts. The solution is, unsurprisingly, to split state 28 into two: one to merge states 12 and 27 for state 5, and the other to merge states 12 and 27 for state 15, as in Figure 44.

By using more states than an LALR(1) parser, we can avoid some conflicts, thus increasing the power of the approach. The class of grammars that can be handled by such parsers is known as LR(1). The complete LR(1) parser is shown in Figure 45.

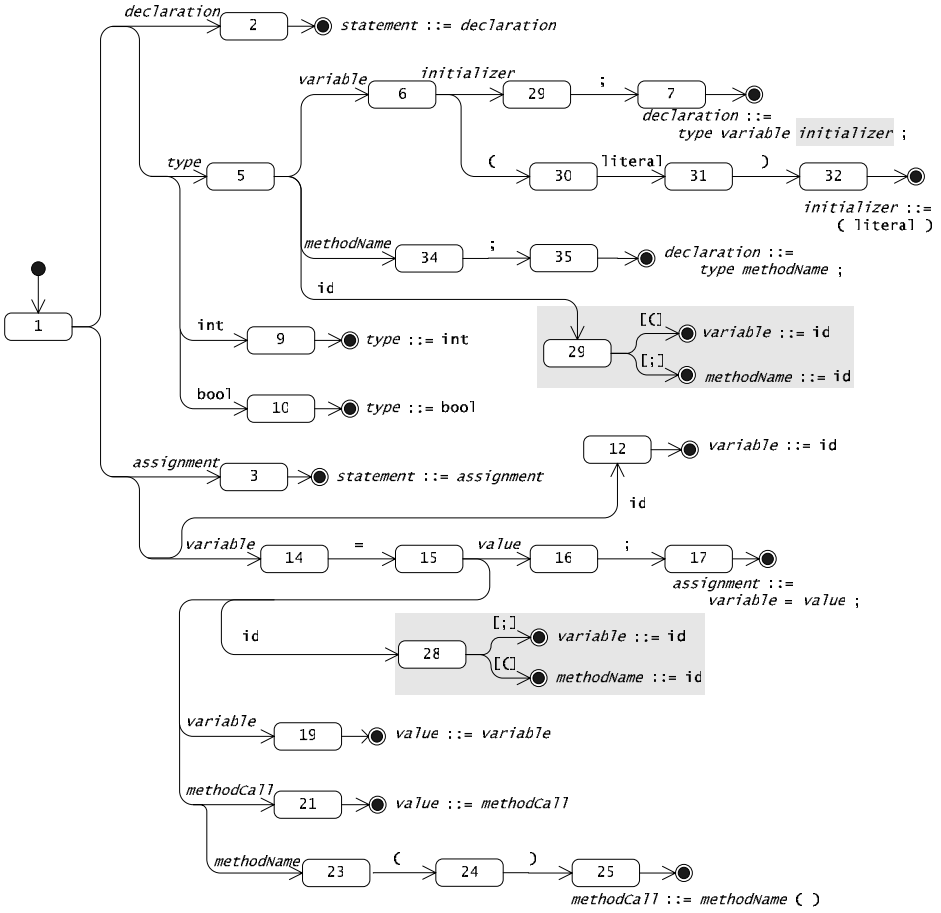

Figure 45: LR(1) PDA

It is worth noting that splitting of inadequate LALR(1) states can eliminate reduce-reduce conflicts, but not shift-reduce conflicts. All shifts will be replicated in the split states, and so will continue to conflict with the reduction lookahead in some (or all) of the split states.

As before, this example uses the more powerful parser features—lookahead and state splitting—only where needed, whereas conventional LR(1) parsers apply them universally. For the parser classes discussed prior to LR(1), this amounted to only a minor variation from tradition: we omitted lookahead where it was not necessary. For LR(1) however, the practice of splitting states only where necessary is a significant variation from standard practice. In fact, the usual way of constructing LR(1) and LALR(1) PDAs differs markedly from our approach. A conventional LR(1) approach in effect splits as many states as possible (without creating duplicates), often without eliminating conflicts. The result is typically a PDA very much larger than the LALR(1) PDA, often with little or no improvement in power. With our

approach, we need only split states that actually improve the power of the PDA. This avoids the proliferation of states that makes LR(1) unpopular, yet offers its full power.

```
statement ::= declaration
declaration ::= vModifier type variable initializer ;
              | mModifier type methodName ;
type ::= int
       | bool
variable ::= id
methodName ::= id
initializer ::= ( literal )
vModifier ::= static
            | volatile
mModifier ::= static
            | abstract
```

```
static int foo;
```

Figure 46: Grammar modified to require a GLR parser

3.3.7 GLR parsing

Unfortunately, it is easy for a grammar to fall outside even LR(1). Figure 46 presents such a grammar, in which a declaration begins with a modifier: `static`, `volatile`, or `abstract`. Variable and method declarations allow different modifiers, but `static` is common to both. (For simplicity, we have removed the *assignment* production from the grammar.)

This grammar produces the PDA shown in Figure 47, in which state 39 cannot choose which production to reduce, even using accurate 1-lookahead. No opportunity for splitting this state exists, because the conflict occurs in one context only.

If the lookahead were increased, an LR(k) approach would work for this grammar: the grammar is LR(3) but not LR(1). This is not a universal solution, how-

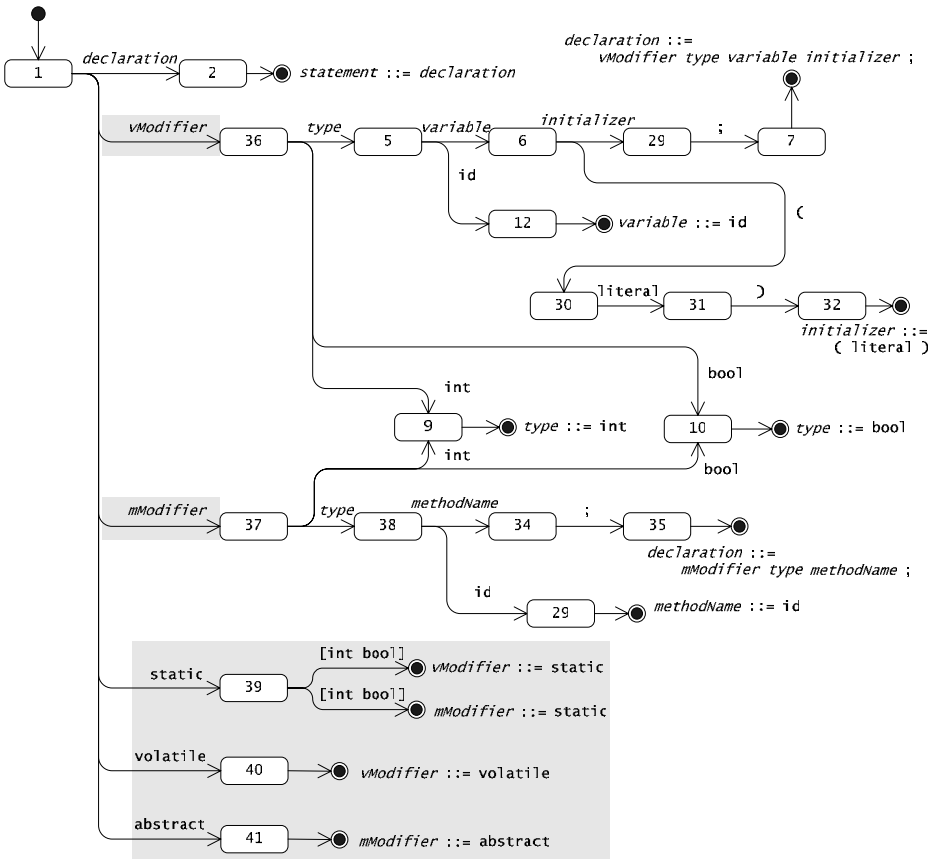

Figure 47: Inadequate LR(1) PDA

ever. If the grammar included recursive productions to allow any number of modifiers (as, for example, the Java standard grammar does), it would not be LR(*k*) for any *k*.

Although an LR(1) parser for this grammar cannot parse a sentence such as

```
static int foo;
```

a human reader can. The sentence does not contain an *initializer*, and so foo must be a *methodName*. The token static must be an *mModifier*.

LR push-down automata are constrained by their need to make decisions based on left context alone. When faced with a reduce-reduce or shift-reduce conflict, they simply give up. As we have seen, the ambiguity may be resolved at some future point, but the automaton does not persist that far. In contrast, a human who is reading the source from left to right is able (to some degree) to tolerate ambiguity, continuing to read the sentence until the ambiguity is resolved by later tokens.

The ability to pursue multiple paths through its state graph can be added to a PDA by allowing the stack to branch. This gives the PDA the ability to continue with all possible paths, until they cease to be viable. For an unambiguous grammar, all paths but one will eventually reach a dead end: they will arrive at a state from which there is no exit on the available input. The remaining path produces the correct parse

For example, Figure 48 shows the execution of the PDA from Figure 47, with the stack branching when a conflict is encountered. Both branches remain viable for several actions, but one branch eventually reaches state 6 with an input token of `;`. There are no actions possible in that state on that input, and so the branch is unviable and can be pruned off. The remaining branch (which has arrived at state 34) can shift on the available input, and the stack reverts to its usual (non-branched) form. (The example stops at the point a declaration is recognised; it will thereafter goto state 2 and reduce to state 1.).

Because inadequate states merely cause branching of the stack rather than outright rejection of the grammar, the use of a tree-structured stack improves the power of a PDA so that it can handle all context free grammars, (including ambiguous ones). This power comes at a price, however. Unlike a simple stack, which always designates one current state, a tree-structured

stack can designate any number of current states. While simple-stack PDAs always parse sentences in linear time (that is, time proportional to the length of the sentence), tree-

structured stack PDAs may exhibit performance that degrades as the input length increases. In the worst case, the number of current states (and branches) will increase with each token, requiring time exponential in the length of the sentence. Of course, this degradation can occur only with grammars that would be rejected by a simple-stack approach; otherwise the stack would not need to branch.

If two or more branches of the stack eventually arrive at the same state, they will subsequently produce identical behaviour in the automaton, until they pop back to earlier, distinct, states. This replicated behaviour can be used to mitigate the impact of stack branching. Rather than using a tree-structured stack, a *graph-structured stack* that allows both branching and merging can ensure that a set of unique states appear on top of the stack. For example, the stack configuration 5 of Figure 48 (which reaches state 9 via two paths) would be replaced by the graph-structured stack shown in Figure 49. The term *graph-structured stack* is due to Tomita [101];

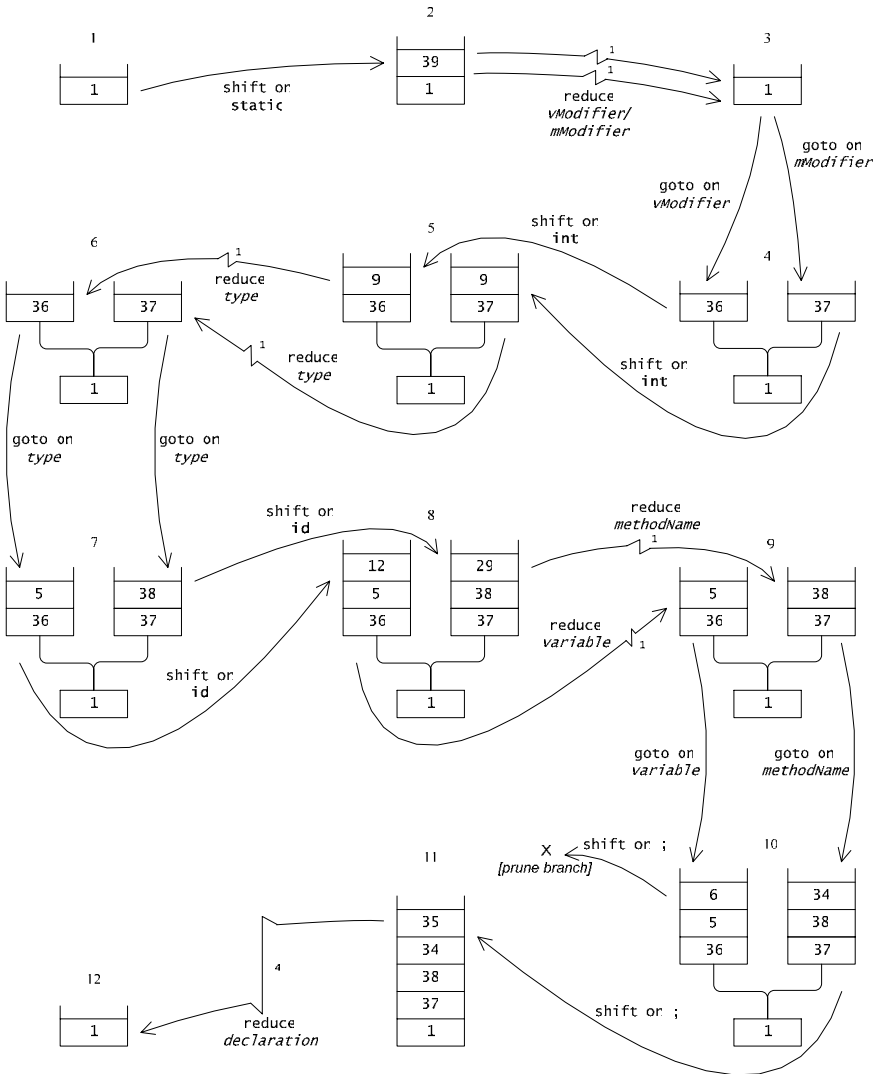

Figure 48: Execution of PDA with branching stack

the graph is in fact a directed acyclic graph (DAG). (We shall later encounter variants that allow cycles and therefore earn the label *graph-structured*.)

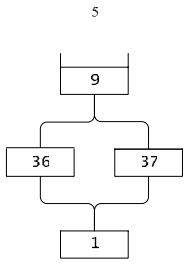

Figure 49: Graph-structured stack configuration

The use of a graph-structured stack incurs no loss of information, so the performance gain does not imply a loss of generality. In this example, when state 9 is popped, both underlying states are uncovered and the parse proceeds as in the original example, producing stack configuration 6 of Figure 48.

In practice, the use of a graph-structured stack allows a GLR parser to achieve close to linear performance for any realistic sentences conforming to a programming language grammar. For some sequences of tokens, repeated branching is possible, but such sequences are rare in actual source code.

3.3.8 Dotted items

Using an example, we have informally described a process of incrementally increasing the power of a parsing automaton. Our parser generator described in the next chapter behaves in a similar way (although the grammar doesn't change as it runs!). Conventional descriptions of parsing algorithms usually take a very different approach: they construct states from *dotted items* (also called *configurations*, or just *items*). Dotted items are productions adorned with a dot and lookahead. The dot indicates the position of the parser as it progresses along a RHS, and the lookahead indicates input tokens that must follow the dot. A single production in the grammar may lead to many dotted items, depending on the length of the RHS and the depth of lookahead. Figure 50 provides an example containing the three dotted items derived from the *super* grammar rule (of Figure 12) using lookahead depth of 1. PDA states are formed by combining dotted items that can be encountered at the same point in the parse to produce a canonical LR(*k*) PDA. Details may be found in [1].

```
[super ::= • extends classType,     {,implements]
[super ::= extends • classType,     {,implements]
[super ::= extends classType •,     {,implements]
```

Figure 50: Dotted items derived from super rule

3.4 Chapter summary

This chapter covers sufficient background information on parsing to establish our area of interest and provide a platform from which to begin a deeper discussion. We are interested in LR parsing because of its linear performance and power. Nevertheless, the currently dominant LR subclasses, LL(1) and LALR(1), are insufficiently powerful for our purposes, because we are concerned with parsing grammars without first transforming them.

We will show an improved LR parser generation approach that provides more power and flexibility than is currently available. These improvements make no difference to the theoretical power of the parser classes, but they extend the range of parsers that may be constructed in practice.

When these improved deterministic parsers nevertheless prove inadequate for a particular grammar, we advocate the use of GLR parsing. We find that GLR parsing need not be restricted to parsing natural languages, and is in fact well suited to the problem of parsing programming languages when grammar modification is to be avoided.

The use of more powerful (near-) linear parsing classes, and the philosophy of fitting the parsing technology to the grammar rather than the other way around, differs from normal practice. To some degree, these differences reflect different tradeoffs present in the static analysis domain than in a more traditional application such as a compiler. We are concerned with constructing a parse tree that faithfully reflects the standard grammar, while a compiler writer might sacrifice parse tree fidelity for the ability to define arbitrary parser actions and provide informative syntax error messages. Nevertheless, even in a traditional domain such as compiler writing, we recognise an historical over-emphasis on LL(1) and particularly LALR(1), and suggest that the alternative parser classes deserve more attention even for traditional applications.

Chapter 4

Yakyacc: Yet another kind of yacc

This chapter describes yakyacc, a new parser generator. We first reprise why it was necessary to write another parser generation tool rather than use existing ones, and note how yakyacc differs from and improves upon existing tools. Subsequent sections describe the design of yakyacc: its architecture and mechanisms for grammar input and parser output, and the object-oriented design of the parser generator, including its algorithms. Finally this chapter relates the contribution made by yakyacc to the existing parsing literature.

4.1 Yet another parser generator?

When yacc—the archetypal parser generator—was new, the name *yet another compiler compiler* was facetious; compiler compilers were still novel. Our use of *yet another* in yakyacc is literal (as well as derivative), acknowledging yacc as the progenitor of dozens of parser generation tools. Many examples can be found in catalogues of compiler construction tools, such as [74].

The profusion of parser generators available today reflects the widespread applicability of parsing technology in diverse software engineering settings. However it does not, as we have stated, indicate a correspondingly high diversity in parsing algorithms used in practice; the great majority of software engineering applications use LALR(1), LL(1), or LL(k) parser generators. Bison is the first widely-used parser generator to attempt support of GLR, and it

may serve as a vehicle for adoption of more powerful parsing, but as we have remarked the current implementation may be incorrect.

Parser generators vary not only in the parsing algorithms they implement, but also in their input and output formats: the languages used to describe grammars (typically variants of BNF) and the languages in which generated programs are written. Yacc, for example, reads a grammar defined using its own particular syntax—a mix of BNF-like rules, parser directives and embedded C code actions—and generates a parser in C code.

These variations in input and, especially, output formats are responsible for much of the proliferation of parser generators. A generated parser is a software component that is typically integrated into some larger application. Yacc, for example, is a natural choice for generating a parser to be integrated into software developments using C, but may be less suitable for other development cultures. In consequence, many variants of yacc exist primarily to accommodate different languages and platforms. Examples include yacc++, SML-yacc, perl-byacc, and pcyacc. Other LALR(1) parser generators—whether or not they explicitly share lineage with yacc—fill the same niche for different development environments.

The need to use different parser generators to target different execution environments is a regrettable complication, particularly when generators differ in their capabilities, input formats, user interfaces, and generated APIs. Yakyacc solves this problem. It offers greater flexibility than existing tools by decoupling grammar input and parser output from the parser generator itself:

- Instead of requiring use of a particular input syntax, yakyacc reads a generated XML grammar definition. This allows grammars to be supplied to yakyacc from unrestricted sources, by mapping them to yakyacc's XML input format. For example, a grammar defined in an EBNF variant, and another grammar using yacc syntax can both be automatically translated into yakyacc's XML input. (We develop these input translators using a yakyacc-generated parser.)

- Yakyacc generates an XML file that describes the parsing automaton. This XML file can then be transformed into a parser in any language. We use XSLT to achieve the transformation, so that by supplying a different stylesheet, a different generated

parser can be produced. As well as allowing us to generate code in any language, this approach also lets us vary the style of code, for example, by generating a table-driven parser or an OO state-based parser, or to generate a nondeterministic (GLR) parser when the underlying PDA is inadequate.

At the price of adding one more yacc descendant to an already large family, yakyacc's decoupled input and output formats reduce the need for continued multiplication of LR parser generators. However, the primary contribution of yakyacc is not its adaptability to different programming environments, but its ability to accommodate given grammars by applying suitable parsing algorithms.

As discussed earlier, grammar modification is undesirable, particularly for static analysis purposes. Yakyacc eliminates the need for grammar modification, instead adapting the choice of parsing algorithm to a given grammar. Yakyacc generates practical LR(0), SLR(k), LALR(k), LR(k), and GLR parsers as necessary. This is a broader range of bottom-up parsing algorithms than is available in any other single tool of which we are aware, and includes some powerful parsing classes that are rarely supported elsewhere. In particular, generators that can produce LALR(k) parsers for values of k greater than 1 are rare, and generators of practical (that is, minimal state) LR(k) parsers for any k other than 0 are unavailable, to our knowledge, although they are the subject of a number of publications and experimental tools, as noted in Section 4.4.

Yakyacc employs an *escalating* parsing algorithm that incrementally improves the power of the parser until it is adequate (or until inadequacies are shown to be unresolvable). Progressively more powerful parsing algorithms are applied to states of the PDA only as necessary to remove conflicts, producing a hybrid automaton with states of differing levels of power and employing heterogeneous lookahead depth. This approach enables the construction of parsers that are as simple as possible for the task they perform. More critically, it avoids the combinatorial explosions in state or lookahead space that previously made the more powerful parsing classes impractical. As we later explain, earlier authors have made some progress in the same direction, but so far without delivering to the software engineering community a parsing approach that improves on current practice enough to see widespread adoption.

The end result of our changed approach to parsing is the ability to parse any programming language according to a given grammar. The use of standard grammars for parsing allows software models and metrics to be defined and constructed more rigorously, because the syntax trees they describe conform to the standard. Additional advantages include the productivity gains made possible by not having to transform grammars, and reduced complexity in static analysis tools because phases can remain decoupled.

4.2 Yakyacc architecture

As explained in the previous section, yakyacc decouples the primary task of a parser generator—transformation of a grammar into a push-down automaton—from the tasks of processing a grammar description and generating code. This separation is reflected in the architecture of yakyacc and its supporting tools.

Figure 51 shows the sequence of transformations involved in producing a parser. In this example, a source document that uses BNF to describe a grammar is supplied. The document contains only the productions that comprise the grammar, such as might be taken directly from a language standard. Our `bnf2xml` utility converts the grammar to the XML format expected by yakyacc. Yakyacc reads it and emits an XML description of a PDA of appropriate power for that grammar. Finally, a stylesheet-driven transformation is applied by XSLT to produce source code for a parser.

The resulting parser may subsequently be compiled and, with a scanner added, used as a stand-alone program or combined

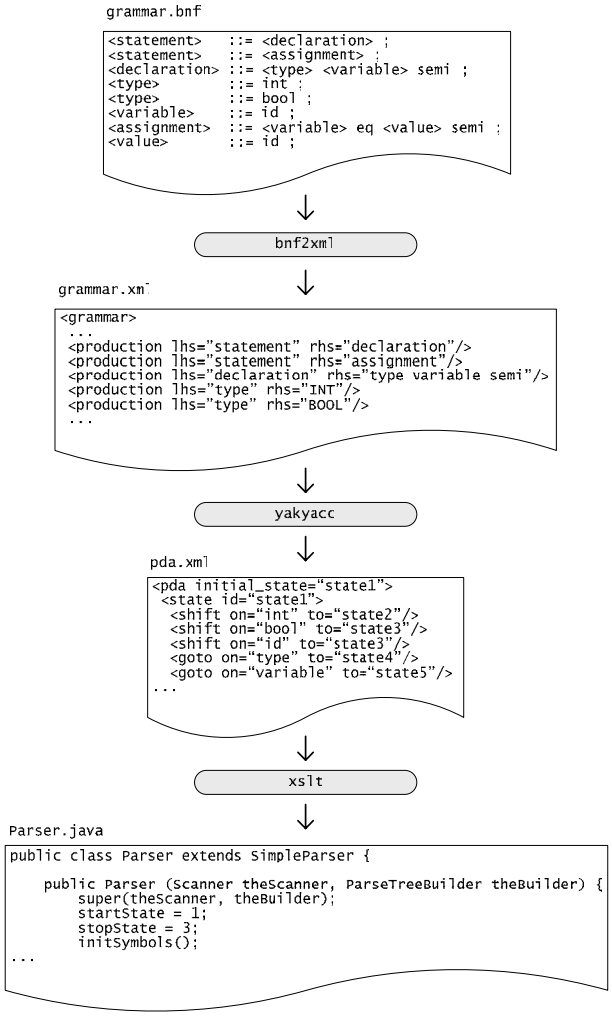

Figure 51: Decoupled architecture of yakyacc

into some other application. In our work, we typically use the resulting parser as the first step in a metrics and visualisation pipeline; the parser emits a parse tree as an XML document, as shown in Figure 52. Yakyacc itself does not directly participate in the pipeline. Rather, it generates a tool used in the pipeline.

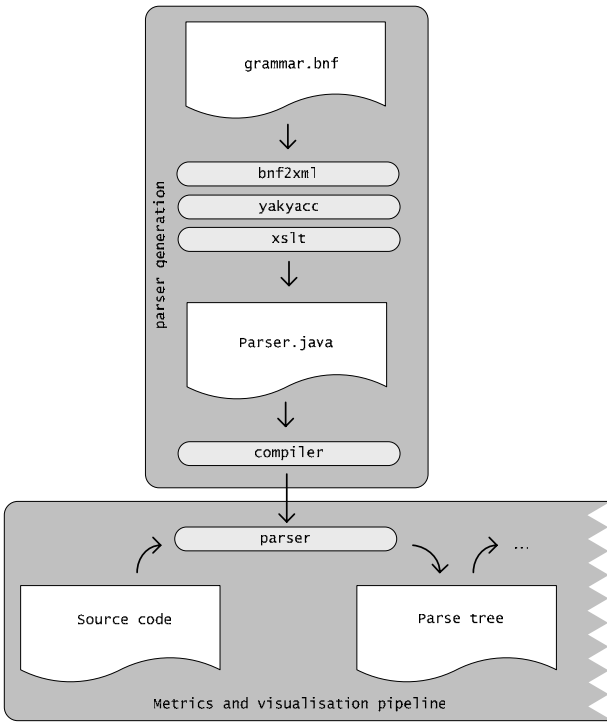

Figure 52: Generating a parser for use in a pipeline

The parser generation process as described above does not include user-defined parsing actions such as might be expected by users of yacc and similar tools. For our purposes, the only action to be performed by a parser is construction of a parse tree, and this can be done without user direction: the tree conforms to the grammar. The resulting data structure is a complete description of the parse; it may subsequently be traversed by other tools to perform arbitrary actions.

Our approach of automatically producing a parse tree does not imply a loss of generality, as custom actions may be performed at any later time while examining the tree. This allows developers freedom to change actions without interfering with parsing. In our research context, this means that code for tasks such as calculating metrics remains decoupled from parser generation and may be changed without regenerating the parser. The price paid for this flexibility is the space needed for storing parse trees, but this is a relatively minor consideration in most modern software environments. In our experiments on a corpus of open-source Java projects it has not produced any problems.

Although the ability to specify custom parser actions is unnecessary for our purposes, it is not *prevented* by yakyacc. Custom parser actions are irrelevant to the core task of transforming a grammar into a PDA, so yakyacc itself is not concerned with them. For completeness, however, we note that actions embedded in a source grammar could be extracted by a utility and supplied to the code generation phase, as shown in Figure 53. In this way, a source file written for yacc, for example, could be supplied to yakyacc and the generated parser would act as expected—even when the original grammar is intractable to yacc's LALR(1) approach. As we have no need of it, we have not yet implemented such a utility, although the design of the generated code makes provision for it through the *abstract factory* design pattern [35] for parse tree construction (see Section 4.3.1). A builder object could execute the custom parser actions rather than constructing the parse tree.

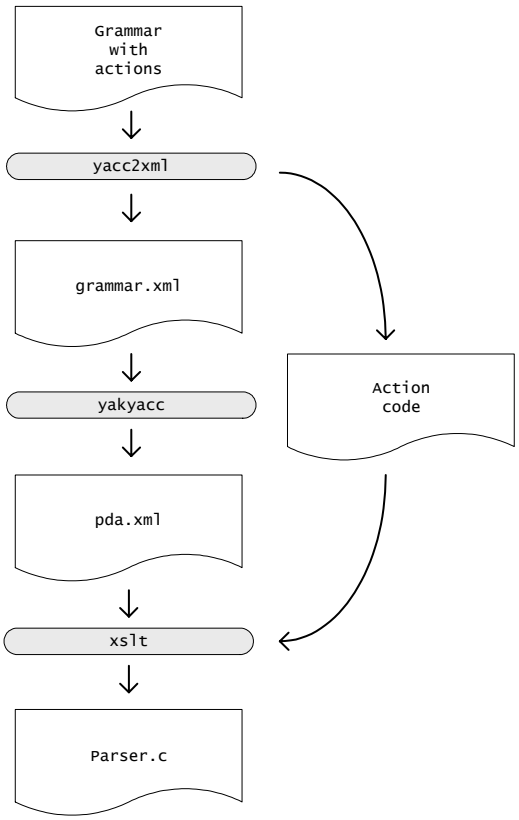

Figure 53: Potential mechanism for custom parser actions

4.2.1 Input of grammars

Grammars are commonly described in a variety of BNF dialects. Variations occur in the way terminals are distinguished from nonterminals (typographic differences, angle brackets around nonterminals, uppercase terminals, etc), lexical differences in punctuation characters (::=, →, etc), and syntactic differences (whether alternative RHSs are allowed without repeating the LHS, how productions are terminated, etc). BNF is also routinely extended (EBNF) to include syntax for alternate, optional and repeating clauses. These extensions allow more concise representation of a grammar, without improving on the expressive power of BNF.

Various parser generators specify their own syntax for grammar input, usually including features such as action specifications and directives to the parser generator. This requires the

users of a parser generator to manually convert grammars from whatever conventions were used in the original grammar into the conventions of the parser generator. This is a minor inconvenience compared to the task of manually transforming a grammar to fall within a particular parsing class such as LALR(1), but nevertheless an unnecessary one. By accepting an XML representation of a BNF grammar, devoid of action code and parser directives, Yakyacc is insulated from the variability of grammar representations, and allows an arbitrary number of such representations to be supported, by developing a translation utility for each grammar syntax.

Figure 54 shows the two such utilities developed so far. The first, bnf2xml, supports the original (and minimal) BNF syntax of Naur [75], and the second, ebnf2xml, adds syntax for alternation of RHSs (|), and for multiplicity of clauses (?, *, and +). Parentheses may be used in combination with any of the multiplicity suffixes.

EBNF syntax is accommodated by replacing the extended-syntax productions with standard BNF equivalents. A production containing optional clauses is replaced by a series of productions, each containing a unique combination of clauses. A repeating clause is handled by introducing an artificial nonterminal in place of the clause, and defining the new nonterminal as a left-recursive list of the original clause.

Yakyacc's input conversion utilities are themselves parsers, as they primarily exist to parse

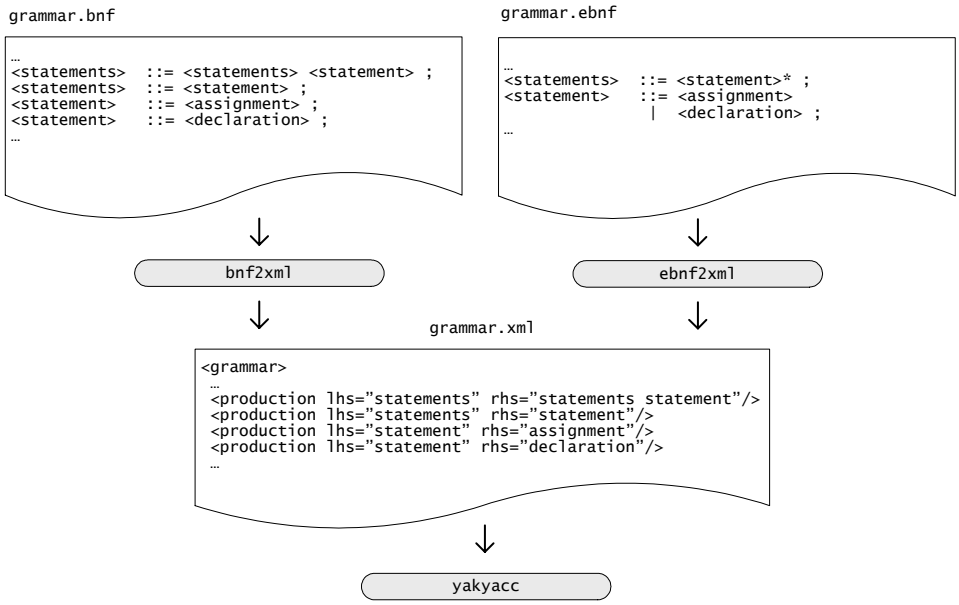

Figure 54: Alternative grammar specifications using BNF and EBNF

grammars. Consequently, they can be implemented using yakyacc, as shown in Figure 55. This is a simple matter of writing a grammar that describes some dialect of BNF (or EBNF), using yakyacc to generate a parser, and adding code to transform parse trees into the XML format expected by yakyacc. Parse tree transformation is done using the Visitor design pattern to traverse the tree and construct an equivalent XML document tree.

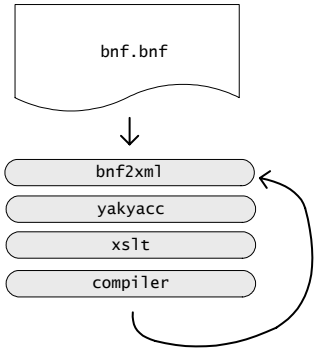

Figure 55: Using yakyacc to generate bnf2xml

4.2.2 Output of parsers

Yakyacc constructs a PDA capable of parsing the input grammar and emits it as an XML file, as shown in Figure 56. The use of XML here decouples PDA construction from code generation. Code is subsequently generated by transforming the XML file in a manner specified by a stylesheet; one example is shown in the figure.

The use of stylesheet-driven code generation means that for any one PDA, a parser may be generated in any language and implementation style (e.g. table-driven or graph-driven, deterministic or nondeterministic) simply by providing a suitable stylesheet. This eliminates one of the main causes of the current proliferation of parser generation tools: the need for

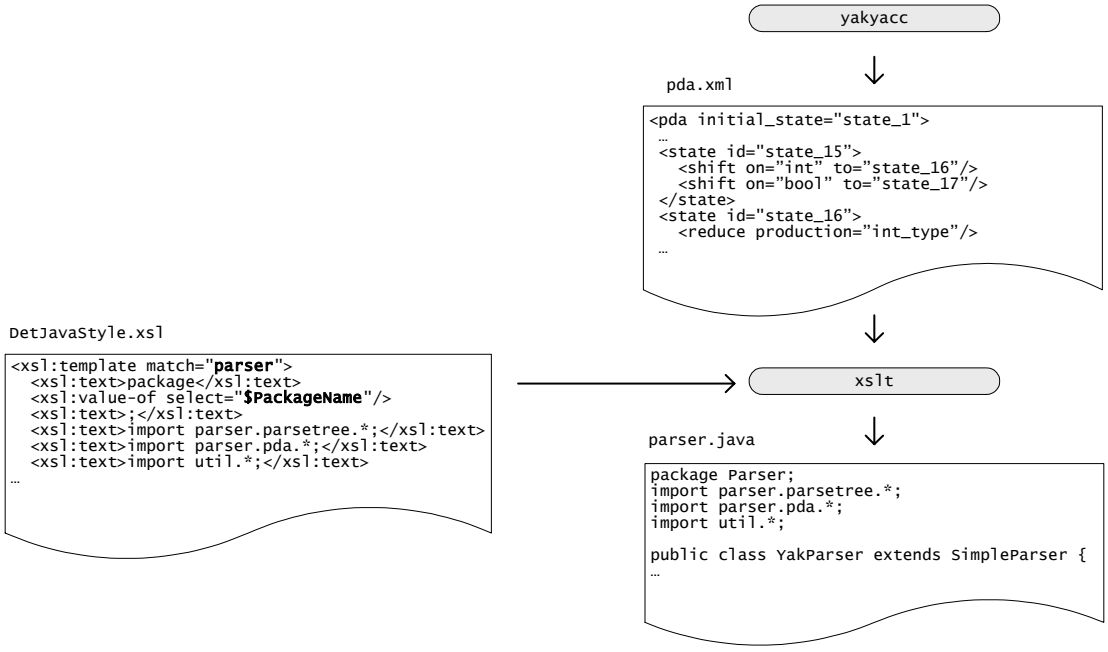

Figure 56: Parser generation using a stylesheet

parsers to be implemented in a variety of languages and contexts. Rather than having to develop a whole new parser generator for some new target implementation, we need only a new stylesheet. This approach also provides users with the ability to tailor the products of parser generation to their own needs.

4.3 Yakyacc design

For a given grammar, yakyacc first attempts to generate a parser with the weakest approach, and then tries progressively more powerful approaches until an adequate (i.e. conflict free) parser is found, or no more improvements can be made. In this way, yakyacc produces a hybrid PDA: different states are created and refined by differing parsing algorithms, as required.

Parser generators can be very complex programs. Combining a range of parsing algorithms into a single tool is potentially more complex still. Our approach endeavours to keep complexity in check by factoring out the common features of the different parsing algorithms into an inheritance hierarchy.

Much of the parsing literature was developed when procedural programming was the dominant paradigm, so object-oriented descriptions of parsing algorithms are relatively rare. (Holmes [43] describes an object-oriented implementation of a Pascal compiler, but the actual parsing is performed by automaton generated by yacc.) Moreover, the parsing literature generally favours formal abstraction over software engineering imperatives such as meaningful variable names and maintainable designs. The design of yakyacc addresses these concerns by providing a new OO implementation that is intended to better meet the needs of current software engineers.

4.3.1 Runtime PDA

The purpose of yakyacc is to construct an executable PDA; we call this the *runtime PDA*, to distinguish it from the dynamic model of a PDA produced by Yakyacc as it progresses through PDA configurations. (The final version of the yakyacc PDA is generated to become the static structure of the runtime PDA.)

A runtime PDA may be implemented in a variety of ways, as specified by a stylesheet. One approach is to embed into the stylesheet the complete code needed to make a PDA. This approach is supported by our architecture and has been implemented, but it has the disadvantage of generating redundant code; each parser contains the 'boilerplate' code of the runtime parser. This minor inelegance becomes a real problem if more than one yakyacc-generated parser is needed in a single program. An alternative is to capture the common code in a library that is reused by generated parsers.

In this section, we describe such a library, written in Java. Our reason for describing the runtime PDA at this point is that the yakyacc PDA – which is our primary concern in this chapter – depends on it, as explained in the next section.

Figure 57 depicts the hierarchy of packages in the runtime library. The main responsibilities of these packages are:

- `parser`: Contains all runtime parser code.
- `grammar`: Represent a context-free grammar.
- `parseTree`: Represent syntax trees produced by parsing.
- `tokenFactory`: Construct individual tokens. Elsewhere these will be used as input for parsing and become leaves in a parse tree.
- `treeFactory`: Construct parse trees.
- `visitor`: Define visitors for accessing parse trees.
- `pda`: Implement a variety of Push Down Automata (graph-driven and table driven, deterministic and nondeterministic).
- `tokenStream`: Provide a stream of tokens as input for parsing.
- `stateMachine`: Implement the state machine of a PDA.

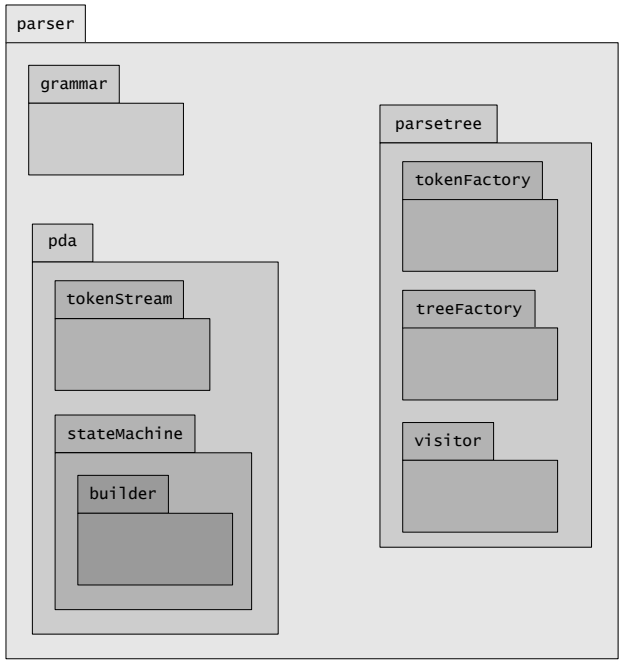

Figure 57: Runtime package structure

- `builder`: Construct a state machine.

4.3.1.1 Modelling grammars

Strictly speaking, grammars do not need to be modelled within the runtime component of a parser; the grammar served its purpose by being transformed into a PDA, which captures all the information essential to parsing. However, we chose to model grammars in the runtime as they provide a valuable reference structure that defines terms used elsewhere in the runtime. In particular, actions performed by a PDA may shift a `Terminal` defined by the grammar, or reduce a `Production` defined by the grammar. Likewise, each `Token` in the input stream will correspond to a `Terminal` and a branch node will correspond to a `Production`.

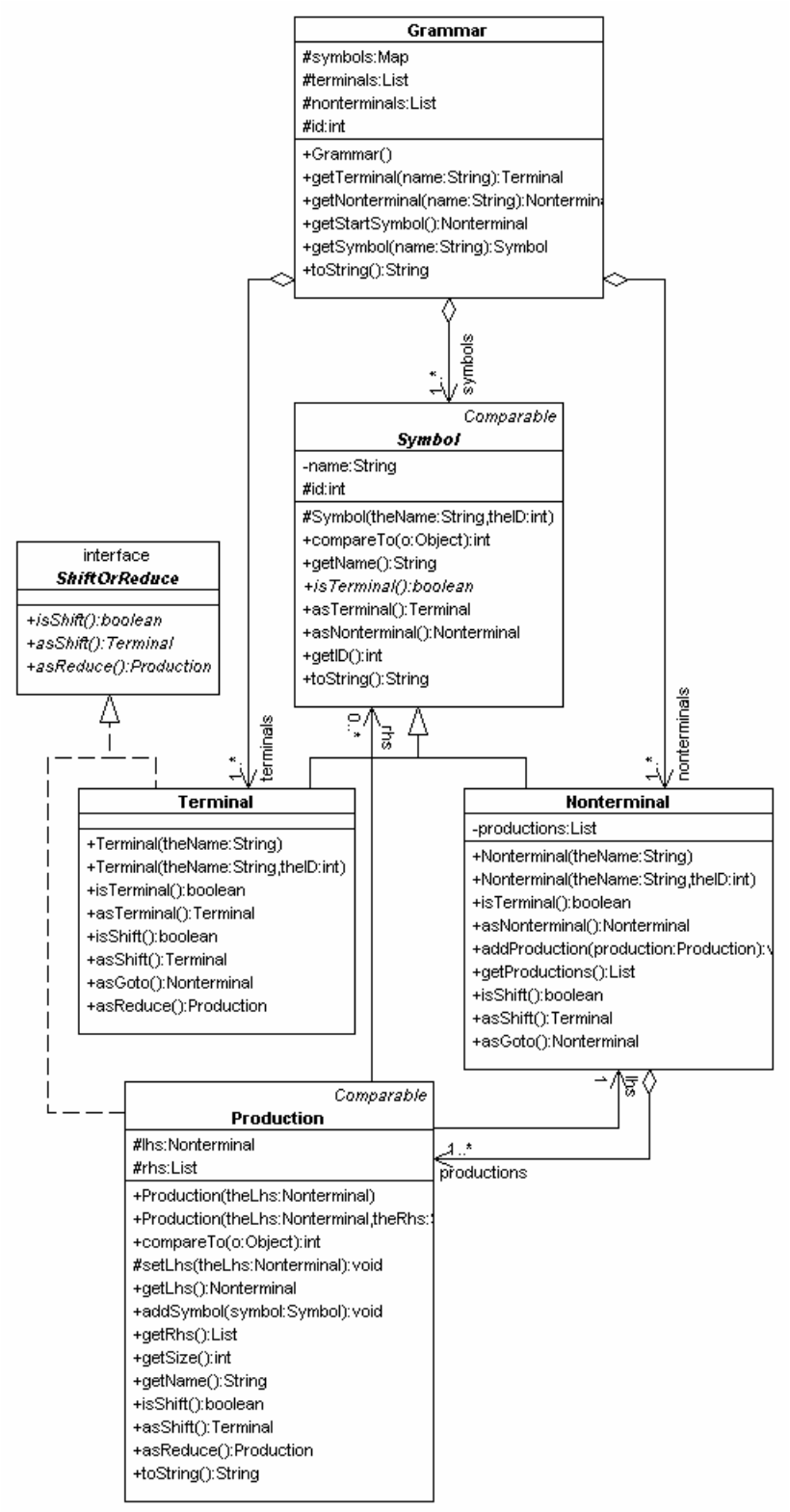

Figure 58: Grammar classes

As can be seen in Figure 58, a Grammar contains a list of Terminals and a list of NonTerminals. The first Nonterminal is defined to be the start symbol.

Terminal and Nonterminal inherit from Symbol, which defines a name attribute. A Grammar is responsible for constructing all of its Symbols and ensuring that they are uniquely named; it maintains a Map of symbols indexed by name for this purpose. Symbols are also given a unique integer identifier (id) when they are constructed. These numeric identifiers are provided as a convenience for table-driven parser generators, which may use them to index tables; they play no role in PDA construction.

A Nonterminal contains a list of Productions for which that Nonterminal is the lhs. Each Production refers to list of Symbols that comprise its rhs.

Finally, ShiftOrReduce provides a common interface for the two parser actions that may be initiated by a state: a Terminal specifies a shift action and a Production specifies a reduce action.

4.3.1.2 Modelling parse trees

Figure 59 shows the parse tree design. The core structure is a variant of the *composite* design pattern [35]; a SyntaxTreeBranchNode is a composite of other SyntaxTreeNodes. The top-level node types are defined as interfaces in order to allow arbitrary parse tree implementations. Our default implementation appears in the same figure.

A SyntaxTreeNode spans a contiguous sequence of input Tokens – zero or more in the case of branch nodes, exactly one in the case of leaf nodes. Any SyntaxTreeNode that contains at least one Token can provide the line and column position of the first Token in the sequence. The concatenated string value of all Tokens in the sequence can be provided by the getValue() method. These methods are useful for communicating information about parse trees, such as syntactic metrics, back to a programmer. Calling getValue() on the root node of a parse tree will return the original source code, minus any whitespace.

The input supplied to a parser is a stream of Tokens, and these Tokens become the leaves in the resulting parse tree. Every Token has a terminalType, which has a reference to a Terminal in the grammar. We differentiate between a ValueToken, which carries a string

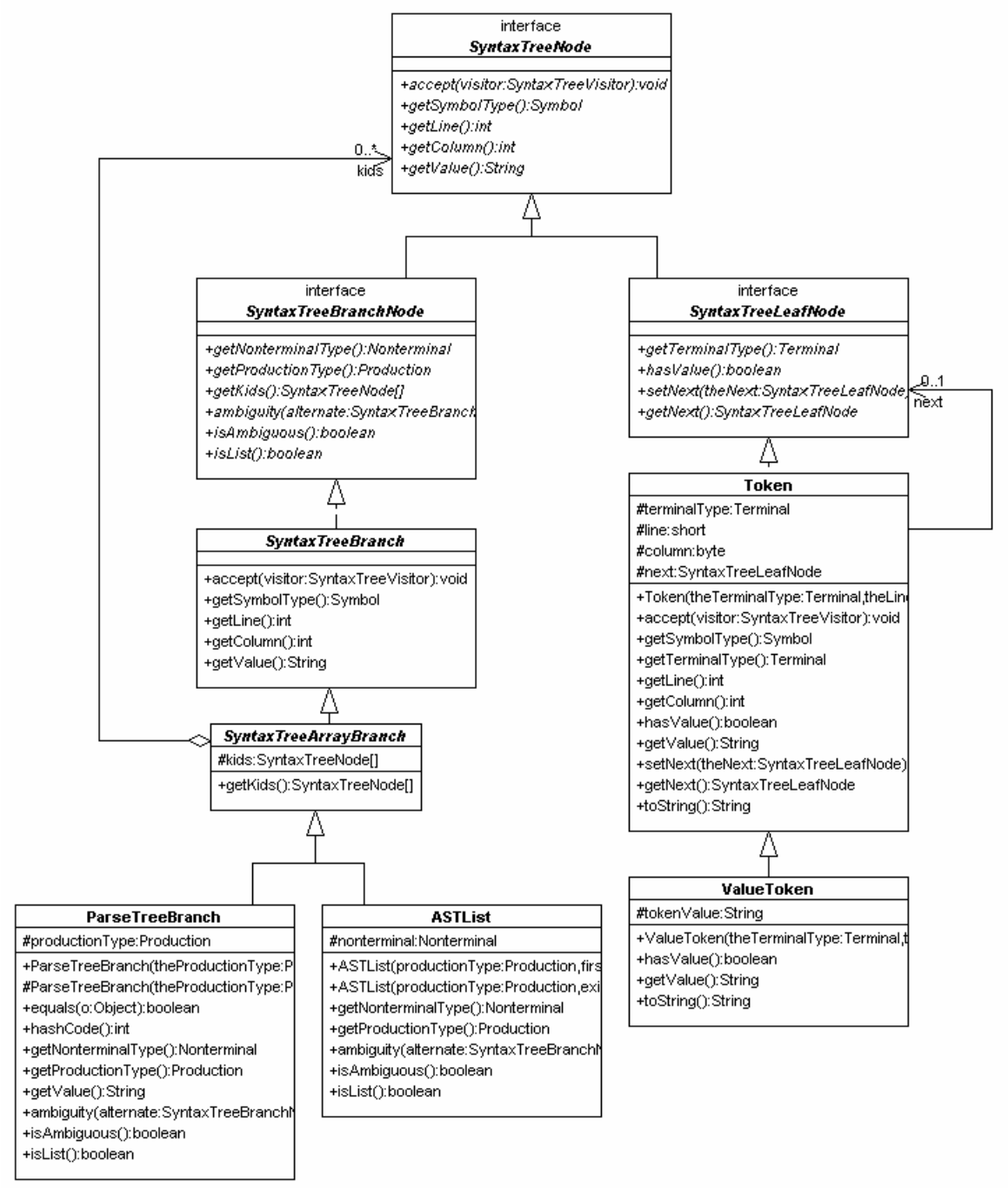

Figure 59: Parse tree classes

value, and a standard `Token`, which does not. The former is necessary for representing tokens such as identifiers. During parsing, we need only know the type of token—that it *is* an identifier—and not its name, but the name will become important during semantic analysis. A `ValueToken` simply carries the string value so it will be available to all subsequent phases of static analysis. For other tokens (such as Java's `int`, `=`, `for`, etc), the value is implied by the token type and need not be carried as it can be retrieved from the `Terminal`.

A branch node in a tree may be either a `ParseTreeBranch`, which corresponds exactly to some production in a BNF grammar, or an `ASTList` (Abstract Syntax Tree List), which allows trees produced by recursive BNF productions to be compressed into lists. Figure 60 provides an example of a recursive production in a grammar, with two parse trees that might be produced. The first uses `ParseTreeBranch` nodes to reflect the recursion. The second is the flattened `ASTList` alternative. The latter structure is particularly useful when a grammar was originally defined using EBNF, because the parse tree can conform to the grammar author's expectations.

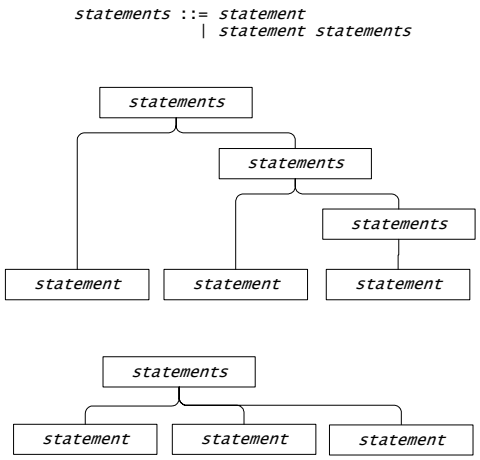

Figure 60: Parse trees for recursive grammar

The result of parsing an ambiguous sentence is often described as a *parse forest*. This term suggests the construction of multiple complete trees. In fact, ambiguities can always be localised to sub-trees within a singly-rooted tree. Ambiguity is the result of having two or more viable productions with the same LHS describing the same sequence of tokens. The ambiguous sub-trees appear identical to all higher nodes in the tree, and so can be packaged into a single ambiguous tree node, making a data structure known as a *packed parse forest*. In fact, since our parsing algorithm also shares identical sub-trees, the data structure is a *packed shared parse forest* and the parse tree is actually graph-structured.

In our parse tree design, we model ambiguous and non-ambiguous nodes using the same `SyntaxTreeBranchNode` interface. The internal state of any `ParseTreeBranch` object indicates which it is. We differentiate between ambiguous and non-ambiguous nodes using the object's state rather than its class because in GLR parsing algorithms a node that was originally unambiguous may later become ambiguous as parsing progresses. If we were instead to replace the old unambiguous node object with a new ambiguous one of a different class, we would have to update all references to the old node. Our solution differs from that used by Rekers [91], which employs a bipartite tree to separate each branch node into two levels: a (potentially) ambiguous node containing one or more unambiguous nodes. In Reker's al-

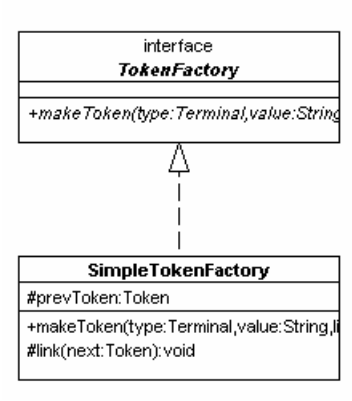

Figure 61: Token factory classes

gorithm, references are kept only to the higher, ambiguous, node, so alternative sub-nodes can be added later. Our design achieves the same effect, and since ambiguity is relatively uncommon in real languages, produces much more compact trees.

Token construction is handled by the `tokenFactory` package, shown in Figure 61. The default implementation chains `Tokens` together (each `Token` knows its successor) so that the original token sequence is retained. This allows whitespace and comments to be retained in the token sequence, even though they are filtered out before parsing and consequently omitted from parse trees. This feature is useful for calculating lexical metrics such as LOC or comment density. The original text of a program can even be recovered by walking the complete Token sequence.

The details of constructing parse trees are hidden by using an *abstract factory* pattern defined in the `treeFactory` package. Figure 62 shows the classes. This factory hides whether the tree under construction is abstract or concrete, and also whether it allows ambiguities. Ambiguous parse trees cannot be produced by a deterministic parser, and in these cases we use a factory that does not support construction of ambiguous sub-trees. Conversely, nondeterministic parsers require the use of a factory capable of making ambiguous sub-trees, or else will produce only one possible parse tree.

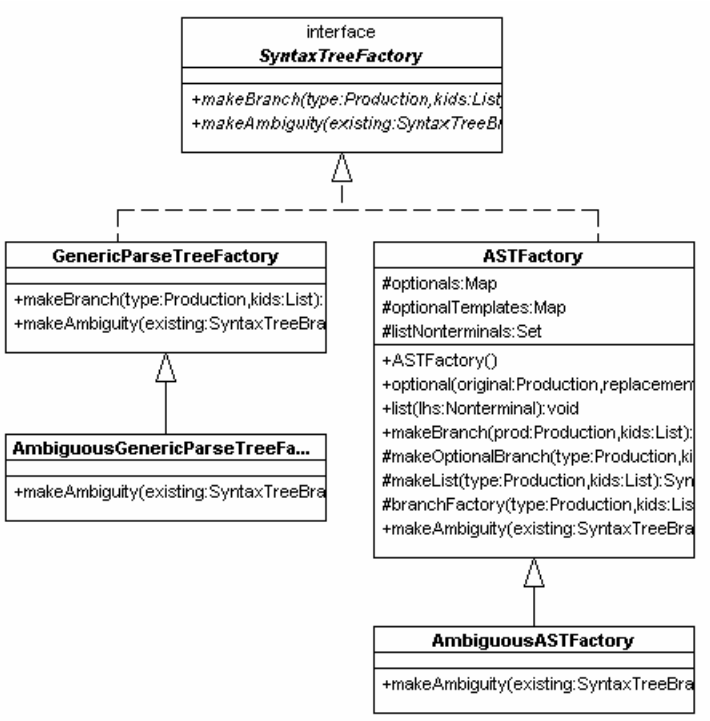

Figure 62: Tree factory classes

By default, `ASTFactory` works like a normal parse tree factory: it builds parse tree nodes that exactly reflect productions in

the BNF grammar. However, it can be configured to translate some reductions into abstract syntax tree nodes that model the lists and optional clauses found in EBNF. Configuration methods of `ASTFactory` specify which translations are to be performed: recursive productions flattened into lists and null symbols inserted to emulate productions with optional clauses.

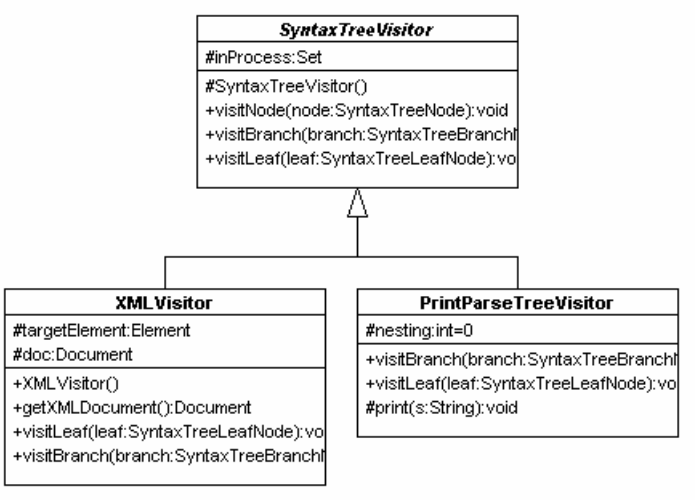

Figure 63: Parse tree visitors

Parse trees contain information that may later be used in a variety of ways, such as when building a semantic model or calculating software metrics. In order to allow access to parse tree information, while keeping coupling in check, the *visitor* design pattern is used. Figure 63 shows the design, with two concrete visitors. One visitor emits the parse tree as an XML file (as used in our pipeline), and the other prints it with indenting. Other visitors may be added as necessary for purposes such as calculating syntactic metrics.

We now discuss the main runtime `pda` package, It is more complex than those described so far, so we present it incrementally.

4.3.1.3 Main concepts

Figure 64 shows the primary concept, `Parser`, that represents an executable PDA that can transform a given `TokenStream` into a parse tree, via the `parse()` method.

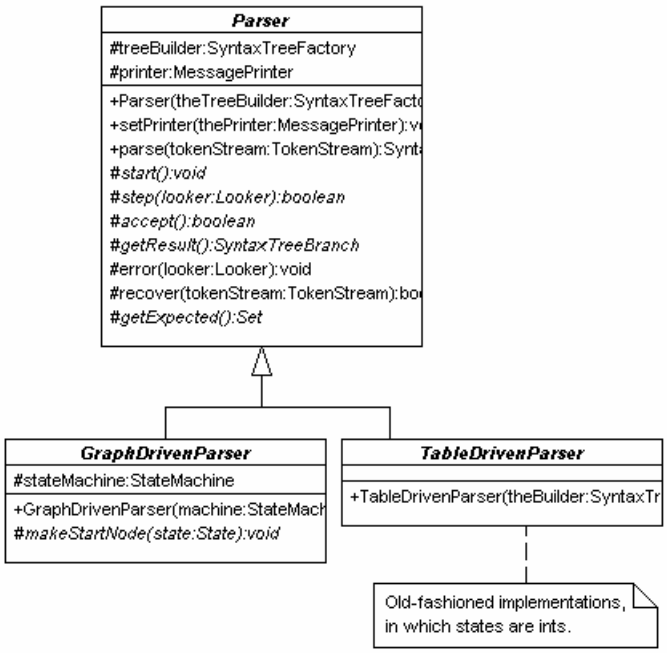

Figure 64: PDA base classes

Historically, parsers have been implemented in a procedural style, using two-dimensional tables as the data structures that describe the PDA graph. In that approach, states are represented by integer identifiers, which are used as indexes into tables to find state transitions and reduce actions. Implementations that use this approach are encompassed by the `TableDrivenParser` abstract class. In object-oriented designs, however, states are better represented as objects, responsible for their own transitions and reduce actions, so no table is necessary. The states and their relationships form a graph data structure; `GraphDrivenParser` represents the approach, but it might equally have been called *state-driven*, as the essential concept is that states are objects.

In this discussion `GraphDrivenParser` is the more important of the two approaches because, as we explain in the next section, yakyacc itself uses a specialised version of `GraphDrivenParser` for constructing PDAs. Table-driven parsers are provided as a runtime library option. A code generation stylesheet may choose to use a graph-driven or table-driven implementation from the library (or may independently generate its own variant of either approach).

Although table-driven parsers are not essential for our purposes, we have implemented them because they provide a useful basis for comparison and testing of our less conventional graph-driven versions. Using traditional parser implementations allows us to verify the behaviour of experimental parser implementations by running back-to-back tests.

The `Parser` class captures common aspects of the table-driven and graph-driven implementations, and in particular the `parse()` method, shown in Figure 65. This method accepts a `TokenStream` and repeatedly invokes the (abstract) `step()` method to process one token of input at a time, until the parse is complete.

```
public SyntaxTreeBranch run(TokenStream tokenStream) {
    reset();
    boolean ok = true;

    while (ok && !accept()) {
        if (step(tokenStream))
            tokenStream.advance();
        else {
            error(tokenStream.current());
            ok = recover(tokenStream);
        }
    }

    return ok? getResult() : null;
}
```

Figure 65: The `parse()` method of `Parser`

The `Parser` class also provides a minimal default implementation of error handling. The `error()` method reports a message with the offending token, including its line and column. Subclasses improve on this message by including a list of expected tokens. (Although not evident in the figures, `Message` and `Mes-`

sagePrinter classes are used to decouple the Parser from the user interface.) The default recover() implementation merely returns false. This is adequate for our purposes of parsing syntactically correct code, but for general purpose parsing, subclasses would have to override the default behaviour with more sophisticated variants.

4.3.1.4 Table-driven parsers

Figure 66 shows our two existing table-driven parser implementations. Like other table-driven parsers they are limited to $k = 1$ to keep space requirements in check. SimpleParser is a deterministic table-driven parser. RekersParser is a nondeterministic table-driven parser that implements the corrections made by Rekers [91] to Tomita's GLR algorithm.

The code of SimpleParser is much the same as can be found in deterministic LR parsers elsewhere. The whole class is only about 100 lines of code. The step() method appears in Figure 67.

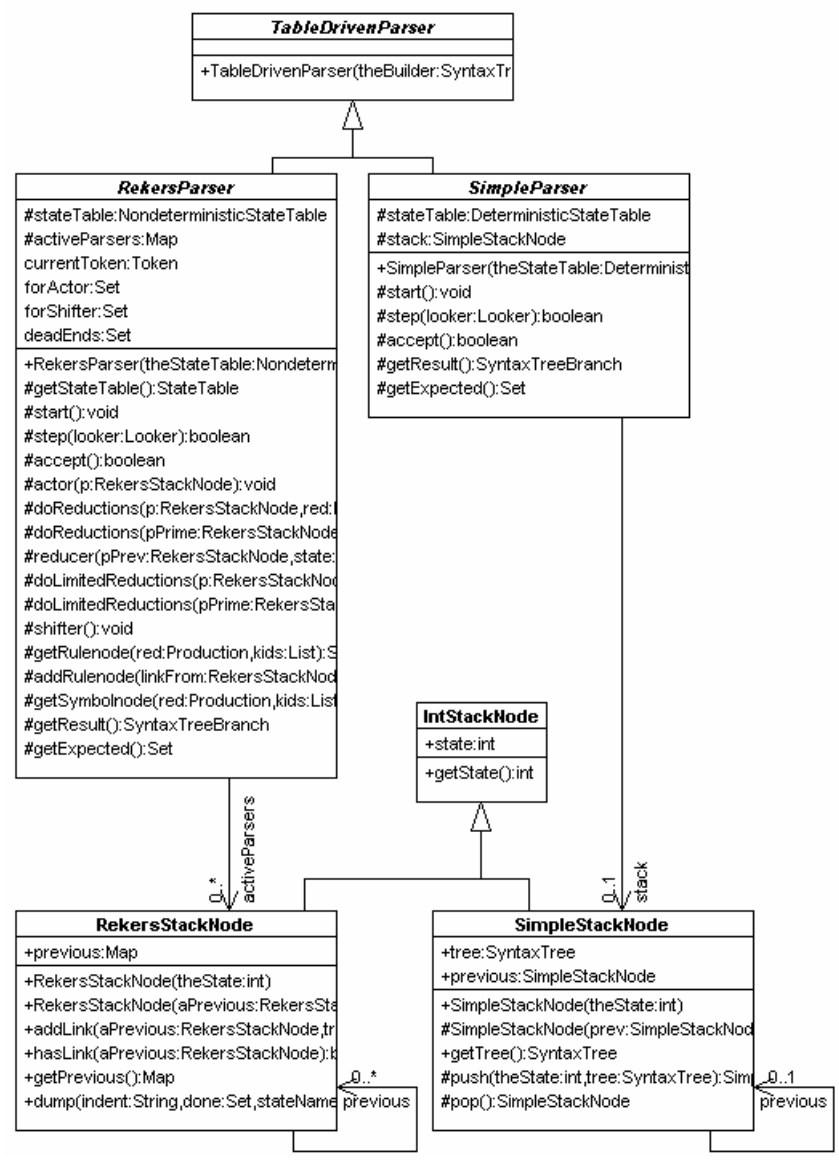

Figure 66: Table-driven parsers

The stack used by the deterministic parser is a singly-linked list implemented by `SimpleStackNode`. Each node stores a state and a parse tree, and is otherwise 'dumb', with little more than getter/setter methods.

```
protected boolean step(Looker looker) {
    Token token = looker.first();
    Symbol tokenType = token.getSymbolType();
    if (tokenType == null)
        return false;

    Production reduce = stateTable.getReduction(stack.getState(), tokenType);

    while (reduce != null) {
        List kids  = new LinkedList(); // SyntaxTreeNodes

        for (int i = reduce.getSize(); i > 0; i--) {
            kids.add(0, stack.getTree());
            stack = stack.pop();
        }

        SyntaxTree branch = treeBuilder.makeBranch(reduce, kids);
        Symbol gotoSymbol = reduce.getLhs();
        int gotoState = stateTable.getTransition(stack.getState(), gotoSymbol);
        stack = stack.push(gotoState, branch);
        reduce = stateTable.getReduction(stack.getState(), tokenType);
    }

    int shiftState = stateTable.getTransition(stack.getState(), tokenType);
    if (shiftState > 0) {
        stack = stack.push(shiftState, token);
        return true;
    }

    return false;
}
```

Figure 67: The `step()` method of `SimpleParser`

The nondeterministic `RekersParser` is more complex because it must support conflicting actions (but still has fewer than 400 lines of code). `RekersParser` uses a Graph Structured Stack (GSS) implemented by `RekersStackNode`. These GSS nodes are like `SimpleStackNodes` except that a node can have multiple predecessors. However, parse trees must be stored differently. In `SimpleStackNode` it was sufficient to store a parse tree directly in each node. A node could have only one predecessor, so it was clear that the tree applied to the transition from the predecessor node to the current node. A `RekersStackNode`, on the other hand, can have many predecessors, each with a different parse tree. Logically, the parse trees can be viewed as being on the links between stack nodes, rather than in the stack nodes themselves. We use a `Map` to store `previous` stack nodes and their associated trees.

The GLR parsing algorithm described by Rekers is shown in Figure 68. Our implementation translates this pseudocode into Java (Rekers uses Lisp) and adapts it to our framework. Rekers' terminology differs from ours: a *parser* is a node on the stack, and an *active parser* is on top. A *rule node* is a non-ambiguous parse tree branch and a *symbol node* is an ambiguous one. The `REDUCER` procedure actually performs a goto action. Within our `RekersParser` class we use Rekers' terminology.

Rekers' PARSE procedure corresponds to our parse() method inherited from Parser, and is therefore omitted from RekersParser. PARSEWORD corresponds to step(), and this is where Rekers' algorithm is fitted into our framework. If step() is considered the 'top' interface, then the 'bottom' returns control to our framework by replacing Rekers' parse tree construction code with calls to our ParseTreeFactory. If we supply our usual factory implementation, we induce Rekers algorithm to construct our usual parse tree structure, without needing to significantly change Rekers' approach.

```
PARSE(Grammar, a1 ... an) :
    an+1 := EOF
    global accepting-parser := 0;
    create a stack node p with state START-STATE(Grammar)
    global active-parsers := { p }
    for i := 1 to n + 1 do
        global current-token := ai
        PARSEWORD
    if accepting-parser != 0 then
        return the tree node of the only link of accepting-parser
    else
        return 0

PARSEWORD :
    global for-actor := active-parsers
    global for-shifter := 0
    while for-actor != 0 do
        remove a parser p from for-actor
        ACTOR(p)
    SHIFTER

ACTOR(p) :
    forall action E ACTION(state(p), current-token) do
        if action = (shift state') then
            add <p, state'> to for-shifter
        else if action = (reduce A::= α) then
            DO-REDUCTIONS(p, A::= α)
        else if action = accept then
            accepting-parser := p

DO-REDUCTIONS(p, A::= α) :
    forall p' for which a path of length(α) from p to p' exists do
        kids := the tree nodes of the links which form the path from p to p'
        REDUCER(p', GOTO(state(p'), A), A::= α, kids)

REDUCER(p-, state, A::= α, kids) :
    rulenode := GET-RULENODE(A::= α, kids)
    if ∃p E active-parsers with state(p) = state then
        if there already exists a direct link link from p to p- then
            ADD-RULENODE(treenode(link), rulenode)
        else
            n := GET-SYMBOLNODE(A, rulenode)
            add a link link from p to p- with tree node n
            forall p' in (active-parsers - for-actor) do
                forall (reduce rule) E ACTION(state(p'), current-token) do
                    DO-LIMITED-REDUCTIONS(p', rule, link)
    else
        create a stack node p with state state
        n := GET-SYMBOLNODE(A, rulenode)
        add a link from p to p' with tree node n
        add p to active-parsers
        add p to for-actor

DO-LIMITED-REDUCTIONS(p, A::= α, link) :
    forall p' for which a path of length(α) from p to p' through link exists do
        kids := the tree nodes of the links which form the path from p to p'
        REDUCER(p', GOTO(state(p'), A), A::= α, kids)

SHIFTER :
    active-parsers := 0
    create a term node n with token current-token
    forall <p-, state'> E for-shifter do
        if ∃p E active-parsers with state(p) = state' then
            add a link from p to p' with tree node n
        else
            create a stack node p with state state'
            add a link from p to p- with tree node n
            add p to active-parsers

GET-RULENODE(r, kids) :
    return a rule node with rule r and elements kids

ADD-RULENODE(symbolnode, rulenode) :
    add rulenode to the possibilities of symbolnode

GET-SYMBOLNODE(s, rulenode) :
    return a symbol node with symbol s and possibilities { rulenode }
```

Figure 68: Rekers' algorithm

It is not necessary for the reader to understand Rekers' algorithm in great detail, but we note two features that will provide a useful basis for comparison in the following discussion. Most of the algorithm is a relatively straightforward adaptation of the deterministic (simple stack) approach, but there are a couple of tricky issues that arise be-

cause of nondeterminism, and were missed by Tomita and later corrected by Nozohoor-Farshi [80].

The first of these is that a grammar containing a cycle of ε-reductions will cause a parser to loop infinitely—unless it can detect the loop. For programming language parsing this is an esoteric concern, as any such grammar will define infinitely many parse trees for a single sentence and we dismiss it as not well-formed for programming language definition. Nevertheless, Rekers' algorithm correctly handles such grammars by constructing cyclic parse trees. These are necessarily ambiguous, and as soon as the ambiguity is detected the loop is broken. We revisit this issue with our own approach, because even though cyclic ε-reductions will not occur when parsing source code, the ability to handle cyclic parse trees in finite time proves very valuable for our parser construction approach.

The second issue is of consequence here because we later propose an alternative solution to the one found by Nozohoor-Farshi and implemented by Rekers. GLR parsers prevent a combinatorial explosion by merging stacks that converge on the same state. Rekers' (and other) existing GLR parsers merge goto states (states reached on a goto) in the same way they merge shift states. Shift states, however, imply the end of the line for an input step and are not processed further during that step, whereas goto states must perform the actions they contain, until shift states are eventually reached. When parsing an ambiguous sentence a PDA may repeatedly goto a particular state while performing a `step()`. This presents a problem if there is a single (merged) top node for this goto state. If the goto state's actions have already been performed, then any reductions it contains will have been done for incoming links that existed at the time the reductions were performed. The same actions must be repeated, but only for newly added links. Rekers defines the `DO-LIMITED-REDUCTIONS` procedure for this purpose. It performs a search up to a fixed depth (the size of a reduction) through all paths at the top of the GSS to determine which ones traverse the new link. As we later explain, our alternative avoids this search by merging top nodes for goto states in a more restricted way.

4.3.1.5 *Graph-driven parsers*

This ends our discussion of table-driven parser implementations and we attend now to the graph-driven alternative, shown in Figure 69. The two graph-driven parser variants are `DeterminsiticParser` and `NondeterministicParser`. These classes are lighter weight than their table-driven equivalents (especially in the nondeterministic variety) because they use objects to represent states (rather than `int`s) and so can delegate more functionality. Their `step()` methods appear in Figure 70 and Figure 71.

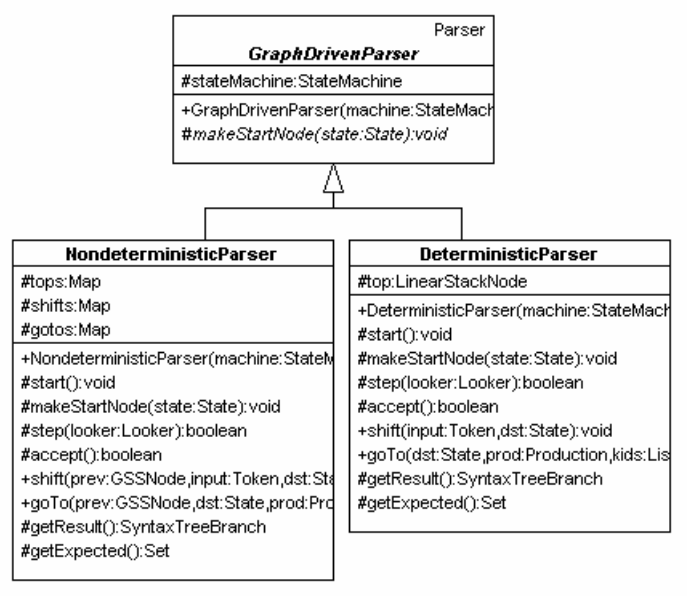

Figure 69: Graph-driven parsers

As can be seen in the methods, the deterministic parser has a single `top` node, while the nondeterministic parser has a collection of tops. The `act()` method of each top node is invoked. The node responds (after consulting the `State` it holds) by calling the parser back with a `shift()` or `goto()` request, or in the nondeterministic case, possibly more than one request. Reduce actions are handled by the stack nodes themselves, by popping their linear or graph-structured predecessors until a goto occurs.

Figure 72 shows how a `DeterministicParser` performs `shift()` and `goTo()` actions. For both actions, `top` is adjusted to reference a newly-pushed stack node. In the case of a goto action, a parse tree node is also constructed, and the new top node is asked to `act()`, perpetuating the process until a `shift()`

```
protected boolean step(Looker looker) {
    LinearStackNode oldTop = top;

    top.act(looker);

    while(top != null &&
          !top.getState().getSymbol().isTerminal())
    {
        top.act(looker);
    }

    if (top == null) {
        top = oldTop;
        return false;
    }

    return true;
}
```

Figure 70: The `step()` method of `DeterministicParser`

```
protected boolean step(Looker looker) {

    shifts.clear();
    gotos.clear();

    Iterator nodeIter = tops.values().iterator();
    while (nodeIter.hasNext()) {
        GSSNode node = (GSSNode) nodeIter.next();
        node.act(looker);
    }

    if (!shifts.isEmpty()) {
        tops.clear();
        tops.putAll(shifts);
        return true;
    }

    return false;
}
```

Figure 71: The `step()` method of `NondeterministicParser`

eventually occurs.

```
public void shift(Token input, State dst) {
    top = top.makeShiftNode(dst, input);
}

public void goTo(State dst, Production prod,
                 List kids, Looker input)
{
    SyntaxTreeBranch tree = treeBuilder.makeBranch(prod, kids);

    top = top.makeGotoNode(dst, tree);
    top.act(input);
}
```

Figure 72: The `shift()` and `goTo()` methods of `DeterministicParser`

Figure 73 contains the nondeterministic versions of `shift()` and `goto()`. These methods implement the merging behaviour of the graph-structured stack. The algorithm used here is new and improves on Nozohoor-Farshi's method implemented by Rekers, as we explain.

`NondeterministicParser` stores (in the `shifts` `Map`) all nodes pushed onto the top of the stack by a `shift()` action during the current step. When a `shift()` occurs, the parser first checks whether a shift to that state has already happened, and if so the same top node is re-used, merging two branches of the stack. Goto nodes, however, are never merged and this is where the algorithm's behaviour is new. Rekers' approach does merge goto nodes, and so creates a situation in which some links entering a goto node have been reduced and some have not. Rekers' `DO-LIMITED-REDUCTIONS` procedure is then necessary to perform a brute-force search through all possible reductions to find those that pop through the new link. Our approach avoids this search by simply not merging goto nodes; reductions then go only where they should, and when a shift subsequently occurs merging takes place as normal.

Our approach re-uses goto nodes only when they are reached again from the same predecessor node (that is, following the same link) during the same step. This occurs only when an ambiguity has been found, and the ambiguous alternative trees can be merged. In this situation the goto node is not asked to `act()` because it has already done so.

```
public void shift(GSSNode prev, Token input, State dst) {
    GSSNode shift = (GSSNode) shifts.get(dst);

    if (shift != null)
        shift.addPrevious(prev, input);
    else {
        shift = prev.makeShiftNode(dst, input);
        shifts.put(dst, shift);
    }
}

public void goTo(GSSNode prev, State dst,
                 Production prod, List kids, Looker input) {

    GSSNode goTo = (GSSNode) gotos.get(new Pair(prev, dst));

    if (goTo == null)
        goTo = prev.checkCycle(dst);

    SyntaxTreeBranch tree = treeBuilder.makeBranch(prod, kids);

    if (goTo != null) {
        SyntaxTree oldTree = goTo.getTree(prev);
        treeBuilder.makeAmbiguity(oldTree.asBranch(), tree);
    }
    else {
        goTo = prev.makeGotoNode(dst, tree);
        gotos.put(new Pair(prev, dst), goTo);
        goTo.act(input);
    }
}
```

Figure 73: The `shift()` and `goTo()` methods of `NondeterministicParser`

Tomita's original GLR algorithm looped infinitely on some grammars containing ε–productions. This occurred when a state could reduce an empty production and the re-

sulting goto cycled back to the same state, either directly or via a series of similar empty reductions. The change we have made re-introduces that problem, but it is easily overcome by a small modification. The `checkCycle()` method visible in Figure 73 checks to see if a sequence of nodes at the top of the stack contains a cycle of empty parse trees. If so, the lowest-positioned node involved in the cycle is re-used, creating a loop in the graph-structured stack in much the same way that Nozohoor-Farshi's algorithm does. The `checkCycle()` method is more efficient, however, because it need only follow a linear-structured segment at the top of the stack until a non-empty parse tree or a shift node is encountered.

4.3.1.6 Stack nodes

We now describe the remaining classes in the `pda` package. These are the graph-driven parser stack nodes shown in Figure 74. These classes have richer functionality than their table-driven equivalents, because they are responsible for interacting with `State` objects (which are defined in the `stateMachine` package,

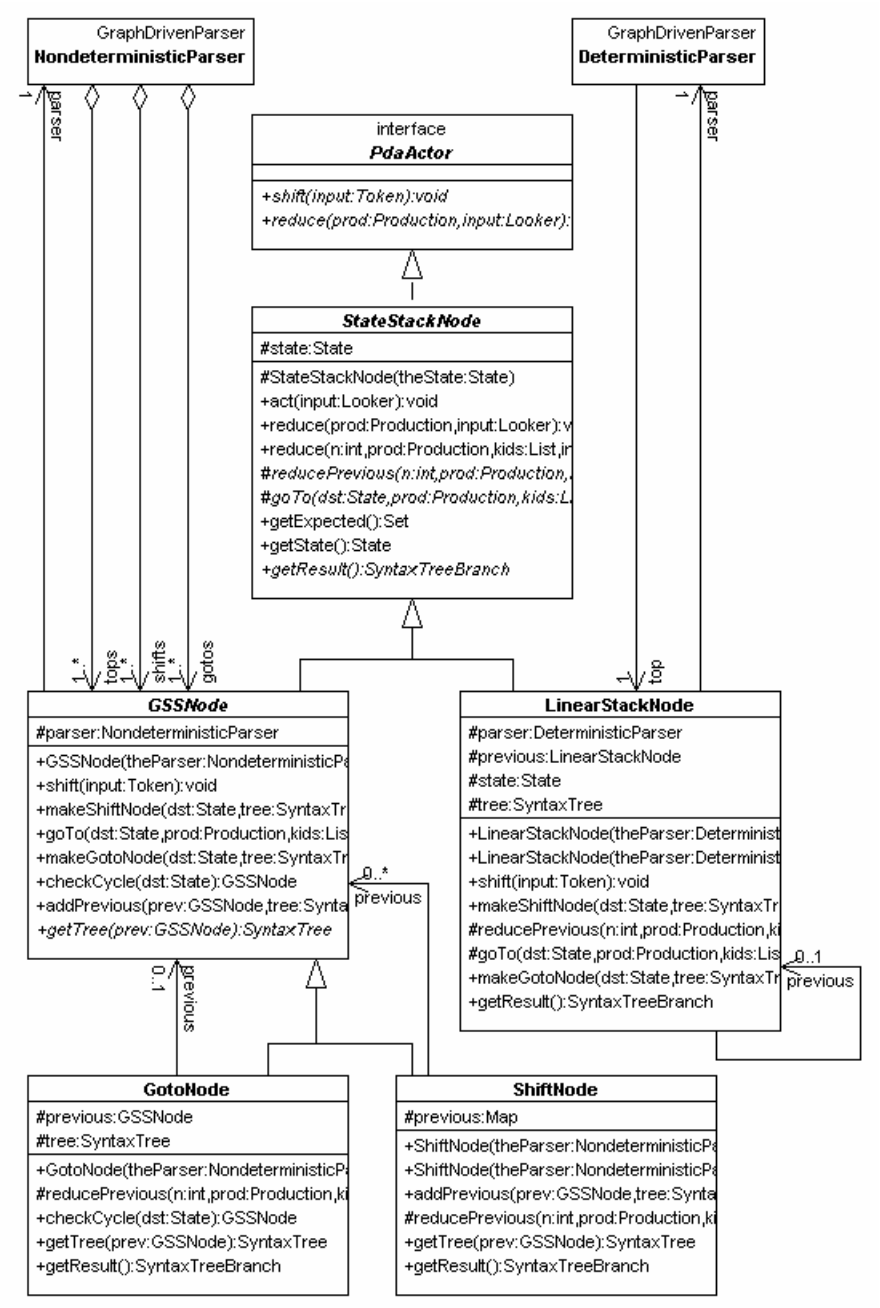

Figure 74: Graph-driven parser stacks

described below). They are, nevertheless, simple classes. Each `StateStackNode` references a `State`. The `act()` method of `StateStackNode` simply forwards the `act()` request to the `State`, which calls back the node using the `shift()` and/or `reduce()` methods defined in the `PdaActor` interface. The `PdaActor` interface is all that is presented to `State` objects, so that node implementations remain hidden from `States`. The state is ultimately responsible for deciding which action(s) it should take.

This completes the description of the contents of the `pda` package, other than classes contained in sub-packages. We now briefly describe the sub-packages: `tokenStream`, `stateMachine` and `builder`.

4.3.1.7 Parser input streams

The `tokenStream` package, shown in Figure 75, provides classes that handle input to the parser. These are largely self-explanatory, with the possible exception of the `Looker` interface, which abstracts the protocol for looking ahead into a token stream.

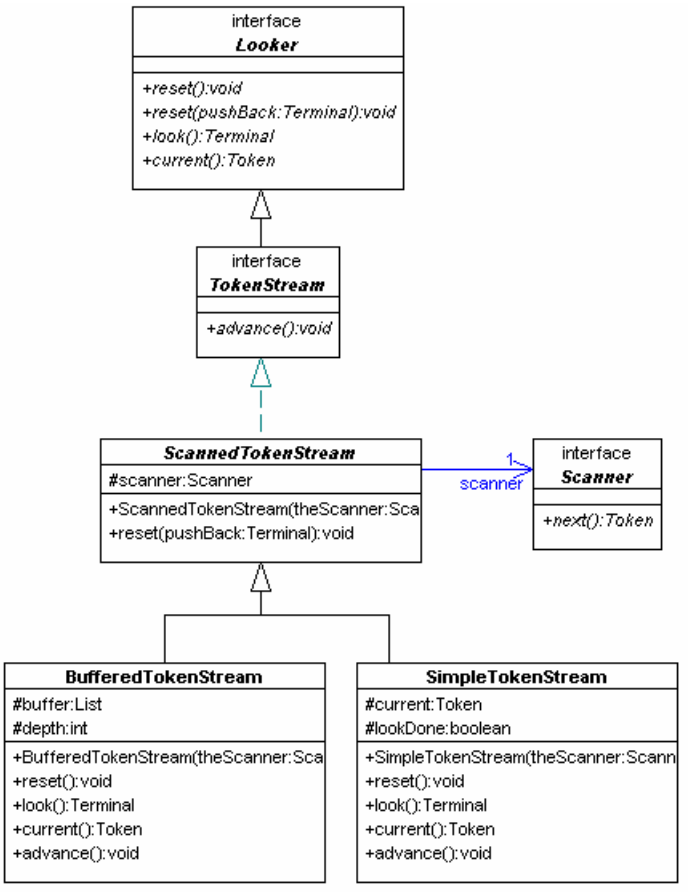

Figure 75: Classes for Token input

4.3.1.8 State machine design

States are defined in the `stateMachine` package, as in Figure 76. (For clarity, we have omitted table-driven state machine classes from the diagram.) `State` itself is an abstract base class, while `RunState` provides a default state implementation for the runtime library.

Every `State` except the start state records the symbol of its immediate incoming transitions. (Although state transition diagrams

are normally drawn with a symbol labelling each transition, all transitions into a state must have the same symbol; this design avoids redundantly storing symbols on links.) `States` store transitions as a `Map` of destination `States` indexed by symbol. Reductions (`Productions`) are stored in a `Set`. States delegate the task of choosing parsing actions to `Lookaheads`.

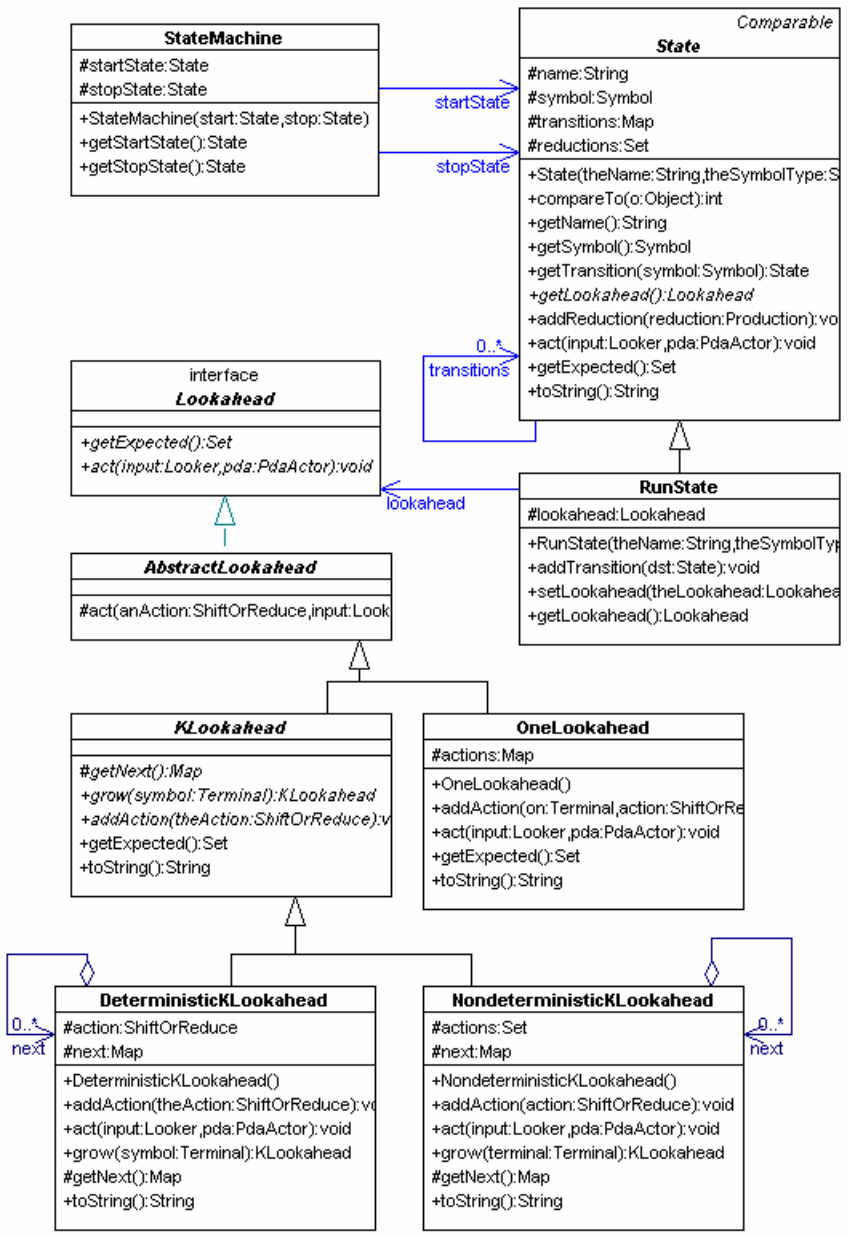

Figure 76: State machine classes

A `Lookahead` examines a sequence of input `Tokens` and initiates one or more parser actions by calling back the given `PdaActor`, as described above. `KLookahead` and its subclasses support heterogeneous values of *k* by storing lookahead sequences as trees of various depths.

4.3.1.9 State machine builder

The final piece of the runtime PDA library is a *builder* design pattern [35], which aids the construction of state machines. It is shown in Figure 77. `StateMachineBuilder` provides

an API for assembling automata, and delegates lookahead construction to a `LookaheadBuilder`.

`GenStateMachineBuilder` is a specialist interface designed to simplify code generation. It accepts arrays of `Strings` as inputs, since these can easily be generated.

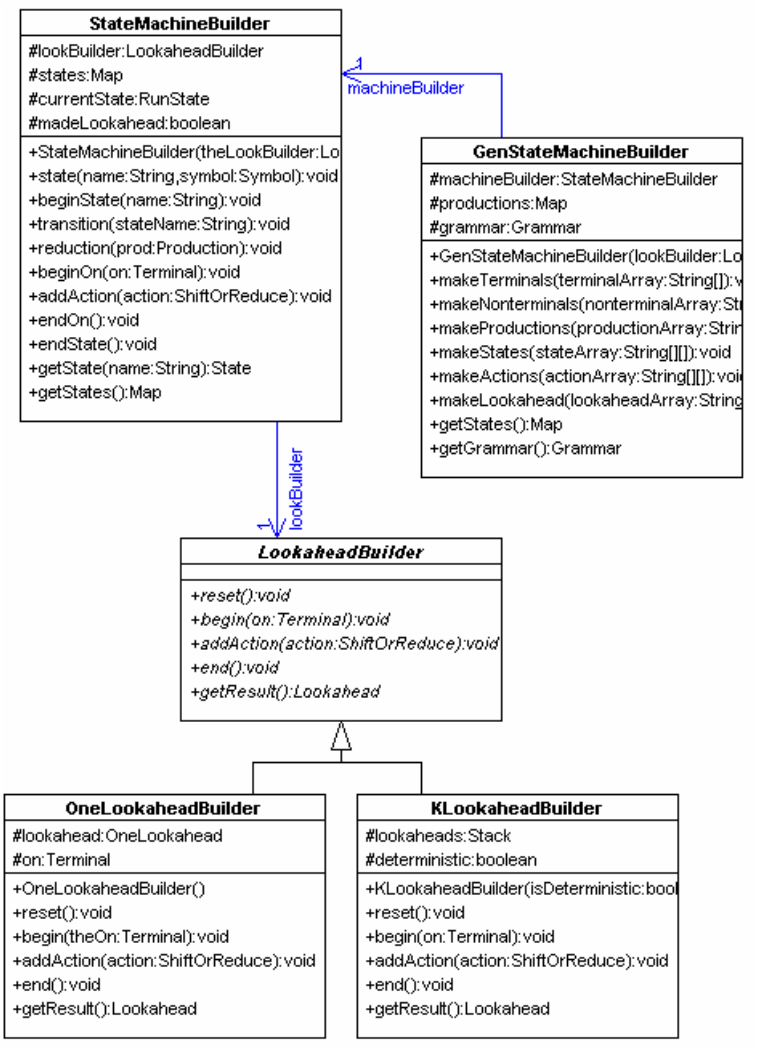

Figure 77: State machine builder

4.3.2 PDA construction

The previous section describes the *runtime* PDA library, which implements deterministic and nondeterministic parsers within a common framework. We now explain how the runtime classes are extended in order to implement a parser *generator*.

4.3.2.1 Main concepts

The central innovation in our approach is the use of an enhanced GLR automaton to explore its own state space, in order to calculate lookaheads and to split states. The automaton is consequently self-modifying. When the modification process is complete, the automaton is ready to be output as a runtime PDA.

The parser generator is implemented by specialising the runtime framework. The runtime `NondeterministicParser` is subclassed to produce a parser capable of parsing *sub-sentences* of length *k*, and then extended further to 'parse' all possible *k* sub-sentences for use as lookaheads. As it does so, it can split states to remove lookahead conflicts.

This design re-uses runtime classes for the parser generator, and consequently significantly reduces the size and complexity of the parser generator. To our knowledge, the use of a GLR-based automaton to implement a parser generator is an original contribution.

The parser generator package structure appears in Figure 78. It parallels the runtime package structure, but omits some utility packages.

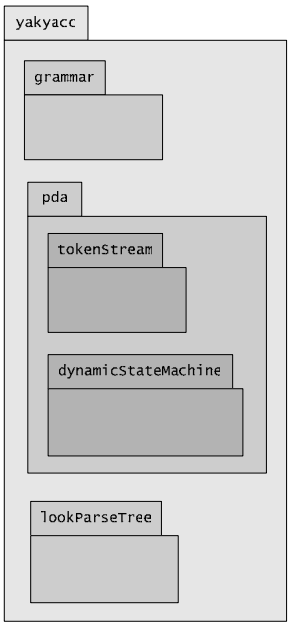

Figure 78: Parser generator package structure

4.3.2.2 Grammars

The runtime library contains classes for representing grammars, as we have seen. It does not, however, need to perform checks on grammars to ensure they are well formed (that is, that all nonterminals are referenced and resolvable, etc) because grammars for runnable parsers must already have been checked. The `yakyacc.grammar` package, shown in Figure 79, provides the `YakGrammar` class to implement grammar validity checks in the parser generator and to augment the grammar with a top-level production. The package also contains a class that can load a grammar from an XML file; this file is the sole input to the parser generator.

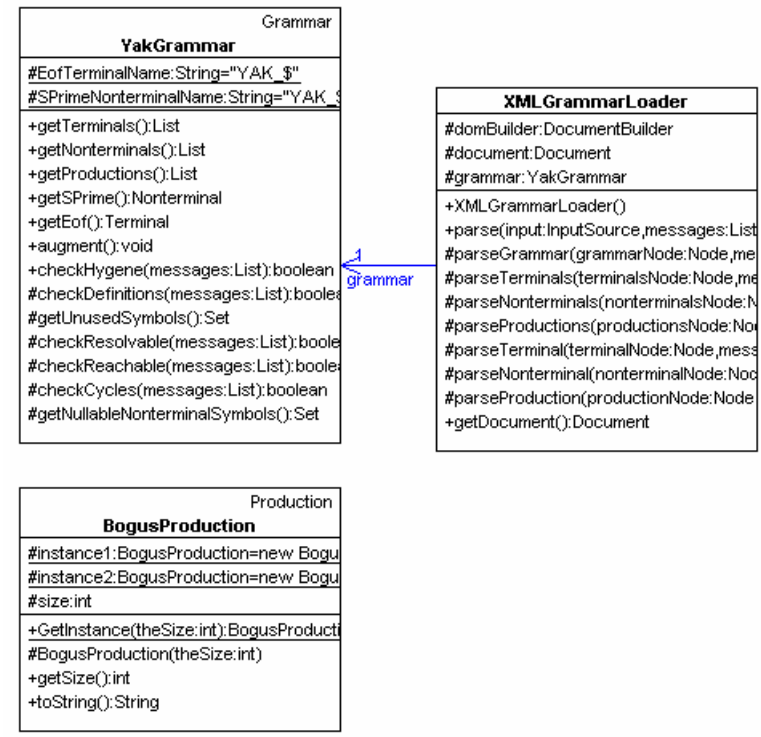

Figure 79: Yakyacc grammar classes

Finally, the `grammar` package contains an artificial `Production` subclass, `BogusProduction`, that is

used during parser generation, as we explain below.

4.3.2.3 Sub-sentence parsers

The `yakyacc.pda` package contains the most important classes of the parser generator. We now discuss two classes of that package that provide a basis for the subsequent design discussion.

Figure 80 shows classes that extend the runtime nondeterministic parser to allow parsing of sub-sentences of length *k*. As can be seen from the diagram, the extended functionality is achieved relatively simply.

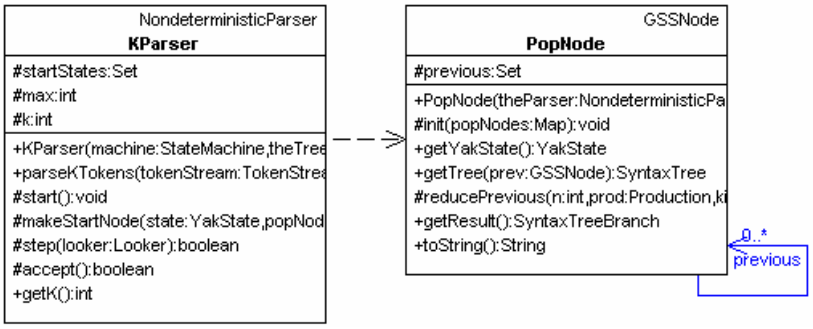

Figure 80: `KParser`

`KParser` inherits the contract of `NondeterministicParser` and behaves like a normal GLR parser when used normally. However, it extends normal parser functionality by allowing parsing to begin at some point other than the start of the sentence. This requires the parser to begin parsing in any state (or states) where the sub-sentence begins. By default, a `NondeterministicParser` primes its stack with the single start state defined by its `StateMachine`. A `KParser`, in contrast, accepts a set of start states when it is constructed and primes its stack with them. If a `KParser` is invoked via the `parseKTokens()` method (rather than the usual `parse()` method), it stops after *k* steps (by overriding the `accept()` method).

Because sub-sentence parsing does not necessarily begin with a state machine's real start state, the automaton may pop *below* known states on the stack. In a normal parser this would be a fatal error. `KParser` accommodates this need by priming the stack with `PopNodes`. When a `PopNode` is constructed, it generates an artificial stack history for itself, consisting of other `PopNodes`. This history contains all possible routes by which the original `PopNode` *might* have been reached. The relevant code appears in Figure 81.

```
public PopNode(NondeterministicParser theParser,
               YakState theState, Map popNodes)
{
    super(theParser, theState);

    previous = new HashSet();

    popNodes.put(theState, this);
    init(popNodes);
}

protected void init(Map popNodes) {

    Set prevStates = getYakState().getIn();

    Iterator prevIter = prevStates.iterator();
    while (prevIter.hasNext()) {
        YakState prevState = (YakState) prevIter.next();

        PopNode prevPopNode = (PopNode) popNodes.get(prevState);
        if (prevPopNode == null)
            prevPopNode = new PopNode(parser, prevState, popNodes);

        previous.add(prevPopNode);
    }
}
```

Figure 81: `PopNode` construction

Only one `PopNode` is constructed for each state reachable by popping below the original state. Whenever a particular state can be reached by popping through different paths, the links in the stack converge on the single `PopNode` for that state. Consequently, the bottom section of the graph-structured stack contains *cycles* of `PopNodes`. The upper section of the stack contains pushed `GSSNodes` as usual. When the parser runs, it (nondeterministically) follows all possible paths, and therefore produces all possible parse trees for the sub-sentence.

This technique causes no performance difficulty because the number of `PopStates` is finite, and a particular goto action will be performed only once for each state; the inherited `goto()` behaviour of the parser catches identical goto actions and records ambiguous parse tree nodes. These ambiguous tree nodes may be cyclic.

4.3.2.4 Transmogrifiers

We now describe the final extensions to the `Parser` hierarchy. Figure 82 depicts the `Transmogrifier` classes; the name evokes their ability to change the structure of a PDA as they explore it.

`Transmogrifier` conforms to the contract of `KParser`, and consequently will work as expected for normal nondeterministic parsing and for sub-sentence parsing. If invoked via the `mogrifyParse()` method, however, it behaves as a parser generator by modifying the automaton as it runs. In this case it does not

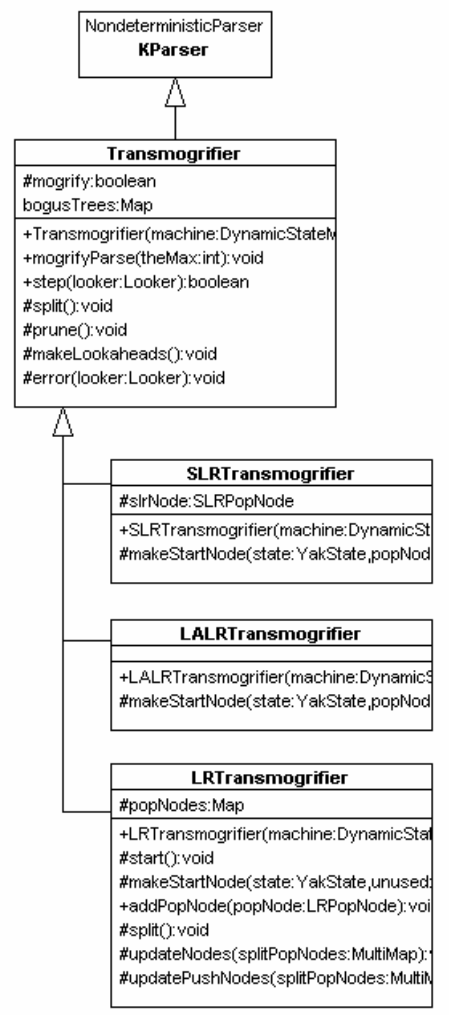

Figure 82: `Transmogrifier` classes

parse a real input stream of tokens, but instead produces all possible parses up to a given maximum value of *k*. In this way it discovers the lookahead sequences for all states from which it starts. Initial states are provided to the constructor, as for `KParser`.

Figure 83 shows how the major methods of `Transmogrifier` are implemented. A sub-sentence parse is initiated with a `BogusTokenStream`, which provides all possible tokens to states request input in order to choose parse actions. This is how all possible token sequences are generated.

```
public void mogrifyParse(int theMax) {
    mogrify = true;

    parseKTokens(new BogusTokenStream(), theMax);

    mogrify = false;
}

public boolean step(Looker looker) {
    super.step(looker);

    if (mogrify) {
        makeLookaheads();
        split();
        prune();
    }

    return !tops.isEmpty();
}
```

Figure 83: Major `Transmogrifier` methods

The `step()` method of `Transmogrifier` works normally (conforms to the inherited contract), unless the `mogrify` flag indicates that a special `mogrifyParse()` is in progress. In that case, it uses token sequences discovered during the step to construct lookaheads for all states from which the parse began. It then calls `split()`, which does nothing by default but is overridden in a subclass. Finally, it calls `prune()`, which removes from the stack any nodes that are involved in producing only adequate lookaheads. Consequently, the only lookaheads that will be explored further (on the next step) are inadequate ones. The process stops when all lookaheads are adequate or the *k* limit is reached.

`Transmogrifier` has three simple subclasses, `SLRTransmogrifier`, `LALRTransmogrifier` and `LRTransmogrifier`. Although little code is required, these three classes modify the parser generator's behaviour so that it produces SLR(*k*), LALR(*k*) or LR(*k*) automata respectively. This is achieved by constructing a different subclass of `PopNode` in each case (shown in Figure 84), and also overriding `split()` in the LR(*k*) case.

To make a `Transmogrifier` that calculates SLR lookahead, `PopNodes` are set up in a differ-

```
protected void makeStartNode(YakState state, Map popNodes) {
    TransmogrifierPopNode top =
        new TransmogrifierPopNode(this, state);

    top.addPrevious(slrNode);

    tops.put(state, top );
}
```

(a) `SLR PopNode`

```
protected void makeStartNode(YakState state, Map popNodes) {
    LALRPopNode top = (LALRPopNode) popNodes.get(state);
    if (top == null)
        top = new LALRPopNode(this, state, popNodes);

    tops.put(state, top);
}
```

(b) `LALR PopNode`

```
protected void makeStartNode(YakState state, Map unused) {
    LRPopNode top = (LRPopNode) popNodes.get(state);
    if (top == null)
        top = new LRPopNode(this, state, popNodes);

    tops.put(state, top);
}
```

(c) `LR PopNode`

Figure 84: Priming `Transmogrifiers` with different `PopNodes`

ent fashion from the cyclic graph structure used by `KParser`. The `KParser` cyclic graph follows actual transitions in the state machine, and consequently produces only those token sequences that are actually possible. SLR lookahead, on the other hand, includes some extra lookaheads. The SLR lookahead calculation does not accurately trace pops and gotos, but instead 'jumps' on a goto to all states that contain the same symbol as the current goto action. We induce this behaviour in `SLRTransmogrifier` by placing at the base of the stack a special `SLRPopNode`, which acts as though it contains a state with transitions to every goto state. This is, to our knowledge, a new way of calculating SLR lookahead. Its advantage in our context is that it integrates SLR parser generation with its alternatives, and very little code is required. For comparison, conventional SLR lookahead calculation is described in [23].

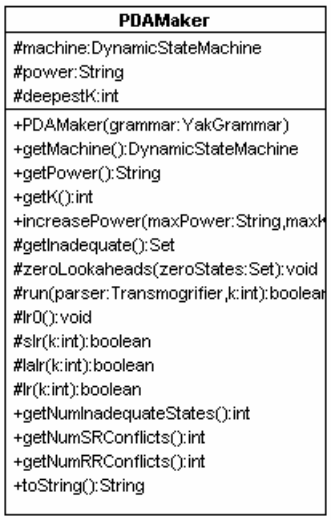

Figure 85: `PDAMaker`

LALR and LR `Transmogrifiers` use cyclic `PopNode` structures like that of `KParser`. Different `PopNode` subclasses are constructed for each case, however, because `LRPopNodes` also support splitting of states.

We are now ready to describe how a PDA is constructed, and will return to the details of splitting states and pruning stack nodes below. `PDAMaker`, shown in Figure 85, controls the process. It works much like the extended example presented in Chapter 3, by progressively escalating parsing power for inadequate states.

PDA building begins by constructing a `DynamicStateMachine` (discussed later) from the grammar. The state machine constructor creates LL automata, adds ε–transitions to make an LR automaton, and merges ambiguous transitions. `PDABuilder` then makes the machine LR(0), simply by constructing 0-length lookaheads. Different types of `Transmogrifier` are then invoked in order of power (SLR, LALR, LR) to escalate the power of the PDA, as shown in Figure 86. Each `Transmogrifier` is provided with the remaining set of inadequate states as its start states. Lookahead depth is limited to 1 for all but the last `Transmogri-`

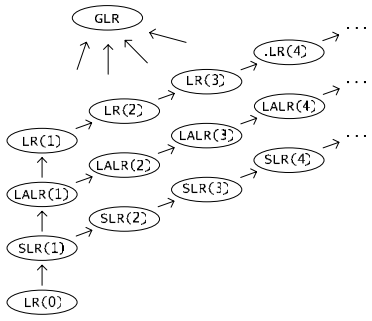

Figure 86: Escalating parser construction

fier invocation; command line arguments specify which parser class should be last and the maximum value of *k*. If any inadequate states remain at the end of this process, a GLR parser will be required.

We now discuss LR state splitting. LRTransmogrifier differs from the other Transmogrifiers by splitting states, if possible, to improve their lookahead. The LRTransmogrifier split() method appears in Figure 87. It attempts to split every LRPopNode; these contain the states that lead by some series of transitions to inadequate state(s). (This must be so, as PopNodes are obtained only by popping back from inadequate states.) If any PopNode is successfully split, the split() method re-visits PopNodes immediately above the split on the stack to check whether the next PopNodes can now be (further) split as a result of the change.

```
protected void split() {
    MultiMap splitPopNodes = new MultiMap();

    Set toDo = new HashSet(popNodes.values());

    while (!toDo.isEmpty()) {
        Iterator first = toDo.iterator();
        LRPopNode popNode = (LRPopNode) first.next();
        first.remove();

        Set next = popNode.getYakState().getOut();

        Set partitions = popNode.split();

        if (partitions.size() > 1) {
            splitPopNodes.put(popNode, partitions);

            Iterator outIter = next.iterator();
            while (outIter.hasNext()) {
                YakState dst = (YakState) outIter.next();

                LRPopNode more = (LRPopNode) popNodes.get(dst);
                if (more != null)
                    toDo.add(more);
            }
        }
    }

    updateNodes(splitPopNodes);
}
```

Figure 87: The split() method of LRTransmogrifier

Some of the lookahead of a state comes from transitions that (only) walk forward from that state. This lookahead will reside in parse trees attached to TransmogrifierPushNodes on top of a starting PopNode. Splitting a state can never eliminate lookahead derived in this way, because any new state must be able to accept the same token sequences as the old one. We describe this inalienable lookahead as *anchored*.

Some lookahead, however, comes from popping back from a state (at some point during the parse) and then pushing forward again. If we pop over a state that has multiple incoming links, then all of them will be followed and the lookaheads subsequently generated by the different paths might differ. We can view these lookaheads as flowing into a state (and its successors) along incoming transitions. When transitions converge on a state, the burden of lookahead carried by each transition is combined and may cause conflicts in that state or subsequent ones. If we split the state where the convergence occurs, we can keep the inflowing lookaheads separate and may thus reduce the number of conflicting lookaheads.

The same situation is echoed by PopNodes, because they 'wrap' states. If two or more incoming links converge on a PopNode, then that PopNode may be split into separate PopNodes reached by different incoming links. The state within the original PopNode must also have convergent incoming transitions and be split like the PopNode.

Our LRPopNode splitting algorithm relies on knowing what lookaheads are carried by links between LRPopNodes. It also knows all anchored lookaheads. An LRPopNode can use this information to calculate and compare the sets of lookaheads that would arise if the LRPopNode (and its state) were split. Details of how an LRPopNode splits are provided in the next section.

Before we move on to the design of Transmogrifier stack nodes, we briefly discuss the final task of a Transmogrifier step(): pruning stack links that are no longer necessary because they produce only adequate lookaheads. The code appears in Figure 88. It delegates to the top stack nodes the task of removing unwanted links. (We look again at this when we discuss stack nodes.) If all paths to a top node prove to be adequate the top node is removed. The remainder of the prune() method removes all artificial parse tree branches created when making lookaheads.

```
protected void prune() {

    tops.remove(stateMachine.getStopState());

    Map cache = new HashMap();

    Iterator nodeIter = tops.entrySet().iterator();
    while (nodeIter.hasNext()) {
        Map.Entry entry = (Map.Entry) nodeIter.next();
        TransmogrifierGSSNode node = (TransmogrifierGSSNode) entry.getValue();

        if (node.prune(cache))
            nodeIter.remove();
    }

    Set bogusKids = new HashSet();
    Iterator bogusIter = bogusTrees.values().iterator();
    while (bogusIter.hasNext()) {
        LookParseTreeBranch bogus = (LookParseTreeBranch) bogusIter.next();

        bogusKids.addAll(bogus.clean());
    }

    bogusKids.removeAll(bogusTrees.values());
    bogusTrees.clear();

    bogusIter = bogusKids.iterator();
    while (bogusIter.hasNext()) {
        LookParseTreeBranch bogus = (LookParseTreeBranch) bogusIter.next();

        bogus.clean();
    }
}
```

Figure 88: The prune() method of Transmogrifier

4.3.2.5 Stack nodes

Figure 89 shows the graph structured stack nodes used by Transmogrifiers. We have already explained the purpose of the PopNodes. The remaining classes exist to support lookahead calculation, state splitting and pruning, as these tasks are delegated by the Transmogrifier to TransmogrifierGSSNodes.

Parse trees contain the sequence of tokens encountered during a parse; the tokens are the leaves of the tree. This means that at the end of the k^{th} `step()` of a `mogrifyParse()`, parse trees contain all *k*-lookahead sequences. Lookahead calculation consists of extracting this information.

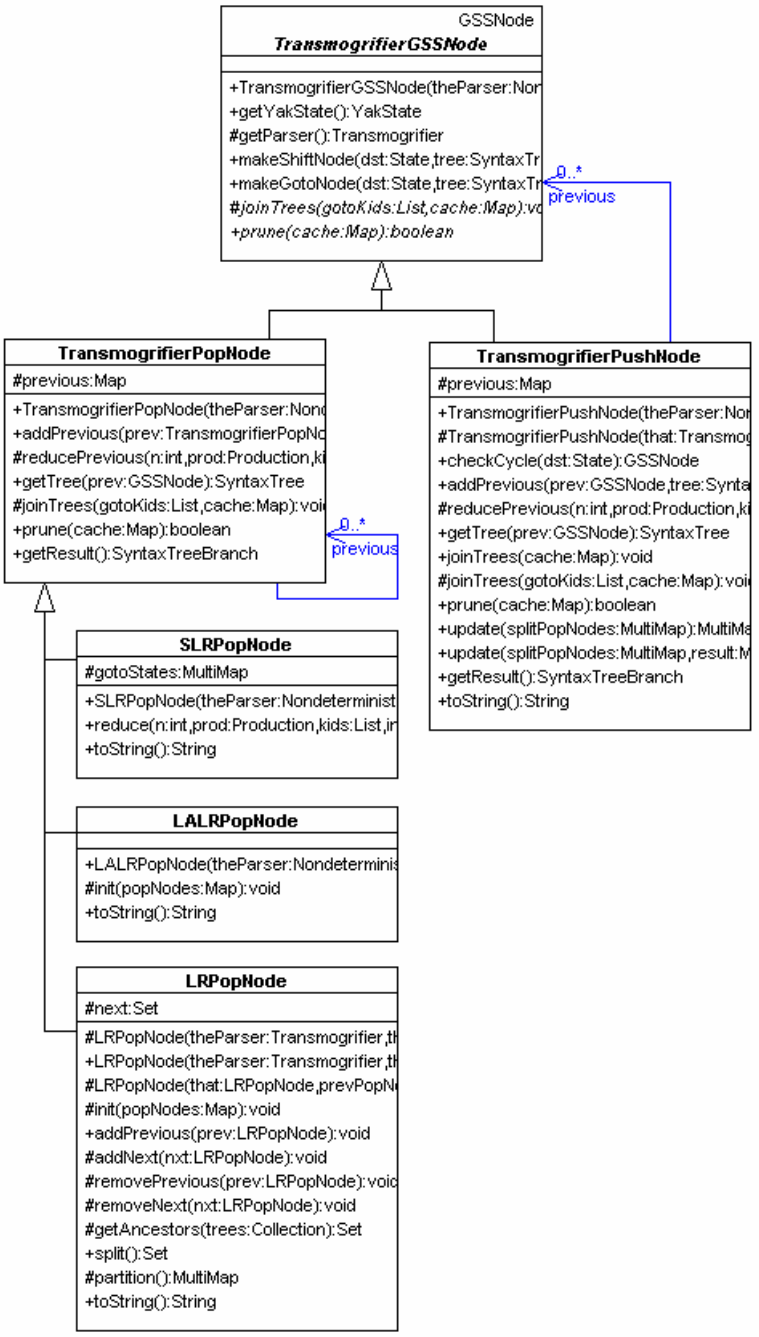

Figure 89: `TransmogrifierGSSNode` hierarchy

The first step of lookahead calculation joins adjacent parse trees together to ensure a full sequence of *k* tokens. *Adjacent* here means the trees are associated with sequential links in the graph structured stack. The `joinTrees()` method of `TransmogrifierPushNode` achieves this task by performing an artificial reduction using the `BogusProduction` introduced earlier. This process creates artificial branches above existing parse trees that concatenate adjacent trees, until a `PopNode` is reached, at which point the joined tree must contain sequences of *k* tokens. This is necessarily so, because popping the stack below the origin state can never add tokens (and we never push a `PopNode`). The `joinTrees()` method uses a cache of intermediate results to avoid repeating work when graph-structured stack nodes are revisited via different paths.

```
public Set split() {
    Set result = new HashSet();

    if (previous.size() < 2)
        return result;

    // LookaheadSet ==> LRPopNodes
    MultiMap partitions = partition();

    if (partitions.asMap().size() < 2)
        return result; // Can't split.

    Iterator partIter = partitions.asMap().entrySet().iterator();
    while (partIter.hasNext()) {
        Map.Entry entry = (Map.Entry) partIter.next();
        LookaheadSet lookSet = (LookaheadSet) entry.getKey();
        Set nodes = (Set) entry.getValue();

        result.add(new LRPopNode(this, nodes));
    }

    // This popNode isn't reachable any more.
    Iterator nextIter = next.iterator();
    while (nextIter.hasNext()) {
        LRPopNode nxt = (LRPopNode) nextIter.next();

        nxt.removePrevious(this);
    }

    next.clear();

    while (!previous.isEmpty()) {
        LRPopNode prev = (LRPopNode)
                         previous.keySet().iterator().next();

        removePrevious(prev);
    }

    return result;
}
```

Figure 90: The `split()` method of `LRPopNode`

```
protected MultiMap partition() {
    MultiMap partitions = new MultiMap();

    Iterator prevIter = previous.entrySet().iterator();
    while (prevIter.hasNext()) {
        Map.Entry entry = (Map.Entry) prevIter.next();
        LRPopNode prev = (LRPopNode) entry.getKey();
        LookPoppedTree tree = (LookPoppedTree) entry.getValue();

        LookaheadSet prevLook = tree.getPartition();

        boolean combined = false;
        Iterator partIter = partitions.keySet().iterator();
        while (partIter.hasNext() && !combined) {
            LookaheadSet part = (LookaheadSet) partIter.next();

            if (part.combine(prevLook)) {
                partitions.put(part, prev);
                combined = true;
            }
        }

        if (!combined) {
            partitions.put(prevLook, prev);
        }
    }

    return partitions;
}
```

Figure 91: The `partition()` method of `LRPopNode`

Splitting of states is initiated by the `split()` method of `LRTransmogrifier`, as we have seen. The actual splitting of nodes takes place in `LRPopNode`. The code is shown in Figure 90 and Figure 91. No split is possible unless there is more than one incoming link, so this condition is checked first. The `split()` method then invokes `partition()`, which determines the set of lookaheads carried by the parse tree of each incoming link. (These sets include anchored lookaheads, so contain the full lookahead sets in which each link participates.) The sets are compared to determine which ones can be combined without introducing conflicts; so that we do not do any splitting that fails to improve the lookahead of some state.

If more than one partition is found, the `split()` method calls a special constructor of `LRPopNode` (Figure 92) to copy the current stack node and adjust the stack links. The constructor also invokes two different parse tree methods so that they will update their lookaheads in order to reflect the now-split states. The first of these calls, `replaceWith()`, informs a parse tree node that the stack link on which it resides has just been changed to point to a new `LRPopNode`, and that therefore lookaheads associated with that parse tree might now apply to a newly created state. The second call, `splice()`, informs a parse tree node that an entirely new stack link has been created by cloning an existing one. The `splice()` method clones a fragment of the parse tree associated with the original link and associates the new

fragment with the new link. The two fragments may merge (as ambiguous sub-trees) at some higher parse tree branches made by popping back to earlier `LRPopNodes`.

```
protected LRPopNode(LRPopNode that, Set prevPopNodes) {
    this(that.getParser(), new YakState(that.getYakState()));

    // Switch prev nodes to me.
    Map thatPrevious = that.previous;

    Iterator prevIter = prevPopNodes.iterator();
    while (prevIter.hasNext()) {
        LRPopNode prev = (LRPopNode) prevIter.next();

        YakState prevState = prev.getYakState();
        YakState oldDst = that.getYakState();
        assert prevState.getTransition(oldDst.getSymbol()) == oldDst;
        prevState.swapTransition(oldDst, getYakState());

        addPrevious(prev);
        LookPoppedTree thatTree = (LookPoppedTree) thatPrevious.get(prev);
        LookPoppedTree thisTree = (LookPoppedTree) previous.get(prev);
        assert thatTree != null;
        assert thisTree != null;

        thatTree.replaceWith(thisTree);
    }

    // Replicate next links
    Set newAncestors = getAncestors(previous.values());
    Set oldAncestors = getAncestors(that.previous.values());

    Set thatNext = that.next;
    Iterator nextIter = thatNext.iterator();
    while (nextIter.hasNext()) {
        LRPopNode nxt = (LRPopNode) nextIter.next();

        nxt.addPrevious(this);
        LookPoppedTree thatTree = (LookPoppedTree) nxt.previous.get(that);
        LookPoppedTree thisTree = (LookPoppedTree) nxt.previous.get(this);
        assert thatTree != null;
        assert thisTree != null;

        thatTree.splice(thisTree, newAncestors, oldAncestors);
    }
}
```

Figure 92: Splitting constructor of `LRPopNode`

4.3.2.6 Lookahead parse trees

`Transmogrifiers` use specialised parse trees that translate tree nodes into lookahead sequences. As usual, a `ParseTreeFactory` hides the implementation from higher-level classes. The `lookParseTree` package, shown in Figure 93, contains the relevant classes. These trees consist of `LookTokens` and `LookParseTreeBranches`, which represent parse tree leaves and branches, respectively. A new type of parse tree node, `LookPoppedTree`, is also introduced. These are leaf nodes that represent missing sub-trees that are uncovered by popping below the origin of the stack. They are all created before a parse begins: when the `PopNodes` for any `Transmogrifier` are initialised at the start of a parse, the links between them are associated with a `LookPoppedTree`. We do this so that every link in a `Transmogrifier` stack has a related parse tree node and can determine what lookahead it generates. `LookPoppedTrees` always generate a lookahead of length zero. They are nevertheless useful, because they locate a fragment of lookahead in the parse tree and consequently assist with determining to which state lookaheads higher up in the tree apply. They also perform an important role in moving lookaheads between states when states are split, via the `replaceWith()` and `splice()` methods described above.

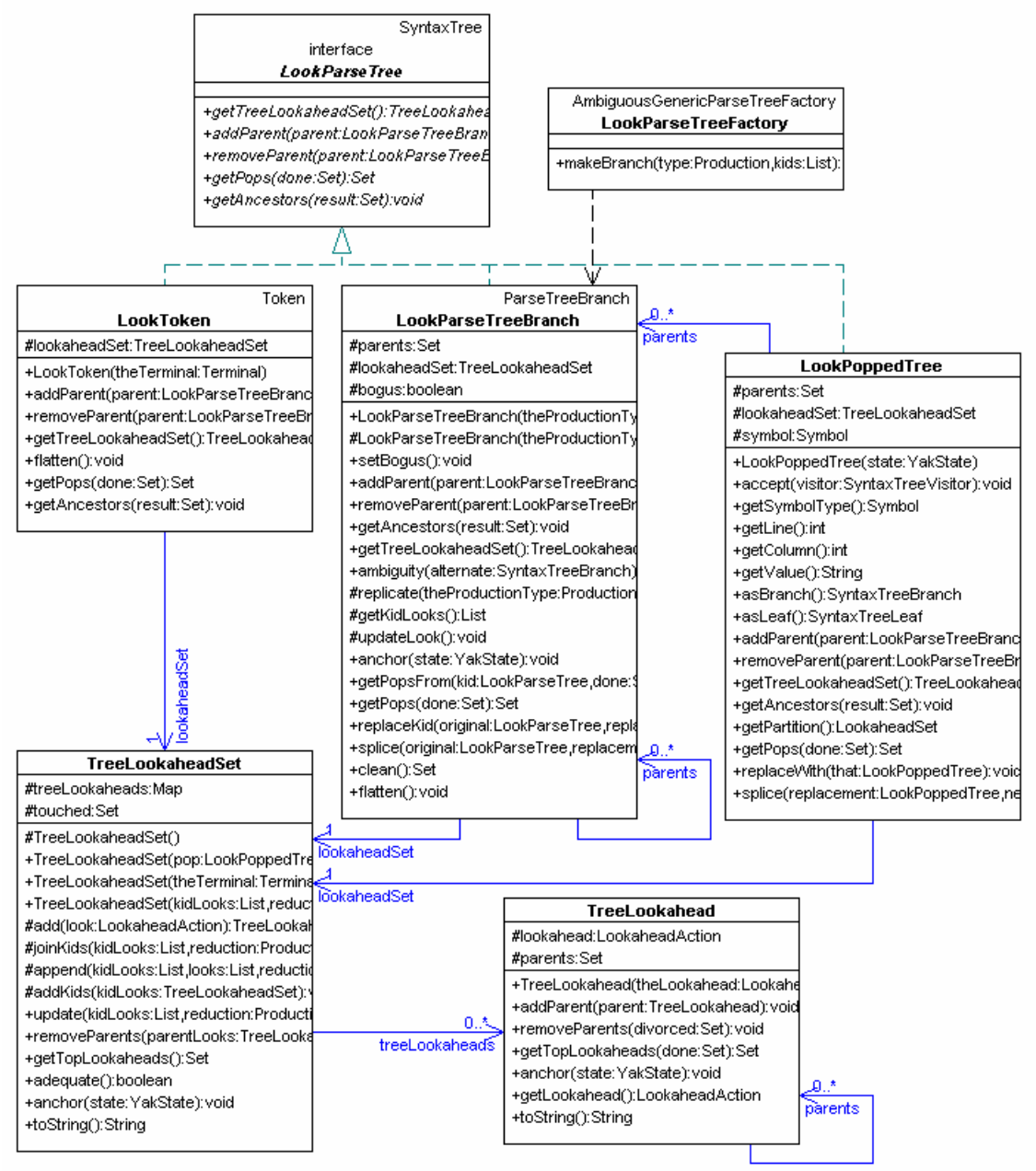

Figure 93: Parse trees that calculate lookahead

`LookParseTree` nodes delegate lookahead assembly to the `TreeLookSet` class so that common functionality appears in one place. Every parse tree node stores one `TreeLookSet` object, which holds a collection of the lookaheads generated by that tree node. It also ensures that each fragment of lookahead has a `parent` relationship to longer lookaheads to which it contributes. (This relationship is used by the `getPartition()` method of `LookPoppedTree` to find all lookaheads in which the `LookPoppedTree` participates.)

Lookahead construction occurs as parse trees are built; each tree node always knows its current lookahead sequences. When an ambiguity is found, new lookaheads may be introduced to an existing parse tree node. That node may already have been reduced to make higher-level tree nodes, and so the new lookaheads need to be propagated up the tree. Care must be taken to make this process work correctly. A naïve implementation would loop infinitely, as parse trees can be cyclic. Even if cycles are detected, a brute-force approach that walks all combinations of parent relationships in the tree will not terminate quickly for complex grammars because of a combinatorial explosion. Itemising all of the routes through a cyclic parse tree can require the same level of complexity as itemising all paths through the state machine: an impossible task for some real grammars (including the Java exposition grammar). Our design detects cycles and avoids unnecessary re-work by stopping lookahead propagation as soon as a tree node's lookahead is found to be unchanged after a new lookahead calculation.

We can detect anchored lookaheads in parse trees; they are the ones in which no `LookPoppedTree` participates. All anchored lookaheads are stored in a `static Set` in the `LookaheadSet` class of the `dynamicStateMachine` package, so that they can participate in every `LookaheadSet`.

4.3.2.7 Dynamic state machines

The remaining package of the parser generator is `dynamicStateMachine` (Figure 94), which defines a specialisation of the runtime state machine in order to support the evolution of lookahead and states brought about by `Transmogrifiers`.

`YakState` extends its superclass implementation by keeping track of incoming transitions as well as outgoing ones, in order to enable the creation of `PopNodes` that back up through the machine. It has a number of methods that assist with initial construction of the state graph, including methods to remove ε–transitions and to merge states that are reached on the same transition. It also provides constructors that split existing states and re-route incoming transitions.

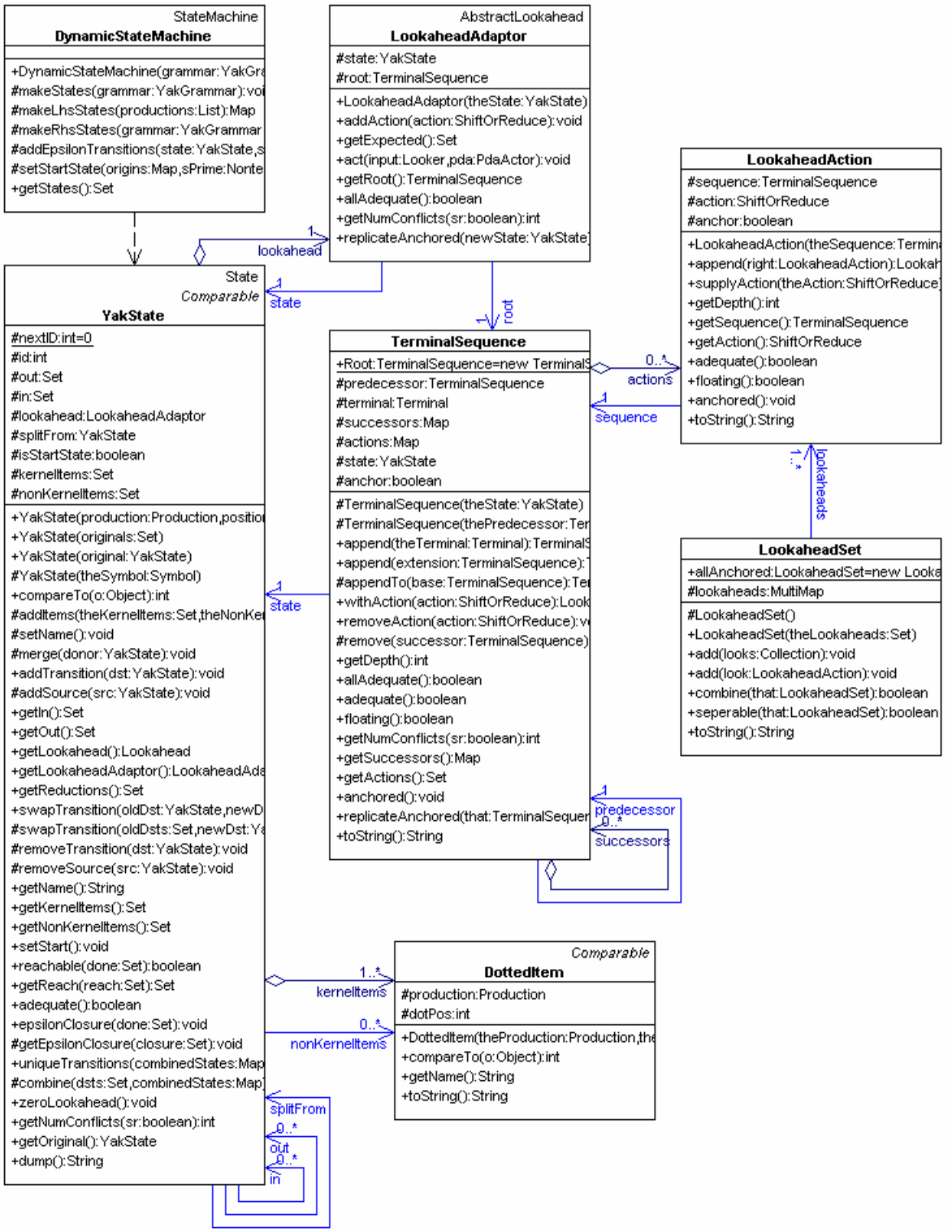

Figure 94: Dynamic state machine classes

`DottedItem` models the 'dotted items' of conventional parser generation algorithms. It takes no part in PDA construction in our algorithm, but serves as a useful mechanism for comparing our results with other approaches.

The lookahead implementation provided in the `dynamicStateMachine` package differs from the runtime lookahead in order to allow lookahead growth and separation. The essence of the design can be discerned from the class diagram and the code is largely straightforward.

4.4 Discussion

We have presented a new practical approach for generating a range of hybrid LR parsers including SLR(*k*), LALR(*k*), and LR(*k*), by escalating the power of the parsing algorithm and depth of lookahead only for states that need it. The approach makes use of an extended variant of GLR parsing to explore its own state machine and calculate lookaheads, and to split states when doing so improves lookaheads. To our knowledge this approach has not been tried before.

We have also described the design of an original framework that integrates different executable LR parser automata implementations, as well as our new LR parser generator approach. We know of no other parser generator that produces such a broad range of practical parsers. The design is very simple (compared to other parser generators), especially considering the variety of parsing algorithms included.

The following sections describe our contribution in the light of earlier work.

4.4.1 GLR parsing

We advocate the use of GLR parsing of programming languages when other LR parsing classes prove insufficient. This is an important contribution of this work, because it is essential to our strategy of using powerful parsing to allow standard (or definitive) grammars to be used without modification. The advantages of doing so are ease of development and improved rigour of downstream analysis.

Other authors have noted the advantages of GLR parsing of programming languages, for example van den Brand et al. [105] and McPeak and Necula [72]. Johnstone et. al [58] say:

> In the last decade the computing community has shown an increasing interest in parsing techniques that go beyond the standard approaches. There are a plethora of parser generators that extend both top-down and botton-up approaches with backtracking and lookahead constructs. As we have noted elsewhere such parsers can display surprising pathologies: in particular parser generators such as PRECC, PCCTS, ANTLR and JAVACC are really matching strings against *ordered* grammars in which the rule ordering is significant, and it can be hard to specify exactly what language is matched by such a parser. In any case, backtracking yields exponential parse times in [the] worst case.
>
> A safer approach is to use one of the truly general context free parsing algorithms such as Earley, CYK or a variant of Tomita's GLR algorithms.

We have produced two GLR parser implementations: a Java implementation of Rekers algorithm, and our own object-oriented design. Most parsing literature uses pseudocode and most parsing tools use procedural designs, so our design provides a contrast and a useful archetype for parser developers who wish to employ object-oriented technology.

GLR parsers are not widely used, but some other implementations are available. ASF+SDF (descended from Rekers work) [104] and Elkhound [72] are examples. Bison [27] is a very widely used LALR(1) parser that now claims to support GLR parsing. As we have noted, however, some commentators [56] suggest the current implementation is seriously flawed.

As we discussed earlier, Nozohoor-Farshi corrected Tomita's GLR algorithm to accommodate ε-reductions and cyclic reductions, and Rekers provided an implementation. Like theirs, our variety of GLR parser works for empty reductions and cycles, but we substitute a

bounded linear search (explained in Section 4.3.1.5) in place of a laborious search through the top of the graph-structured stack. We have not seen this enhancement elsewhere.

Although we reject cyclic grammars supplied to yakyacc as errors (because we assume that no designer of a programming language would want infinitely ambiguous sentences), we make use of our GLR parser's ability to handle cycles to implement our parser generators.

Rekers also makes some improvements to the amount of sharing of sub-trees in the parse forest. We have made no attempt to do the same.

Johnston et. al [57] have taken a different approach to avoiding infinite loops in a Tomita parser: they automatically modify grammars to remove offending constructs. Their approach is known as Right Nullable GLR. It has a fast implementation (in part because it avoids the brute-force stack search) but the approach of modifying grammars is contrary to our goals.

When using GLR parsing, it is possible to use any of the lower-powered LR automata as the state machine that is executed nondeterministically. There has not been a lot of research into the performance characteristics of different internal automata, but one study by Johnstone et al. [56] finds that the use of LR(1) automata within GLR parsers affords little advantage over using SLR(1) or LALR(1) automata internally. Longer lookaheads were not investigated.

4.4.2 GLR-based sub-sentence parsing

We have described a new extension of GLR parsing that allows parsing of fragments of sentences. It achieves this by placing cyclic 'pop nodes' at the base of the stack to allow popping into unknown territory, and thus produce all possible parses of the sub-sentence. The cyclic graph structure avoids a combinatorial explosion of pop nodes.

Rekers also used a GLR parser variant for sub-sentence parsing, but his approach used an approximation of goto behaviour akin to SLR lookahead calculation. In fact, if our `KParser` were implemented to use an `SLRPopNode` in place of the more accurate `PopNode`, it would work exactly like Rekers version.

4.4.3 Hybrid parsing algorithms

Our parser generator escalates its parsing algorithm to produce an automaton of hybrid power. This has the advantage of keeping each portion of the automaton as simple as possible. More importantly, it leads to the technique of building an LR(*k*) parser (for k > 1) by splitting states, and doing so only when it actually helps. We call the resulting parsers *minimal* LR(*k*), to differentiate them from canonical LR(*k*) parsers produced by Knuth's algorithm [61], which maximally splits all states and is consequently impractical for real languages. Our state-splitting technique (combined with the heterogeneous *k* technique discussed below) yields practical LR(*k*) parsers. We are not aware of another current parser generator that does so.

State splitting has been proposed before as a way of achieving practical LR(*k*) parsers. In fact, the first paper to describe LALR(*k*) raised the possibility [22]. An algorithm was produced by Pager [82], [83]. This "lane-tracing" algorithm examines dotted items to determine the paths down which lookahead flows, and thus to find states that can be split. Lane-tracing is conceptually similar to the way our GLR parsers explore backwards from a start state. However, the description of the algorithm is complex and in places lacking detail, especially of how splitting is achieved efficiently. It is described fully only for LR(1) parsers, but Pager reports success with longer lookaheads on real grammars.

The approach that is, perhaps, most similar to ours was produced by Fischer [32], although he had different motivations. Fischer escalated parsing power from LR(0) through NQLALR(1) and LALR(1) to LR(1). (NQLALR(1) is essentially an incorrectly implemented LALR(1) parser; it can be obtained with our approach by having goto nodes pop to all states that have a transition to the goto state, rather than just popping to the previously pushed state.) Fischer used conventional (dotted item) parsing algorithms, and a version of lane-tracing for splitting states. He did not use $k > 1$.

An algorithm to expand an LR(0) parser into a full LR(1) parser is presented by Spector [96] and later extended [95]. Spector describes Pager's algorithm as "extremely slow". He further states:

> Perhaps what has been lacking to popularize full LR(1) parser generation has been that no one seems to have created an easy-to-understand and efficient algorithm. This paper (and an experimental 2300-line program written in C) takes a first step in that direction.

The spirit of the algorithm appears similar to ours: "it determines look-ahead sets by searching the underlying LR(0) FSM". Nevertheless, Spector's papers do not specify the algorithm in sufficient detail to allow them to be implemented by others.

An alternative (not state-splitting) route to obtaining practical LR parsers is described by Korenjak [62].

4.4.4 Heterogeneous k

Conventional parsers use lookahead depth of only 1, in order to keep parse table sizes in check. Our parser generator avoids table-driven implementations and grows lookaheads only for states (in fact, only for some lookaheads within states) that need more lookahead, producing an automaton with varying lookahead levels.

Parr's thesis [86], as already noted, promoted the advantages of heterogeneous depths of lookahead. His work concentrated on LL parsers, but the findings were applied to LR parsers, with the exception of LR(*k*). This exception was based on the assumption that LR(*k*) parsers could be built only using the dotted item algorithm. The use of a state-splitting algorithm also enables heterogeneous *k* for LR(*k*) parsers.

Pager's lane-tracing algorithm could also produce heterogeneous *k* parsers [82].

4.5 Evaluation

For the purpose of constructing rigorous static analysis tools, yakyacc represents a substantial improvement over current parser development practice. It provides tool builders with an integrated set of facilities that were, in any practical sense, previously unavailable. By this measure alone the development has been highly successful.

Our motivation for developing new parsing technology is to facilitate construction of new software engineering tools, and in this regard yakyacc has also demonstrated its utility. The following chapters describe complex applications built on yakyacc. These include our metrics and visualisation pipeline, Cook's collaborative IDE [19], and Neate's CodeRank engine [77]. These applications have tested yakyacc-generated parsers much more thoroughly than is normally the case for research tools. Moreover, we know of no other tool that would have allowed these applications to be constructed as efficiently.

Yakyacc has been tested using a battery of JUnit tests, and a collection of over a dozen test grammars designed to elicit problems across a range of lookahead depths and parsing classes. More tellingly, we have generated parsers for difficult real grammars, including the ambiguous Java exposition grammar, about which Tucker and Noonan say "The complete Java syntax is immense in its number of grammar rules" [102]. The behaviour of our Java parser has been verified by successfully parsing hundreds of thousands of lines of Java, including many open source programs. We have also conducted back-to-back tests to compare the behaviour of our original PDA implementation against the alternative Rekers-derived version.

We have also generated parsers for yakyacc's own utilities (bnf2xml and ebnf2xml). An early prototype successfully tested our GLR implementation using the C++ grammar (and also the C pre-processor grammar) on 250,000 LOC.

The runtime performance of generated parsers depends on the stylesheets used for code generation, and we have not yet made any formal measurements of the parser as a separate process. Parser performance has always met user expectations, even in very demanding real-time settings.

Parser generation normally takes a matter of seconds for parsers of lower power, even for very large grammars. A large portion of this is IO overheads, reading in the grammar and writing out the PDA. For difficult parsing classes (LR, or higher values of *k*), time is highly dependent on the particular grammar used and the parsing class and lookahead depth. Generating an LR(1) parser for Java takes on the order of 15 minutes (running in an IDE on a standard workstation). The current implementation makes no concessions to efficiency, and is profligate with processing cycles in places. Further experimentation is needed to establish

a clearer understanding of parser generation performance, including finding the realistic limits for *k* for real grammars.

4.6 Chapter summary

We have described a new parser generator with sufficient power and flexibility to enable a fundamental shift in how parsers are developed for static analysis purposes: the parsing technology adapts to suit the given grammar. By using an original GLR-based parsing algorithm, we have been able to combine several previously separate techniques in one tool, along with some new innovations, while keeping the design relatively simple. The result is easier parser development and more rigorous static analysis.

Chapter 5

JST: Semantic modelling of Java code

Parsing exposes the *surface* structure of source code. Semantic analysis examines the *deeper* structure and, in our work, exposes it in the form of a semantic model. Semantic analysis recognises the conceptual entities that comprise software, rather than merely the syntax that describes those entities. It also recognises relationships between entities, and consequently models a program as a connected graph, rather than as discrete trees in the way that parsers must.

The term *semantic* is overloaded, and to avoid confusion we note that our work is concerned with programming language semantics, rather than with the semantics of problem domains for which programs are written. In other words, we address concepts that are found in programming language definitions (classes and methods, for instance), and not concepts that are only found in specific applications (such as customers and accounts).

Even within the field of programming language semantics, *semantic analysis* encompasses a range of activities, including formal ways of describing program behaviour such as operational semantics, axiomatic semantics and denotational semantics [102], and also analysis of code characteristics such as semantic error checking, statement reachability, data flow, and so on. We do not undertake these specific semantic analysis activities in our work; we are instead interested in software structure information—the *type system* of the language—that underpins all forms of semantic analysis, and also defines the data we need for metrics, visualisations and other tools.

By modelling the way in which a program uses the type system of its language, we can make semantic concepts and relationships explicit, whereas they are implicit in parse trees. For example, an `extends` clause in a parse tree provides the name of a superclass, but the superclass itself is defined in another unconnected parse tree, and there may be more than one class with that name. Similarly, semantic relationships that exist between types and the variables that use them, variables and the expressions that use them, methods and their invocations and so on are not evident in parse trees; related entities are named but no identifying connection is made to their definitions. The fundamental task of our semantic model is to identify the entities from which a program is made and resolve named references by identifying target entities.

The task of looking up symbols (names) to resolve references is normally the responsibility of a *symbol table* in compilers. From this term the name of our semantic modeller is derived: Java Symbol Table (JST). The term *symbol table* evokes a simple data structure: a table indexed by names. In early programming languages, a simple table may sometimes have sufficed, but in current object-oriented languages, resolving names is a complex task involving convoluted scope topology. Packages, source files, classes, inner classes, methods, blocks and other semantic concepts all influence the look-up process, as do different relationship types such as inheritance and containment, and look-up behaviour is modified by access restrictions (`private`, `protected`, etc), `static` modifiers and other mechanisms. A particularly challenging issue—one that defeats many experimental tools—is resolving calls to overloaded methods. In a language such as Java, almost the entire type system participates in resolving method calls. It is necessary, for instance, to know the types of any expressions used as parameters, because parameter types are significant in choosing among candidate methods.

Although JST is an entirely new application, its design was influenced by work on symbol tables for Java and C++ [66], [55]. Unlike those earlier tools, JST's model is complete, including resolving overloaded method invocations.

We have previously described JST in [49] and [51]. Some passages of this chapter appear in those publications.

5.1 Why Java?

Different languages support different semantic concepts. Unlike yakyacc, which by definition can generate a parser for any language, our semantic model had to represent the elements of a specific language in order to allow us to devise exact metrics and visualisations for that language. Pragmatically, we also wanted to develop and test our semantic modelling approach without the additional burden of making it language independent. In subsequent work, we have extended the approach to cater to a wide variety of languages by modelling the semantic entities of .NET [76], and by mapping them to our JST model [46]. This chapter, however, documents the original JST model on which the extensions were based.

The language had to be object-oriented—because we are concerned with exposing OO structure—and statically typed. Statically typed languages present more information to static analysis tools than dynamically typed languages. This is unsurprising, of course, since the objective of static typing is to allow code to be checked before program execution.

Our initial attempts at semantic modelling were aimed at C++, in order to support a project with a large (2,000,000 LOC) commercial C++ code base. This exercise was instructive, and provided the initial motivation for a stronger parser generator, as explained in [48]. Yakyacc solved the problem, and allowed development of a C++ semantic model decoupled from the parser. Even so, we changed to modelling Java instead of C++, primarily because C++ uses a pre-processor and Java does not. The use of a pre-processor substantially complicates the task of static analysis, because the code seen by the tools does not match the code seen by the programmer; this inhibits communication of metrics and visualisations, for example. To illustrate the extent of this problem, in the C++ code we examined the `#include` pre-processor mechanism multiplied the number of lines of code 250 times.

The version of JST documented here conforms to Java 1.3. Since this work began, the C# language has emerged. It is similar to Java, but offers some advantages for static analysis. In particular, it improves the static type system by including *generic types*. Java has subsequently been enhanced in a similar way and is now fully type safe, providing richer type information than before (although its *type erasure* approach means Java metadata is not as helpful as the .NET alternative). For example, Java collections previously were defined to hold only `Objects`, and so a static analyser could not readily determine relationships to ac-

tual classes of objects held in the collection. With the generic types of Java 1.5, the actual classes are available to a static analyser (of source code, rather than `.class` files). This evolution of mainstream languages towards fully type safe semantics indicates an increasing recognition of the value of static analysis information, and underlines the value of our approach. Recent work that we have conducted to accommodate improvements in .NET and Java is recorded in [76], [46] and [12].

5.2 The type system of Java

The Java Language Specification [36] is a document of some 500 pages. Its bulk is due not to the complexity of the syntax (which is described concisely by a grammar), but to the exposition of the language's semantics. Like any OO programming language, Java has a rich system of types and values. The language specification describes this system, including how the concepts of the language are declared and accessed. These concepts include features such as classes, interfaces, inheritance, methods and control structures, and the rules governing their use, including scoping, type conversion, overload resolution, hiding, multiply inherited fields, and so on.

Java has been promoted as a simple OO language and, in comparison to C++, it is. Nevertheless, its type system and scope rules are elaborate and sometimes subtle. The language specification contains many details and special cases that are likely to be outside the sphere of knowledge of typical users of the language. For example, the specification contains statements such as "Inner classes may inherit static members that are not compile-time constants even though they may not declare them" (p 140) and "If an anonymous class instance creation expression appears within an explicit constructor invocation statement, then the anonymous class may not refer to any of the enclosing instances of the class whose constructor is being invoked" (p 194). Examples of some of the language's darker corners can be found in [5].

Although human users of a programming language can avoid intimacy with obscure language features, rigorous software tools should not. Our objective in building JST was to capture and expose a complete model of the type system of a Java program suitable for comprehensive static analysis. In the Java Language Specification, semantics are described in terms

of an exposition grammar. The final chapter of the document introduces an alternative grammar, saying: “This chapter presents a grammar for the Java programming language. The grammar presented piecemeal in the preceding chapters is much better for exposition, but it is not ideally suited as a basis for a parser. The grammar presented in this chapter is the basis for the reference implementation.” (p 449)

The suggestion that the exposition grammar is unsuited to parsing arises, we suspect, because it is not LALR(1). (Neither is it LR(k), because it contains fundamental ambiguities.) Using conventional parsing technology, developers of static analysis tools for Java are confronted with a problem: the semantics of the language are defined in terms of one grammar, but parse trees conform to another. As a result, developers must choose either to map semantic rules onto the implementation grammar, or to transform parse trees so they conform to the exposition grammar. Yakyacc eliminates this dilemma by generating a parser of the exposition grammar. Ambiguous constructs may be pruned from the parse tree afterwards, using simple semantic rules.

Our semantic modelling approach takes advantage of the conformance of the yakyacc-generated parser to the exposition grammar, to model semantic concepts with high fidelity to their descriptions in the language specification.

5.3 Development approach

A small number of Java symbol tables are already available. Stanchfield and Parr [99] describe a symbol table that is part of a simple cross-reference tool for Java 1.1. It was subsequently developed into `javasrc`, an open-source hypertext cross-referencer for Java 1.3 [55]. These tools do not attempt to resolve overloaded method calls, and have several other limitations, including simplified package naming, some syntax limitations, and no handling of anonymous classes or member access specifiers.

We investigated improving code from `javasrc` as the basis for our symbol table, replacing the ANTLR-generated parser with a simple reader of our XML parse trees. However we ultimately chose to design our own classes to more closely reflect the concepts described in the Java Language Standard. This allowed us to be more confident that our code conforms to

the standard, and gave us a good structure on which to graft the missing features, including overloaded method resolution.

An alternative approach using Java's reflection API was also considered. We are interested only in analysing syntactically correct source code, and such code will have a corresponding `.class` file emitted by a Java compiler. Java compilers embed metadata—symbol table information—in `.class` files so that this information may be reported by the reflection API. Typically, Java programs use reflection to dynamically load classes that were unavailable at compilation time. In Chapter 2 we noted that the reflection API constitutes a form of semantic model, although it omits some details and abstracts others. We might, however, use reflection to derive our own more comprehensive model.

We investigated using reflection to extract symbol table information, and then using that information to connect semantically related portions of our parse trees. The appeal of this approach was threefold:

- All of the symbol table information was available in advance, so it reduced the complexity added by order dependencies between declarations and look-ups as they were discovered in the parse trees (because properties may be used before they are declared, for instance).

- Symbol table information was available for classes for which source code was not present, including standard library classes such as `java.lang.Object`.

- Third-party support for resolving overloaded method calls was available.

The reflection API can look up a method, given its name and parameter types. This is sufficient data for the API to perform method resolution, but as Hosler [44] has noted, it does not. The reflection API performs only an exact match on parameter types, whereas full method resolution must consider type promotions and find the most specific method from a set of applicable methods. Hosler provides a `BetterMethodFinder` class that extends the reflection API to include proper method resolution.

This reflection-based method resolution proved to be less helpful to us than anticipated. The reflection API is designed to expose the public interface of a class, and the method resolution

is accordingly suitable for client classes that are not part of the protected, private or default (package) scopes. This restriction, while appropriate for reflection's intended purpose, does not necessarily hold for the classes we are analysing; we need to resolve calls of all methods, not just public ones. In addition, we are interested in a class' internal use of methods, fields, local variables, and parameters. This information is not available through reflection.

We concluded that reflection did not eliminate the need to build our own complete symbol table from parse tree information, but was useful for resolving references to classes for which we did not have source code. This means that our symbol table is populated with declarations found in parse trees, and whenever a look-up references some external symbol for which source code is not present, (`java.lang.String`, for instance) the missing information is supplied by reflection.

Our semantic model is, in essence, the result of an object-oriented data modelling exercise for the domain of the Java type system. The model represents the scopes and declarations that can be identified in the language specification and implements their look-up behaviour.

5.4 JST architecture

Parse trees represent a program as discrete translation units, but semantic dependencies connect them into a single graph. In order to make all semantic dependencies explicit, the semantic model must span an entire program. JST builds the semantic model monolithically: all parse trees for a program are loaded and the complete model is assembled in memory before being saved as an XML file. This approach keeps implementation relatively simple, at the expense of requiring sufficient memory to contain the entire model. (Section 5.8 discusses an extension that allows the model to be constructed incrementally.)

JST runs once per program, much like a conventional linker. It reads all the parse trees for a program into memory, and walks through them to populate the symbol table with declarations and look them up to resolve references. Any references that are not resolved by declarations from the source code are supplied by reflection. When the process is complete, the symbol table is exported as an XML file, which also contains all the parse trees.

5.5 JST model

The objects that make up the symbol table are:

- Packages and source files.
- Built-in types: the primitive types (`int`, `boolean` and `void`, for instance), the null type and arrays.
- User-defined types: classes and interfaces. Here *user-defined* means types that are defined by a programmer in Java code, that is *all* classes and interfaces, including library classes such as `java.lang.Object` and `java.lang.String`.
- Typed declarations: declared entities (variables or operations) that have a type, that is, fields, local variables, parameters, methods and constructors.
- Executable sections of code: blocks and field initialisers.

This is the complete set of elements of the Java type system. The objects are all concepts with which every Java programmer is familiar and the reasons for including most of them are self-evident. However, the inclusion of source files, blocks and field initialisers warrants further explanation:

- Although source files are not declared within source code (but instead contain it), they are included in this set because they define scopes that are significant in look-ups of packages, classes and interfaces.
- Similarly, blocks (sequences of statements delimited by braces) define scopes that contain local variables, classes, interfaces and other blocks. However, statements other than declarations and expressions within blocks are not represented as semantic model objects because they act only as clients of the type system and do not define elements of it. They are adequately described by their syntactic structure. The client relationships between semantic model objects and the expressions that use them are represented in the semantic model, as we explain below.
- Field initialisers, which provide initial values for class fields, are a special case in which expressions that use semantic model objects do not occur within blocks. We include field initialisers in the model to provide a place for storing these client relationships.

All direct relationships between semantic model objects are modelled. A containment hierarchy is defined by the following relationships (all one-to-many or one-to-one):

- A symbol table contains the default package.
- A package contains sub-packages.
- A package contains user-defined types.
- A user-defined type contains inner user-defined types.
- A user-defined type contains fields.
- A user-defined type contains methods.
- A class (a special user-defined type) contains constructors.
- A class contains initialiser blocks.
- A field contains a field initialiser.
- An operation (a method or constructor) contains parameters.
- An operation contains a block (the body of the operation).
- A block contains inner blocks (including catch blocks).
- A catch block (the scope of a catch statement) contains a parameter.
- A block contains local variables.
- An executable section of code (a block or initialiser) contains user-defined types.
- A type contains an array of that type (which is itself a type).

This containment hierarchy spans the entire contents of a program. (It does not, however, span the entire semantic model. The portion of the semantic model that represents built-in types is not included in this hierarchy, in the same way it is not included in any program.) Beneath the level of packages, the containment hierarchy reflects the syntactic containment structure of programs and needs no further explanation. At the top of the composition hierarchy, packages—rather than source files—contain classes. This design was chosen because source files are not available for all classes, but we still need to model classes obtained by reflection. All classes, however, must reside in a package.

In addition to the containment relationships, several association relationships are modelled. Omitting reciprocal relationships, the associations are:

- A package is defined by source files.
- A source file imports scopes (*import on demand*). The scope must be a package or user-defined type.
- A source file imports user-defined types (*import a single type*).
- A source file defines user-defined types.
- An interface extends super-interfaces.
- A class extends a superclass.
- A class implements interfaces.
- An operation throws classes.
- An executable section of code refers to typed declarations (e.g. uses a variable or invokes a method).

Most of the association relationships listed above are straightforward reflections of the Java type system. An exception is the last one, the *refers to* relationship from executable code sections (blocks and field initialisers) to typed declarations. Executable code sections contain expressions that use declared variables and operations. The *refers to* relationship connects each such expression in an executable code section with the semantic model object that the expression uses. In this way, client relationships produced by expressions are recorded in the nearest enclosing semantic model object (without loss of information—the parse tree nodes of the expressions are also referenced). This means that the semantic model is self-contained: all relationships between semantic model objects can be found without reference to parse trees if desired.

The listed objects and relationships comprise the entire semantic model. The model includes all direct relationships that occur between elements of the Java type system, including inheritance, data usages and method invocations. JST implements full resolution of overloaded methods (saving the result in *refers to* relationships), using type promotions, overriding and access control. It connects all declarations with their types, and all usages of declared properties with the declarations. The original parse trees are retained, with references into them from the semantic model.

Figure 95 provides an overview of the classes in the model. Although the diagram appears complex, the objects and relationships represented are only those listed above. Class type declarations are represented by `ClassType`, for instance, and method declarations by `MethodDecl`. The number of classes is somewhat greater than might be expected from the list above because of the use of abstract classes to capture generalisations. For example, `UserType` (corresponding to a user-defined type) is an abstract class that captures the commonalities of `InterfaceType` and `ClassType`. A level higher, `ReferenceType` captures the common features of `UserType` and `ArrayType`. Higher still, `TypeDecl` generalises all types. Containment relationships can be recognised in the diagram by the diamond-shaped aggregation symbol.

The next sections supply additional detail about the design of the semantic model classes.

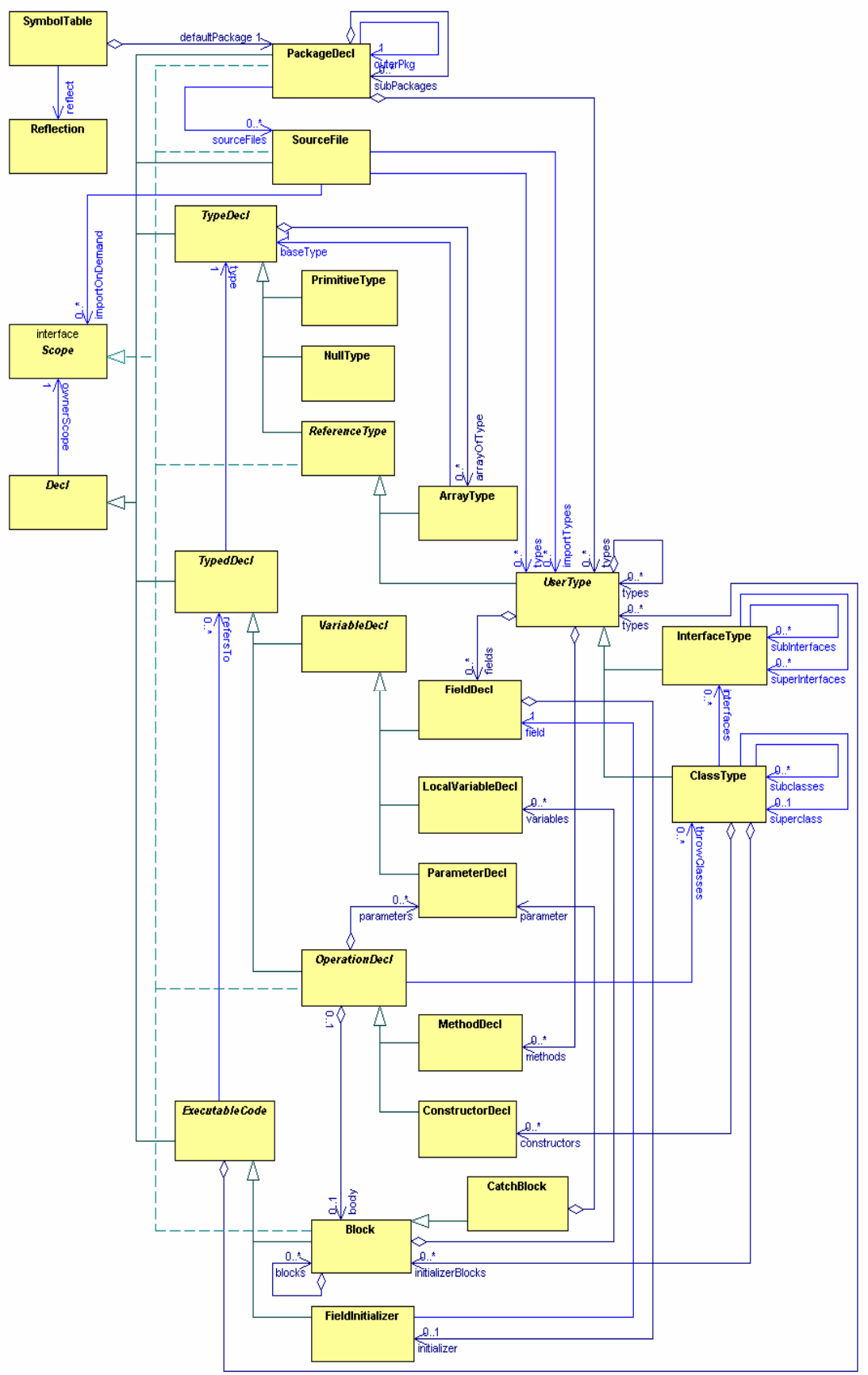

Figure 95: JST overview class diagram

5.5.1 *Main classes*

The entrypoint for the JST semantic model is `SymbolTable` (Figure 96). It contains the *default package* object, which ultimately contains all other declarations of a program. `SymbolTable` provides entry points for adding semantic objects and for looking them up in order to build relationships. It is a lightweight class (for a symbol table) that delegates model representation and lookup to the declaration classes themselves.

When JST is executed, it first invokes `SymbolTable`'s `addParseTree()` method for each parse tree. This method walks through a parse tree to locate declarations and instantiates them as model objects of the appropriate class. This process builds the containment hierarchy, but not association relationships. Once all parse trees are loaded, the `crossReference()` method is called to find association relationships. This process indirectly invokes the public `get…()` methods of `SymbolTable`, which look up names in the scope structure. The details of populating the model are described in the Section 5.6.

The `Reflection` class encapsulates the Java reflection API. Whenever a look-up fails to find a named element, `SymbolTable` passes the request to the `Reflection` class. `Reflection` instantiates the requested element as part of the model and, using a greedy approach, also instantiates all related elements that are revealed by the reflection API. In this way, library classes used by source code are included in the semantic model so there are no dangling references. Reflection provides only the public interfaces of classes, however, so the internal structure of reflected classes is not modelled.

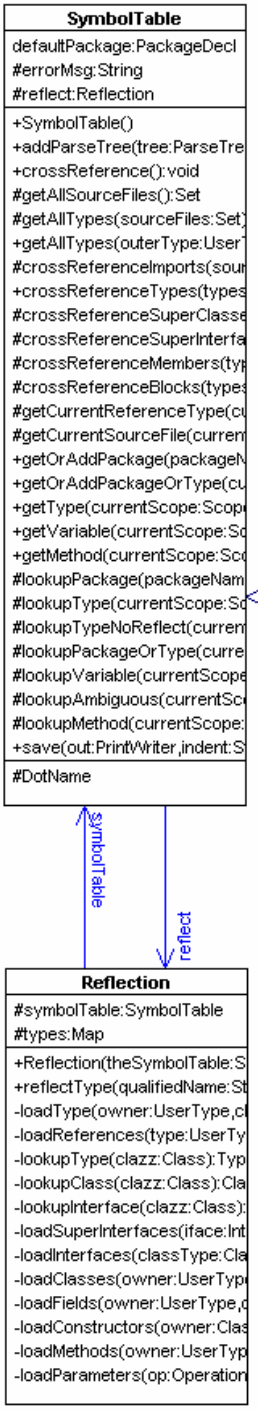

Figure 96: Main JST classes

`Scope` and `Decl`, shown in Figure 97, are the main abstractions of the model. A `Scope` is a container of declarations, and provides lookup functions to retrieve the declarations. Although scopes vary in what they can contain—a package scope, for example, can contain

classes but not methods—`Scope` provides a lookup method for every possible search. Irrelevant searches simply return an empty result. `Scope` is implemented by `PackageDecl`, `SourceFile`, `ReferenceType`, `OperationDecl` and `Block`.

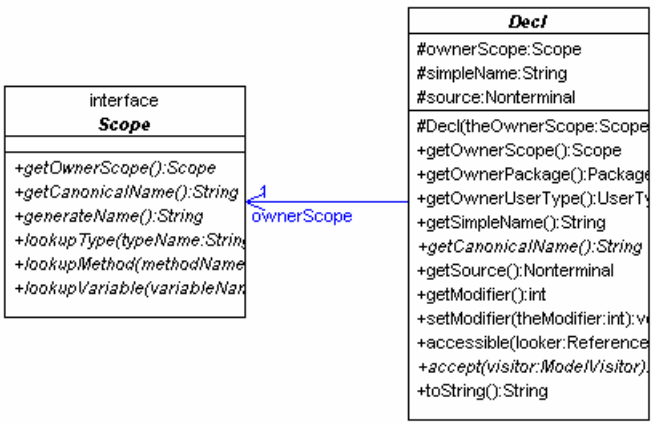

Figure 97: Scope and Decl classes

`Decl` is the root of the declaration hierarchy. `PackageDecl`, `SourceFile`, `TypeDecl`, `TypedDecl` and `ExecutableCode` inherit from it. All declarations have a name and occur in some scope; names are generated for anonymous declarations. The concept of a declaration has been broadened to encompass all relevant semantic features in a Java program; source files and blocks (sequences of statements delimited by curly braces) are also considered to be declarations. This simplifies the design by allowing all program features to be treated consistently at an abstract level. Similarly, all `Decl`s can return a modifier (`public`, `final`, `static`, etc), although only member declarations will return non-zero values. (We represent modifiers as bit-mask `int`s to be consistent with the Java reflection `Modifier` interface.)

Any declaration that can contain other declarations is also a `Scope`, and consequently provides lookup methods to retrieve its contents by name. Thus, declarations implement the scope and lookup rules of the language; each type of decla-

6.5.5.1 Simple Type Names

If a type name consists of a single *Identifier*, then the identifier must occur in the scope of a declaration of a type with this name, or a compile-time error occurs.

It is possible that the identifier occurs within the scope of more than one type with that name, in which case the type denoted by the name is determined as follows:

- If the simple type name occurs within the scope of a visible local class declaration (§14.3) with that name, then the simple type name denotes that local class type.
- Otherwise, if the simple type name occurs within the scope of exactly one visible member type (§8.5, §9.5), then the simple type name denotes that member type.
- Otherwise, if the simple type name occurs within the scope of more than one visible member type, then the name is ambiguous as a type name; a compile-time error occurs.
- Otherwise, if a type with that name is declared in the current compilation unit (§7.3), either by a single-type-import declaration (§7.5.1) or by a declaration of a class or interface type (§7.6), then the simple type name denotes that type.
- Otherwise, if a type with that name is declared in another compilation unit (§7.3) of the package (§7.1) containing the identifier, then the identifier denotes that type.
- Otherwise, if a type of that name is declared by exactly one type-import-on-demand declaration (§7.5.2) of the compilation unit containing the identifier, then the simple type name denotes that type.
- Otherwise, if a type of that name is declared by more than one type-import-on-demand declaration of the compilation unit, then the name is ambiguous as a type name; a compile-time error occurs.
- Otherwise, the name is undefined as a type name; a compile-time error occurs.

This order for considering type declarations is designed to choose the most explicit of two or more applicable type declarations.

Figure 98: Example name look-up rules

ration knows its own structure and rules for looking up names. If a lookup fails locally in a scope, the request is passed to other, higher-level scopes. In this way lookups search progressively wider scopes without the need for a current scope stack. For example, if we look up a method name in an inner class, the search will traverse the inheritance hierarchy, including interfaces, followed by the containment hierarchy beginning with the outer class.

The rules controlling name look-ups in Java are not trivial. For instance, the rules for looking up unqualified (simple) type names are shown in Figure 98. As can be seen in this description, the order in which scopes are searched is not always self-evident: when a source file is searched for a type, so too are *single type* imports (that is, import statements without a wildcard); the package that owns the source file is searched next, and finally the *import on demand* (wildcard) packages and classes specified in the source file are searched. JST implements all of the name lookup rules defined by the JLS.

Figure 99 shows details of `PackageDecl` and `SourceFile`. The `importOnDemand` collection of `SourceFile` contains `Scope` objects, which in practice will always be `PackageDecls` or `ClassTypes`. When searching for types, `SourceFile` calls the `LookupType()` method of these `Scopes`. Strictly speaking, `Scope` is a more general type than necessary (since other declarations also implement `Scope`), but the resulting design does not cause problems and is simpler than alternatives. The remainder of the design of `PackagDecl` and `SourceFile` is largely mechanical, providing implementations for the inherited contracts and getters and setters for relationships.

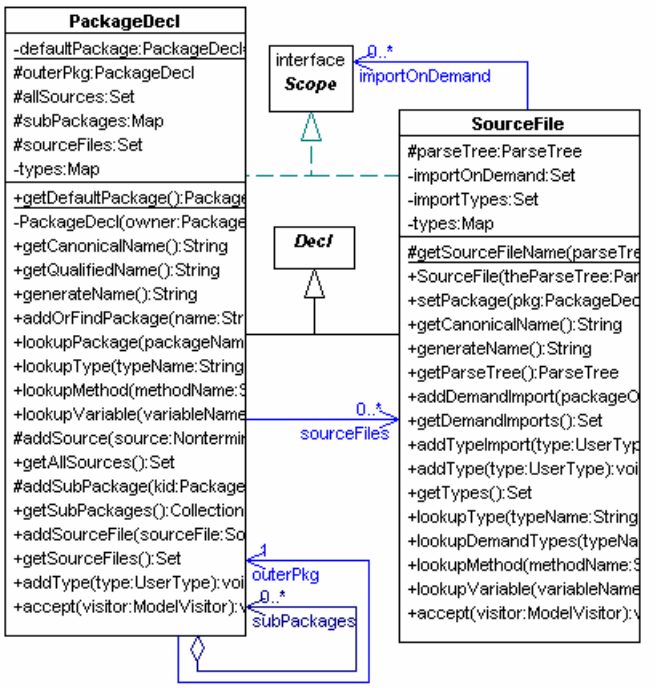

Figure 99: `PackageDecl` and `SourceFile`

5.5.2 *Types*

Figure 100 shows details of `TypeDecl` and its subclasses. Most of the methods are predictable: constructors for building the model, setters and getters for relationships, look-ups to implement the `Scope` interface in `ReferenceType` and its subclasses.

Primitive types and the null type are implemented as `public static` objects. Each `TypeDecl` may have only one `ArrayType` object, as the dimension of the array is a runtime property and not part of the type. `ArrayType` is an example of the decorator pattern, and allows nested decorators to form arrays of arrays, to any depth. Java arrays "magically" have a `length` field and `clone()` method. These are supplied by `ArrayType`.

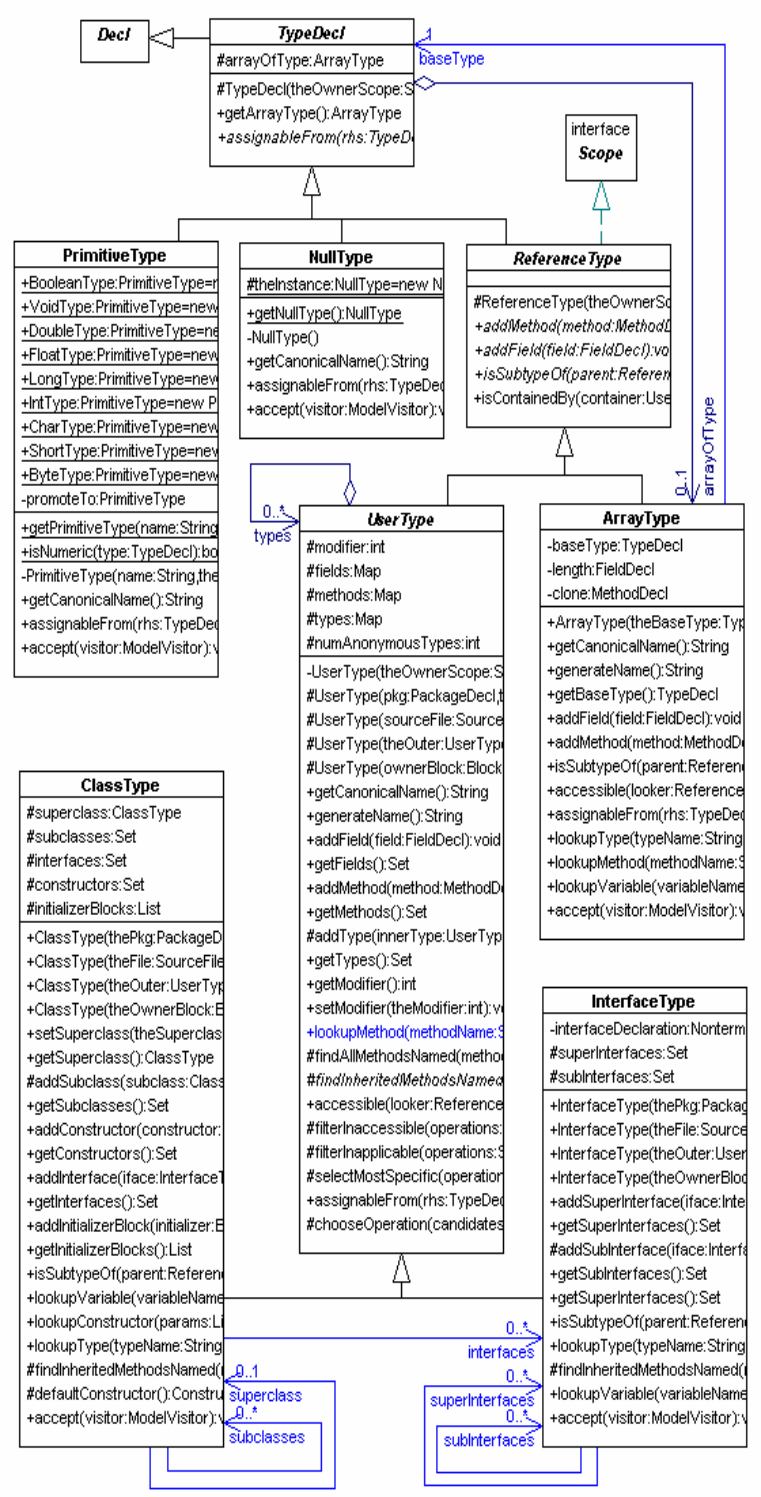

Figure 100: `TypeDecl` and its subclasses

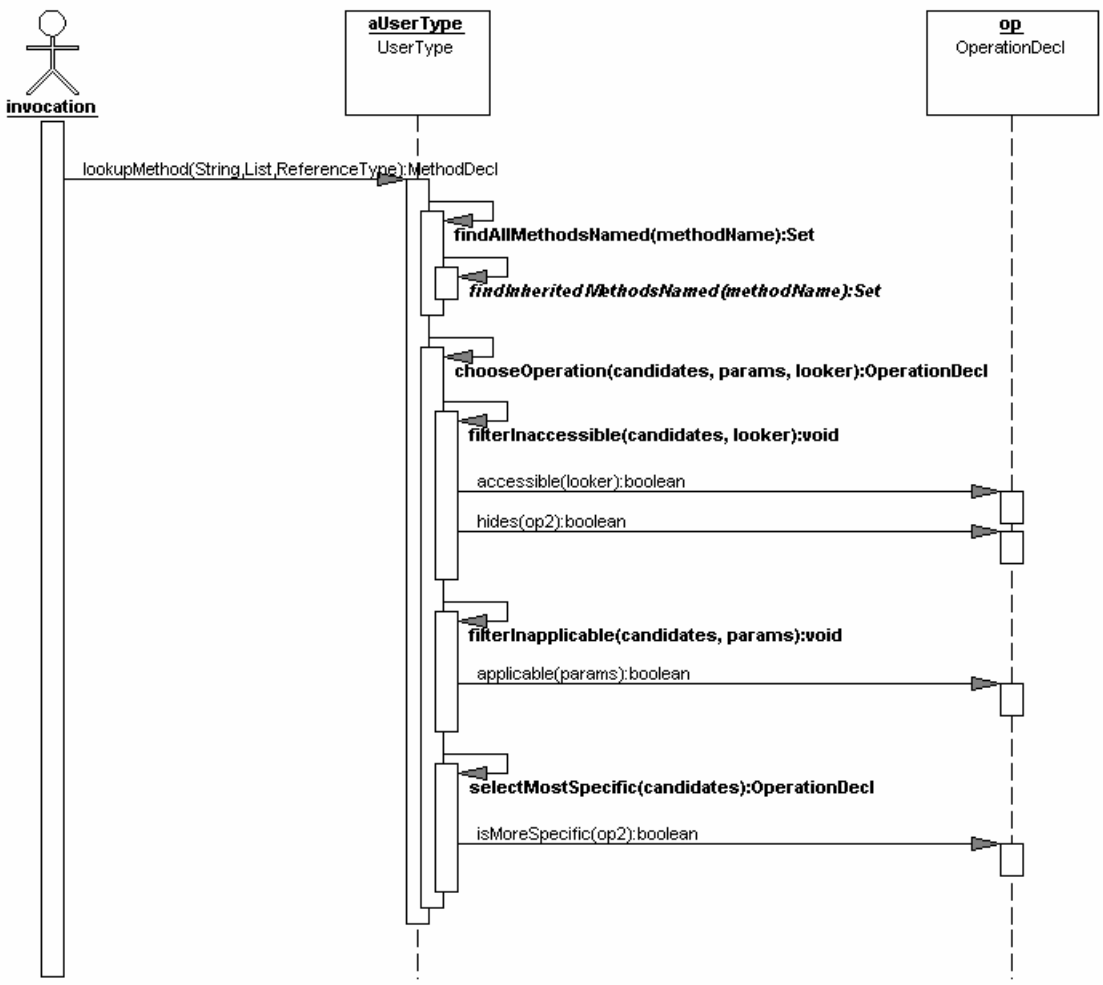

Figure 101: Method invocation resolution

A significant fraction of the implementation of type classes supports method look-up, including resolving overloaded method calls. Figure 101 is a UML sequence diagram of `UserType`'s `lookupMethod()` implementation. The full JLS description (§15.12) of how method invocations are resolved is too involved to warrant repetition here, but the main ideas are evident in our sequence diagram.

5.5.3 Typed Declarations

Figure 102 is a detailed class diagram for the hierarchy that represents declared variables and operations, using the `VariableDecl` and `OperationDecl` abstractions, respectively. Again, most methods and attributes of the design need no further elaboration. `OperationDecl` is a little more complex because it is a `Scope` that contains `ParameterDecl`s, and because it (with its subclasses) provides methods that support resolution of overloaded methods, as

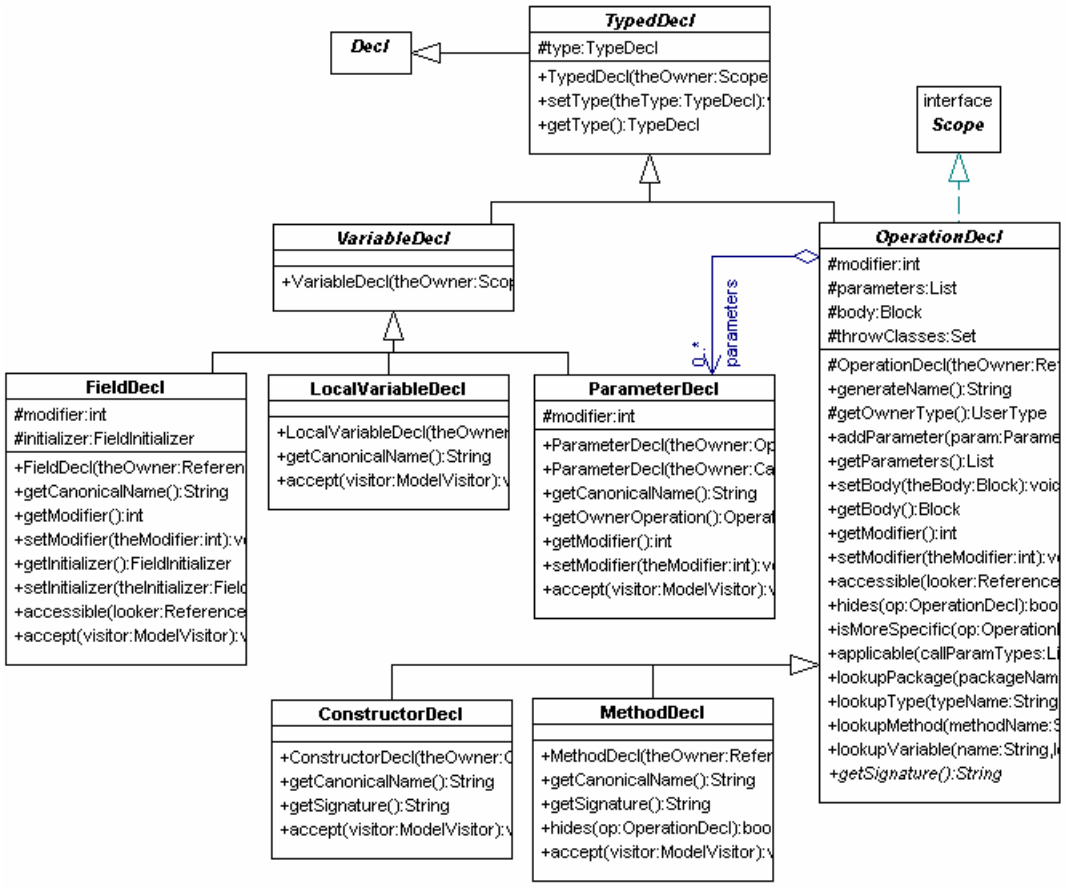

Figure 102: `TypedDecl` and subclasses

seen in Figure 101. The `accessible()` method checks access modifiers, `hides()` checks overriding in the inheritance hierarchy (including interfaces), `applicable()` checks that the types of expressions used as actual parameters can be assigned to the formal parameter types of the method, and `isMoreSpecific()` determines which of the possible methods most closely matches the actual parameter types. These last two methods make use of `TypeDecl`'s `assignableFrom()` method to handle type promotion and conversion. The vocabulary used for naming these methods is that of the language specification.

5.5.4 Executable classes

The remaining classes of the model are those that contain sections of executable code, shown in Figure 103. Although we describe the classes in this figure as the lowest-level elements in the semantic model, all `ExecutableCode` objects can contain `UserTypes` and so are not necessarily leaf nodes in the containment hierarchy. `Blocks` also add a collection of `LocalVariableDecls`, and `CatchBlocks` add a further `ParameterDecl`. As we have noted, when-

ever any expression is found within the parse trees of blocks or field initialisers, the `TypedDecl` named by that expression is identified and stored in `ExecutableCode`'s `refersTo` collection. The collection is implemented as a `Map`, in which the keys are `TypedDecl`s and the values are sets of parse tree nodes defining the expressions that refer to that `TypedDecl`.

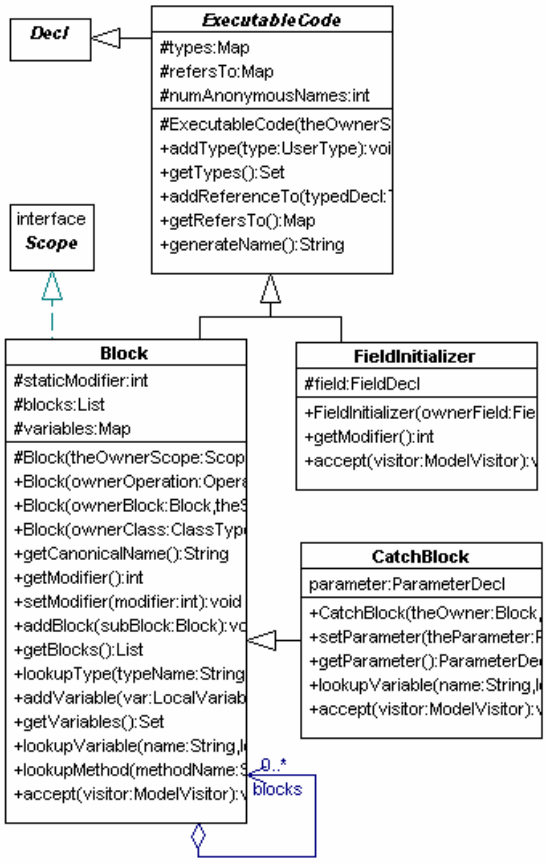

Figure 103: `ExecutableCode` and its subclasses

5.6 Populating the model

Many name look-ups in Java must occur in an order different from the syntactic structure of a file. For example, attributes may be used in a source file before they are declared. More generally, Java (unlike C++) does not distinguish the declaration of a feature from its definition, so the syntactic structure does not guarantee that features will be declared before they are used. A semantic analyser for Java might choose to remember unresolved references until the target declarations are discovered, or alternatively, might process parse tree nodes in an order that ensures declarations occur before usages. JST (in this version) takes the latter approach.

JST walks parse trees using the *visitor* design pattern [35]. As it encounters declarations, it instantiates the classes that comprise the semantic model. Multiple passes are made through any one parse tree, so that the model is assembled in increments. Some passes discover relationships, such as inheritance and invocations. JST imposes an order on its processing of parse trees to ensure that every feature of a program is declared before it is looked up, or needed for looking something else up. For example, all packages, classes, methods and parameters are declared before any method invocation is looked up. By ordering model construction in this way, relationships between objects are discovered only after the target objects are sure to be present.

Initially, all parse trees are examined for declarations; every declared feature in a nameable scope is instantiated in the model. These declaration objects are, at this stage, related only by their containment structure. Each declaration is given a name, but no names are looked up yet.

To complicate matters, some declarations (such as attributes) have semantic scope while others (such as local variables and classes declared inside blocks) have syntactic scope: They are accessible only from the point of declaration forward to the next closing brace. Syntactically scoped declarations are omitted from the initial population of the model, as they should not be found by lookups until the appropriate point in the parse tree is reached.

Once all initial declarations are known, they are connected together by looking up the names of features they reference. This cross-referencing happens in the following order:

- Packages and classes named in import statements are found. These must be known before classes can be looked up.
- Superclasses and interfaces named in extends and implements clauses are found. This inheritance structure is needed for subsequent lookups.
- The types of all class members (fields and methods) are found. Types must be known in order to analyse expressions such as a.b.c.d – the type of a must be known to determine the existence and type of b, etc.
- Each code block is processed statement-by-statement, in parse tree order. Lexically scoped types and variables are instantiated. Identifiers in expressions are looked up.

At any point during cross-referencing, a lookup of a named type (class or interface) will fail if a parse tree for that class or interface was not provided to JST. Whenever this happens, Java's reflection API is used to load public interface information from .class files so that the semantic model is complete and all references are resolved. Reflection, however, does not expose the internal features of the reflected classes, such as local variables or method invocations.

5.7 Emitting the model

At the end of this processing, all declarations, types, scopes and the connections between them have been constructed and the semantic model is complete. The model is emitted as an XML file in order to make it available for further static analysis. Examples are provided in the next chapter.

Just as we used the visitor design pattern to walk parse trees, we use another type of visitor to walk the semantic model. The `ModelVisitor` hierarchy (Figure 104) is designed to allow visitors to be developed for arbitrary purposes, including metrics calculations, without requiring modifications to the model itself.

Figure 104: `ModelVisitor` hierarchy

`ModelVisitor` is an interface with a signature for visiting each class in the model, including abstract classes. (This is a variation on the usual decorator pattern that simplifies some metrics and other processing.) `CompositionVisitor` is an implementation that includes code to navigate through the entire composition hierarchy of the model, but otherwise does nothing; this means subclasses can omit navigation code unless they need to take some alternate route. `XMLWriter` is a concrete visitor that translates the model into XML.

5.8 Discussion

5.8.1 Strengths

JST models a complete Java program with high fidelity to the language specification, and exposes it for further processing by other static analysis tools. This is a significant change from conventional tool-building, in which each different tool performed its own ad hoc extraction of semantic information customised for the purpose of the tool. Compilers are an example: they must acquire all of the same information found by JST, but do so only for the context of compiling and do not reveal a model.

In the absence of comprehensive semantic models, some metrics (and other) research tools have relied on simplifying assumptions (often unstated in the literature) such as having unique names in a whole program. In real programs such assumptions do not hold and research tools are often unsuitable for examining software outside laboratories. JST addresses this difficulty by providing a comprehensive model that captures all the semantic entities and relationships of the Java language without imposing restrictions beyond those of the language itself.

JST is distinguished from other semantic models by being based on a parser that conforms to the Java *exposition* grammar. The semantics of Java are defined in terms of this grammar, and we are consequently able to derive a semantic model directly from the language definition and implement it without having to translate between alternative syntaxes. JST records the relationships between semantic and syntactic structures, producing a model that makes semantic-syntactic connections clear while maintaining rigorous separation of concerns.

Clear separation between syntactic and semantic models is another distinguishing characteristic of JST. A more conventional approach is to annotate parse trees with semantic information [102], resulting in a tightly coupled model in which semantic entities cannot be unbundled from syntax. Some systems, such as the Eclipse JDE described in Chapter 2, have evolved into hybrid syntactic-semantic systems that carry the legacy of this approach and exhibit a less clear structure and even redundant ways of storing and accessing information.

Eclipse's semantic model is also an example of a system subject to requirements other than just modelling software structure—it also provides for the needs of coordinated IDE tools—and does not possess the singularity of purpose of our modelling approach. Many other systems capable of representing software concepts have been developed with agendas other than faithfully modelling software structure to enable further static analysis, and consequently they are less well suited to our purposes.

Java's reflection API is probably the most widely-used model of type system objects. Our model can be examined, much like Java's reflection API, by tools requiring information about the structure of Java programs. Our model exhibits higher resolution (as can be seen by comparing the number of classes in Figure 8 and Figure 95) and is more comprehensive than that of reflection. It resolves overloaded method invocations, includes information about the internal structure of methods, shows uses of variables, and relates the semantic structure to the syntactic structure. Further, the model is available as an XML file, making it accessible to other forms of processing, including XSLT.

JST is still a research tool, but, in combination with yakyacc-generated parsers, it has received significant use in a number of related research projects by several Software Engineering and Visualisation Group (SEVG) members. In our CodeRank work [77] we tested JST on a corpus of 345,000 lines of open source Java code. In Carl Cook's work [19], JST was used as the repository of a collaborative IDE that used very frequent re-parsing of source and extracted semantic information to be fed back to developers in real time. More examples are presented in the next chapter.

Memory usage requirements of JST have so far proven manageable. The size of the semantic model itself is small compared to the size of parse trees. Our approach, however, retains all parse trees for a program in memory (and does not attempt to be miserly with their storage); this is a concern for large code bases. Nevertheless, usage to date has not encountered problems.

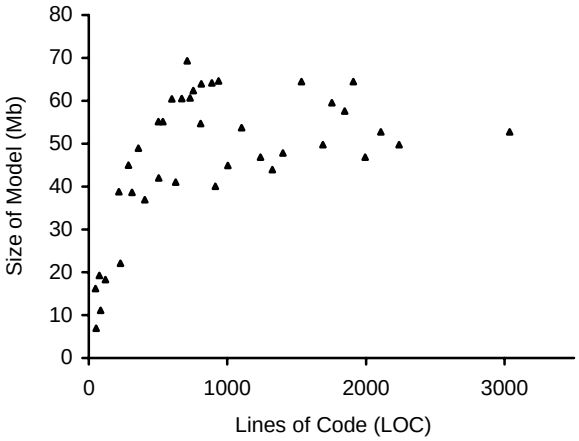

Figure 105: JST memory usage

Cook recorded the memory footprint of his IDE—which is determined largely by the size of the JST model and its parse trees—for a range of small projects. We reproduce his results in Figure 105, which suggests that memory growth will scale without incurring disproportionate consequences.

5.8.2 Weaknesses and limitations

Our programs have not yet been sufficiently tested to be considered industrial strength. JST (in the version described here) requires complete, error-free code, and all library classes that are reachable (transitively) from the source code must be available in the classpath. If errors do occur they are handled inelegantly. Error messages are few and terse. This is not a concern for our current pipeline applications, which analyse only code that is known to compile, but future applications may be more demanding.

In some places the code would benefit from refactoring to remove the legacy of earlier design decisions. In particular, JST still uses an early parse tree implementation and its own parse tree visitors, rather than re-using the better ones now provided by yakyacc; this is a mechanical (but tedious) change.

JST does not currently prune ambiguous reductions from the parse tree; they are just ignored. This was initially a deliberate decision to keep full information, and even allow metrics that measure the level of ambiguity in code. However, the presence of parse tree nodes that are not semantically meaningful can be misleading when calculating other syntactic metrics.

JST implements reciprocal relationships whenever it needs them to perform look-ups, but in some other cases relationships remain directional. This is not a problem when calculating metrics using XSLT, for example, because the XSLT processor automatically builds reverse indexes when needed. In order for JST to serve as an API that supports arbitrary traversals of the model, however, reciprocal relationships (and their getters) should be available. This is a simple change.

JST generates names for un-named elements of the model such as anonymous inner classes. So too does the Java compiler, but it uses different names. This will become a problem if library code for which we have no source references code for which we have source: lookups

won't find the names generated by the Java compiler. The solution is to have JST emulate the name generation of the Java compiler.

As we have remarked, JST uses a monolithic approach that might cause problems with large amounts of code. Large XML files can be greatly compressed because of the redundancy of mark-up, but reducing the memory usage of the program requires a code change. This could be achieved by adding a layer of indirection between the semantic model and parse trees, allowing trees to be unloaded when not in use.

5.8.3 Extensions

We have mentioned several projects that extend JST in various ways. In [21] we drop the requirement that source code be complete. This is achieved by modelling looked-up relationships between semantic concepts using a reference object, rather than just a pointer. This allows a reference to be in an unresolved state, but store the name for so that it may be checked later.

Reference objects could also be used to simplify the task of building the semantic model. Rather than requiring multiple passes over parse trees in order to ensure objects are declared before being looked up, we could declare everything with broken references and then resolve them.

JST is Java-specific, and currently conforms to version 1.3. We have, however, used JST as the basis of other projects, including developing a similar model for .NET [76] and extending the JST model to include Java 1.5 semantics, as well as providing a mapping between JST and .NET [46].

Chapter 6

Measuring and visualising Java programs

The software described in previous chapters produces a model that exposes the syntactic and semantic structure of Java programs, in a form suitable for further processing by software tools. In this chapter we describe examples of tools that make use of the model, with emphasis on the role the model plays in underpinning research into software metrics and visualisations. The work documented in this chapter was undertaken by several members of the SEVG, in collaboration with the author. Many of the figures shown here are reproductions or variations of figures in SEVG publications. However, all of this work relies on the underlying model technology.

The information in JST has many potential applications that may help software engineers understand and improve their designs. For example, it may be used to:

- Calculate software metrics.
- Construct software visualisations.
- Enable auditing of software to ensure it complies with code standards, design policies and heuristics.
- Support translation of source code into UML diagrams and other alternative representations.

- Support collaborative software engineering environments by identifying software neighbourhoods and characterising their proximity.

- Track software evolution by recording different versions of the model over time.

- Direct software testing efforts by identifying regions of greatest interest.

- Assist with project management by characterising the size and topology of software components.

- Underpin further static analysis, such as flow graphs and code reachability analysis, by supplying the base information from which these are derived.

- Underpin dynamic analysis such as performance measurement and memory profiling by providing a static structural framework to which dynamic information may be attached.

The main thrust of this thesis is the acquisition and representation of static software structure information to facilitate applications such as these. The above list is far from comprehensive, and can be expected to grow as the discipline of software engineering increasingly emphasises maintenance, evolution and refactoring of existing code rather than "big up-front design" approaches.

Many software tools of the kinds listed above already exist, with a correspondingly diverse range of techniques for acquiring syntactic and semantic information. Our approach confers advantages by elevating the importance of a software model as a fundamental component of this family of tools and isolating it from its applications so that it is general-purpose and reusable. Our mode is constructed with improved rigour, including by decoupling syntactic and semantic model construction.

To support a broad range of potential tools, the model must be rigorous so that the data can be reliably interpreted in diverse contexts, comprehensive so that it caters to various needs, and flexible so that it can be accessed and applied in different ways. As we have noted, our approach improves rigour by conforming to the JLS exposition grammar and its associated semantic description. The complete set of Java semantic concepts and relationships is cap-

tured, ensuring the model is comprehensive. The model may be accessed directly through its API, or via an XML representation that externalises the information, allowing unconstrained manipulation of the data. Both interfaces have strengths in different areas; the user is free to choose the most suitable alternative for a particular task.

The SEVG research group has developed a number of experimental applications that make use of our model and models derived from it. These include original metrics and visualisations, auditing of software using design heuristics, translating source code into UML class diagrams, providing a repository for a collaborative IDE, and others. The success of these research projects provides evidence of the efficacy of our modelling approach in a range of roles. The full set of developed applications is too extensive to cover adequately here. In this chapter we describe our approach for deriving software metrics and visualisations.

Although we have created a number of new metrics and visualisations as part of this research, our main goal here is not to evaluate or promote any specific metrics or visualisations, but instead to provide evidence of the suitability of our framework for deriving and presenting software structure information. We suggest that many valuable metrics and visualisations have yet to be developed. By enabling arbitrary new metrics and visualisations to be developed without also requiring custom data acquisition tools to be developed, we hope to encourage experimentation with, and ultimately adoption of, better metrics and visualisations.

All of the metrics and visualisations work described here was undertaken collaboratively with Dr. Neville Churcher, and credit for the visualisation development in particular is largely due to him.

6.1 The role of metrics and visualisations

As we remarked earlier, programs are routinely of such size and complexity that they cannot be understood in their entirety. This difficulty is compounded by incessant change as software is developed and maintained. Even so, every part of a program must be constructed with precision and exacting attention to detail.

In order to make progress on overwhelmingly complex programs, software developers must be able to concentrate their attention on characteristics salient to some problem under consideration, and suppress inconsequential details. This is a challenging task because in any non-trivial software design a multitude of forces are present, and these simultaneously influence and are influenced by the designer's judgement of which software features are pertinent. In a single design problem, a designer may have to balance diverse influences such as architectural constraints (for example, keeping a system structured in layers), hiding information, minimising coupling, conforming to design patterns, modelling domain concepts, ensuring adequate performance, communicating the intent of the design to human readers, and many more. This is an extreme case of the *focus+context* information visualisation challenge that arises when observers need to see some region at a high level of detail and the environs at a lower resolution, yet retain the relationship between them.

To solve design problems of this nature, a designer must form rich mental models of software structure and the forces operating on that structure, while abstracting, approximating or eliminating most of the system from consideration. In conventional programming practice this task is often performed with source code as the sole input to the design process. The manifest detail and innately linear organisation of source code oblige the designer to select, filter, cluster and abstract information in order to synthesise a suitable mental model.

Some tools that help designers filter and assimilate relevant information do exist. For example, code browsers aid navigation around source code and support searching for features such as variable usages. UML diagrams provide alternative perspectives that emphasise different aspects of designs, such as class inheritance structure or object interaction. Javadoc provides another view. While these tools are very valuable, they are not sufficient to eliminate the problems of software complexity and information overload.

Software metrics have long been advocated as a means of distilling noteworthy observations from a morass of code. Metrics are an important branch of software engineering with an extensive literature. Textbooks such as [31], [18] and [109] provide an overview. Our interests lie in the sub-field of object-oriented metrics [42] and in particular with static software structure metrics [9] [68].

Although some software metrics have made inroads in the domain of software development processes and project management, particularly for tracking program and component size, they have not in general become an integral part of software design activities. One obstacle to their wider use has been the difficulty of acquiring correct, complete and self-consistent data from which they may be calculated; earlier chapters describe our efforts to address this. A further obstacle is the inherent complexity of design with its multitude of non-orthogonal dimensions. No single metric can capture all facets of a design problem. Indeed, some dimensions of design such as the degree to which it models the problem domain are beyond the reach of automatable metrics, while still others have, as yet, no suitable metrics defined. Even when appropriate metrics can be calculated, inappropriate communication of the metrics may merely compound information overload problems.

Despite these concerns, we maintain that metrics can fulfil a valuable role in informing software designers. To attain their potential, we need a framework that allows calculation of diverse metrics targeted toward specific features of interest in design problems, and efficient means of communicating the results. Just as a set of relevant features in a design problem will span a range of abstractions and localities, metrics should capture information at a variety of levels of granularity and precision, in specific software neighbourhoods—where a neighbourhood is defined by the proximity of semantic elements.

At one extreme, metrics can usefully quantify concrete, local aspects of code such as Lines of Code (LOC), NPATH [78], or number of parameters. Metrics of this kind have received most attention in the literature. At the other extreme are metrics that capture inherently fuzzier, more holistic and more ambient characteristics of the software. Such metrics can help a designer answer questions such as: Which parts of the software are relevant to the current problem? Which features of the design are most central? Is the design growing too complex? How heavily coupled are its components? Which parts of the design are most in need of restructuring? Questions such as these cannot be answered by any single metric, but carefully chosen families of metrics, communicated unobtrusively, might allow a designer to reach a judgement more efficiently than by just reading code.

Large tables of metrics—even if they contain potentially valuable observations—are an ineffective means of communicating results or resolving software designers' information overload problem. Tables do not make trends and relationships explicit. The field of *information*

visualisation tackles the problems of communicating large, complex data sets. Information visualisation is a substantial research field—[7] and [97] provide an overview—and one that is encountering new opportunities as processing power and display technology continue to improve.

Conventional graphs and charts (histograms, line graphs, Kiviat charts, etc), are established visualisation techniques that can play an important role in communicating software metrics information, just as they do in other information rich domains. However, software raises new challenges for conventional visualisation techniques:

- It is difficult or impossible to capture in 2D graphs or charts the volumes of information and the many dimensions of interest.

- It is harder still to accommodate the mercurial changes in perspective as a designer considers various aspects of a program. The use of different views and metaphors can convey a variety of perspectives, but introduces the need for smooth transitions between them.

- Software metrics often exhibit extremely nonlinear distributions and extreme outliers. Adjusting linear scales to accommodate the full range of values can suppress virtually all information in the graph except the extremes.

- Similarly, software metrics distributions are often heavily skewed and spread across large ranges, so that the density of data to be portrayed is inconsistent across different regions of a visualisation.

- Software designers need information at very diverse levels of abstraction, from architectural structures to code details: the *focus+context* problem.

- Conventional information displays are often optimised for communicating precise, undistorted detail of some isolated aspect of a system, rather than the more holistic, interconnected and faceted issues of importance to software designers.

- Software exists in a domain of pure information that lacks physical underpinnings. This means that it does not have any inherent geometry around which visualisations may be structured, in the way that scientific visualisations do.

A further obstacle to communication encountered when using conventional display techniques is the problem of disassociation of metrics from the underlying structures that the metrics describe. Information gleaned from a line graph, for example, needs to be integrated into a designer's mental model by associating graph data with model features. The graph representation may provide few cues about how to achieve this. Our work explores the possibility of reducing this problem by integrating metrics into software structure visualisations. We first apply some visual metaphor that depicts software structure, giving us a framework to which metrics information can be added as adornments of various types.

The most familiar and immediate representation of software structure is source code itself. We can visualise many metrics by decorating source code in a variety of ways. For example, we might indicate the age of a segment of code (since it was last edited) using colour, perhaps by yellowing the background progressively as the segment ages. In this way newly edited code appears brighter and is more likely to draw the attention of a developer—a boon when tracking down recently introduced bugs. Figure 106 shows a screen-shot of a text editor augmented with code age line colouring. This editor is part of a collaborative IDE based on our JST model [21]. The IDE makes use of multiple versions of parse trees so that age of any section can be calculated.

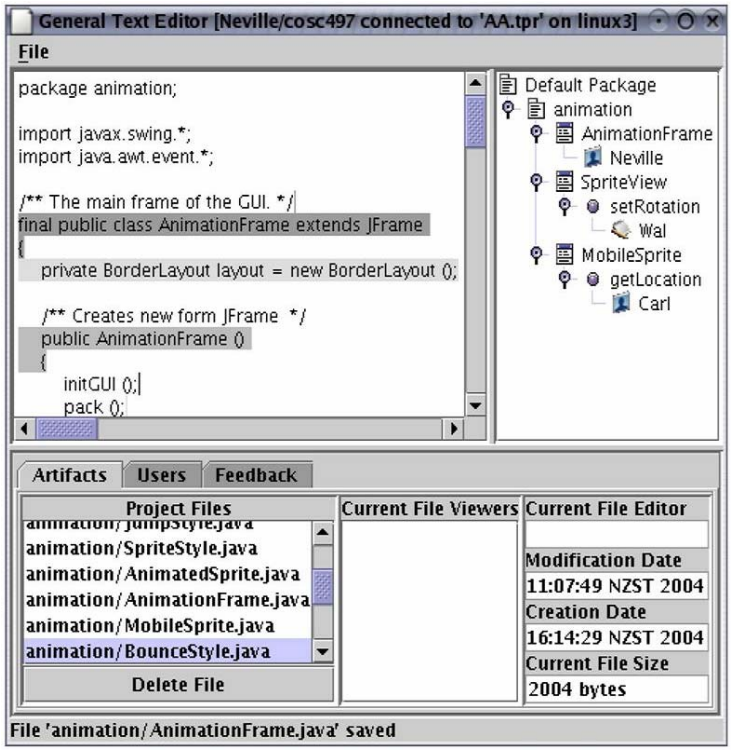

Figure 106: Code age editor

Many similar visualisations can be achieved by colouring lines of code according to a metric. In [13], we describe

SeeSoftLike, an experimental visualisation that decorates code with metrics including code age. SeeSoftLike follows the approach of Eick et al. [30] by allowing code to be shown in a miniscule font so that the observer can gain a broader perspective at the cost of diminished detail. Figure 107 shows a screen-shot, in which lines of code are coloured to indicate their author (the programmer who most recently edited them). 'Showing the tracks' of authors in this way is a new and potentially very helpful perspective in a collaborative project. It requires our model to be supplemented by author information associated with parse tree nodes. This extra information is recorded by the IDE.

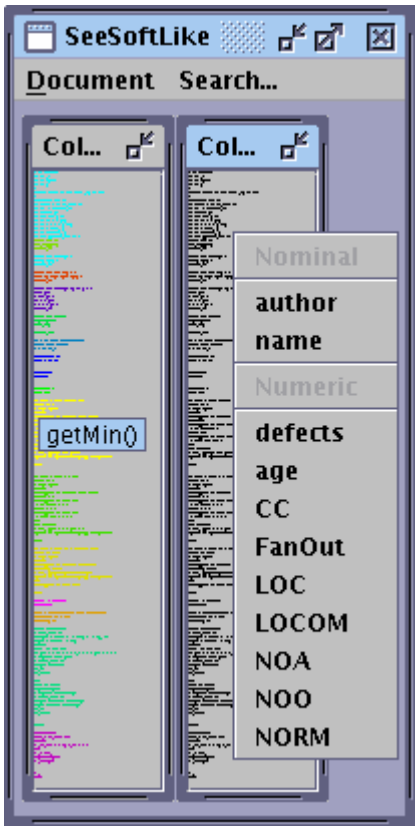

Figure 107: SeeSoftLike

The file supplied to SeeSoftLike contains (groups of) lines of code, their associated metric values, and metadata describing the metrics. The range of metrics available in the example file is evident in the pop-up dialog on the right of the figure. Most of these are conventional size, complexity or coupling metrics that can be calculated from data in our model. Supplementary information is required for the code age and author visualisations discussed above, and for the defects visualisation, which shows the number of defects that have been found in a region of code and thereby suggests to programmers the level of care appropriate when editing that code.

SeeSoftLike supports side-by-side displays of different metrics so that they may be compared and perceived in concert. Users may configure the criteria used to map metrics to colours, in order to colour only regions with more than ten defects, for example. To assist with browsing code, a simple focus plus context technique is used: the cursor may be hovered over a line to elicit a 'tool tip' naming that code section (as shown for the getMin() method in Figure 107). Full code details can be revealed by expanding the font to a legible size.

Many more useful metrics might be grafted onto source code displays. For instance, we might highlight method invocations and variable uses, adjusting the intensity of highlighting to show the degree of coupling to the component being accessed. Information hiding [85]

might be emphasised by decorating attributes and methods according to their access levels, breadth of scope, or number of places in which they are actually used. A developer might switch between visualisations (or combinations of them) as different design forcers were considered.

The use of tiny fonts in `SeeSoftLike` allows metrics that convey detailed, concrete aspects of code to be perceived on a broader scale and to work in a more approximate, holistic way that they otherwise would. The user can form an overall impression of the amount of collaboration in authorship of a module or the distribution of defects across a module. Side-by-side views of these two metrics could suggest to the observer relationships between authors and defect rates.

An extension of this code-centric visualisation approach would display not just measures of program features, but would flag problems suggested by design heuristics, code smells, local design policies and other constraints derived from metrics and observations of the model. One possible visual presentation of these heuristic checks would be similar to MS Word's grammar checker, which uses a green underline to indicate possible problems, and a pop-up dialog box to provide details. Although we have not (yet) created this user interface, we have developed a tool that detects heuristics violations based on metrics from a semantic model (derived from JST) [12]. It audits the model, checking for cyclic dependencies between packages [69], inheritance hierarchies that have grown too deep, overlarge classes, and similar warning signs [93]. A recent study shows other authors [73] are experimenting with exactly this style of tool using semantic model data acquired from Eclipse's JDT (which we discussed in Chapter2), with encouraging results.

An alternative representation of software structure that is familiar to many software developers is UML, and this too offers a promising substrate for visualising metrics. We might highlight cyclic dependencies on a package diagram, for example. In a class diagram, we might make inheritance relationship lines thicker to represent the number of subclasses, colour methods to show their size, and so on. An example of embellishing a class diagram with metrics—by making the thickness of class borders proportional to `ClassRank`—is provided in [77]. (`ClassRank` will be described in Section 6.3.1.)

We have not yet developed these UML-centric visualisation ideas beyond simple prototypes, although we foresee no great difficulty in using our model in this way and expect to do so in the future. We have chosen initially to investigate more experimental visualisations using 3D virtual worlds, as they escape the prescriptive structures of source code and UML notation and allow us to construct frameworks using semantic concepts that, we hope, can more closely match designer's mental models.

It is an open question whether the advantages of more abstract semantic visualisations outweigh the problems introduced by using some unfamiliar alternative representation of programs. Experienced software developers are adept at mapping source code and UML to mental models despite the difficulty of the task, and are not experienced with 3D representations of software. Indeed, designers' current mental models are likely to be heavily influenced by current software representations including source code and UML, and so new alternatives are at a disadvantage.

Similarly, the relative merits of 3D visualisations of software over more traditional 2D ones are a matter for more research. Three dimensional visualisations introduce issues of occlusion and navigation beyond those encountered in two dimensional representations. It is, however, conceivable that occlusion and 3D perspective can sometimes be used to advantage to hide (or shrink) details that are unimportant from some vantage points, or that improvements to fluidity of navigation and manipulation of models will reduce these problems. More fundamentally, the question of what software looks like has not been finally answered by source code and UML. We don't try to answer these questions here, but instead seek to show that our modelling approach provides a viable basis for experimentation with metrics and visualisations.

6.2 Pipeline architecture

We use a pipeline architecture [6] for transforming source code into metrics and visualisations, as shown in Figure 108. A pipeline is a series of *filters* that each read data from an input file (or data stream) and write data to an output file (or stream). The initial input to the pipeline is source code and the final output is a visualisation file in some format suitable for rendering, typically VRML [8] in our work. All intermediate files throughout the pipeline use XML.

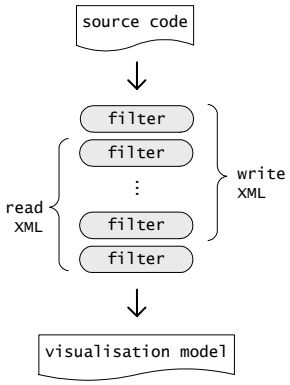

Figure 108: Pipeline input and output

The production of software visualisations is, in general, an intricate process involving many intermediate stages, each with a different role and employing different data structures and algorithms. A pipeline is well suited to this problem [94]. Our approach contributes the use of XML as the medium for data representation, and provides features specifically targeted at software visualisation, including our static analysis technology.

Artefacts employed in our pipeline include:

- Source code to be analysed.
- Scanners that recognise tokens in source code.
- Grammars for specifying the syntax of source code and the corresponding structure of parse trees.
- Parse trees that describe the syntactic structure extracted from sequences of tokens.
- Automata that describe a mechanism capable of parsing a given language.
- Generated parsers that execute an automaton to translate source code into parse trees.
- Semantic models that represent semantic entities and relationships extracted from parse trees.

- Metric utilities that produce measurements of parse trees and semantic structures.
- Transformed models that contain filtered, clustered, or derived forms of the raw models. A single visualisation may involve a variety of transformed models.
- Visualisation models that represent software structure after some visual metaphor has been applied.
- Geometry computation and layout algorithms.
- Mappings that configure visualisations by describing how various features are to appear.
- Visualisation data such as VRML files that describe the resulting visualisation.

Exploratory development of new visualisations requires creative involvement of a user to specify what information is relevant, how it should be transformed, and how it should appear. Re-processing of some stages is often necessary as decisions are revisited, tools calibrated and variations tried. Further, different parts of the process may be the responsibility of different developers, and may use different programming languages or technologies. This situation calls for a flexible approach that decouples the stages of visualisation development and provides opportunities for the user to evaluate intermediate and final results and to reconfigure components as necessary.

A pipeline affords a high degree of flexibility because filters are coupled only by external file formats, allowing the filters to be independently developed, employ differing technologies, and be composed in diverse ways. Pipelines may branch, merge and contain cycles. This flexibility is particularly helpful for experimenting with metrics and visualisations, as discoveries gained from the pipeline can influence its subsequent development.

Figure 109 presents a typical pipeline, and Figure 110 shows excerpts of XML files produced in the data acquisition portion of the pipeline. (We defer fuller discussion of the metrics and visualisation portion to Sections 6.3 and 6.4 of this chapter.) The steps in the example are:

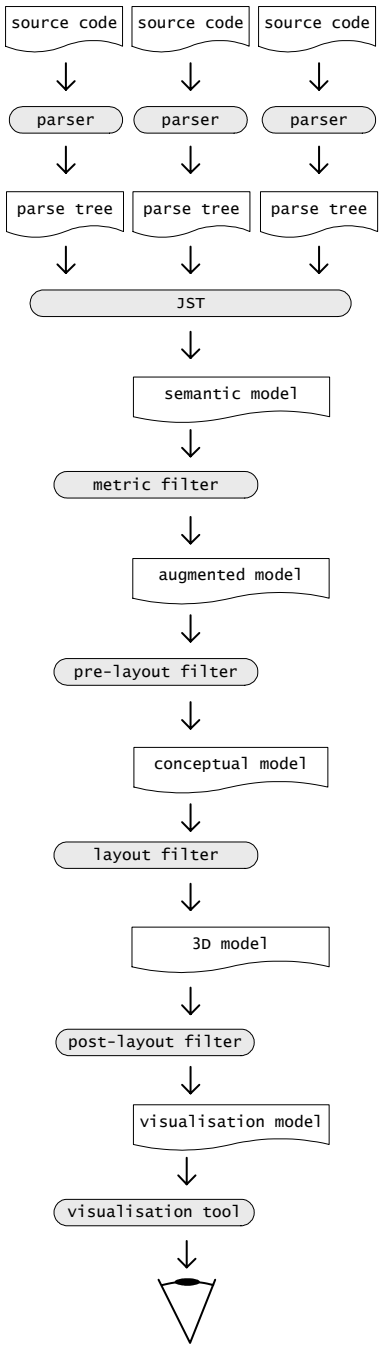

Figure 109: Pipeline example

- A *parser* transforms source code by adding XML tags that describe the parse tree. The original source text is retained between the tags. Figure 110 (a) shows a fragment of the Java grammar and (b) gives example code corresponding to that part of the grammar. Part (c) of the figure shows the consequent parse tree fragment. The tags in the parse tree XML originate from symbols in the grammar, while the text originates from the source code.

- *JST* transforms a set of individual parse trees into an integrated semantic model. The XML output file includes the original parse trees and a separate XML sub-tree that describes the semantic concepts and links them to each other and the parse trees. Figure 110 (d) shows a fragment derived from our example parse tree. Parse tree nodes now contain identifiers so they may be referenced from the semantic model.

- Any number of *metrics filters* may be used to augment the model with tags describing observations. As we remarked earlier, these filters may access the model as an XML file (perhaps via XSLT), or may load the model into memory and use its API (including visitors) if that is more convenient; different filters will have different needs.

- The remaining steps transform the augmented model into a visualisation. A *pre-layout filter* selects the concepts, relationships and attributes to be visualised. In the

example, the filter selects classes and methods as nodes to be visualised, and class-contains-method and method-invokes-method as relationships. The filter outputs NGML, an XML format for describing graphs. Fig Figure 110 (a) shows the format.

- A *layout filter* computes the geometry of the visualisation. For 3D visualisations we typically use ANGLE, an inhomogeneous force-directed layout engine developed by the SEVG research group [14].

- A *post-layout filter* maps features of the 3D model to visual forms. For example, method nodes are ascribed shape, colour, size and transparency. Post-layout filters are concerned only with the appearance of visualisations; they decorate a structure

```
// 19.8.3) Method Declarations

<method_declaration> ::=
        <method_header> <method_body>
    ;

<method_header> ::=
        <method_modifiers>? <result_type> <method_declarator> <throws>?
    ;

<result_type> ::=
        <type>
    |   VOID
    ;

<method_declarator> ::=
        IDENTIFIER LPAREN <formal_parameter_list>? RPAREN
    |   <method_declarator> LBRACK RBRACK
    ;
```

(a) Grammar

```
public abstract class TypedDecl extends Decl {

    protected TypeDecl type;

    public TypedDecl(Scope theOwner,
                     String theSimpleName,
                     Nonterminal theSource) {
        super(theOwner,
              theSimpleName,
              theSource);
    }
    // ...
}
```

(b) Source code

```
<method_declaration>
 <method_header>
  <method_modifiers>
   <method_modifier>
    <token id='PUBLIC'>public</token>
   </method_modifier>
  </method_modifiers>
  <result_type>
   <reference_type>
    <class_or_interface_type>
     <class_type>
      <type_name>
       <token id='IDENTIFIER'>TypeDecl</token>
      </type_name>
     </class_type>
    </class_or_interface_type>
   </reference_type>
  </result_type>
  <method_declarator>
   <token id='IDENTIFIER'>getType</token>
   <token id='LPAREN'>(</token>
   <token id='RPAREN'>)</token>
  </method_declarator>
 </method_header>
<method_body>
...
```

(c) Parse tree

```
...
<method id='MTH_jst.symtab.TypedDecl.getType()' name='getType()'
        source='NTL_53719'
        type='TYP_jst.symtab.TypeDecl' modifier='public'>
 <block id='BLK_jst.symtab.TypedDecl.getType().@BODY'>
  <reference to='FLD_jst.symtab.TypedDecl.type' from='NTL_53694'/>
 </block>
</method>
...
<nonterminal id='NTL_53719' type='method_declaration'>
 <nonterminal id='NTL_53691' type='method_header'>
  <nonterminal id='NTL_53682' type='method_modifiers'>
   <nonterminal id='NTL_53680' type='method_modifier'>
    <terminal type='PUBLIC'>public</terminal>
   </nonterminal>
  </nonterminal>
  <nonterminal id='NTL_53687' type='result_type'>
   <nonterminal id='NTL_53686' type='reference_type'>
    <nonterminal id='NTL_53685' type='class_or_interface_type'>
     <nonterminal id='NTL_53684' type='class_type'>
      <nonterminal id='NTL_53683' type='type_name'>
       <terminal type='IDENTIFIER'>TypeDecl</terminal>
      </nonterminal>
     </nonterminal>
    </nonterminal>
   </nonterminal>
  </nonterminal>
  <nonterminal id='NTL_53689' type='method_declarator'>
   <terminal type='IDENTIFIER'>getType</terminal>
   <terminal type='LPAREN'>(</terminal>
   <terminal type='RPAREN'>)</terminal>
  </nonterminal>
 </nonterminal>
 <nonterminal id='NTL_53718' type='method_body'>
...
```

(d) JST model

Figure 110: Pipeline XML files

built by earlier filters. This separation of concerns affords a high degree of flexibility, allowing easy customisation of visualisations' appearance.

The use of XML in the pipeline offers several advantages over approaches that depend on monolithic tools or less transparent data formats [50]. XML is text based and easily read. The files are self-describing because they contain their own metadata, and so encapsulate all information that couples filter programs in one place.

Transformation of XML is well supported by existing tools, including XSLT. XSLT applies a *stylesheet* to an XML file, transforming the data into a new format—usually, but not necessarily, another XML file. Stylesheet-driven transformations are a powerful mechanism for obtaining configurable filters in the pipeline, particularly for pre-layout and post-layout filters and many metrics calculations. In other filters, where general-purpose programming languages have advantages over XSLT, support for reading and writing XML files is widely available. We typically use DOM and SAX [41].

6.3 Metrics calculation

Many metrics can be derived directly from information extracted from a JST model. Simple counts such as number of classes, number of methods for each class, and number of parameters for each method, are common examples. These metrics use the same containment relationships found in code (classes contain methods, which contain parameters), but other relationships in the model are equally valid subjects for metrics, yielding measures such as number of supertypes (classes and interfaces) and subtypes, number of declarations that use each type, number of invocations of each method, fan-in and fan-out of methods, and so on. Following chains of relationships allows us to accumulate transitive measures such as depth of classes in the inheritance tree, number of inherited attributes, number of methods invoked indirectly, etc.

The metrics mentioned above are all defined in terms of semantic concepts, but syntactic and even lexical metrics are also supported. Parse trees are retained in the JST model, with parse tree nodes linked to the semantic objects they describe. In turn, parse trees contain tokens, which are chained together so that the original program text—including whitespace—is re-

trievable. Metrics filters may traverse the connected data structures to calculate, for instance, lines of code (a lexical metric) or cyclomatic complexity (syntactic) for each method.

Because complete parse trees are stored, every object in the semantic model is connected to the syntactic representation from which it was derived. For example, a semantic object representing a class is linked to its class declaration in a parse tree. We might therefore measure some characteristics of a program by defining either a semantic or a syntactic variant of a metric. Number of classes might be defined as a count of class objects or of class declarations. In general, the semantic version is preferable as it is defined at a higher level of abstraction (independent of syntactic details) and participates in semantic relationships that are not evident in parse trees. In the example of counting classes, a syntactic metric might fail to detect anonymous inner classes that are declared implicitly within constructor invocations, whereas the semantic alternative would detect all classes regardless of their declaration syntax.

The lowest level unit of a program represented in our semantic model is a block (statements within a pair of braces). Blocks are part of the semantic model because they define scopes, and so are necessary for resolving name look-ups. Blocks contain statements, which are not modelled as JST objects because they do not define entities that may be referenced in any way other than that already captured by the syntactic structure. Because statements are represented only in parse trees, metrics involving statement features (cyclomatic complexity, for example) must be defined syntactically; that is, in terms of parse trees. Expressions, which usually occur within statements, are a similar case, except that whenever any expression uses a semantic concept a semantic relationship is recorded from the containing semantic entity (usually a block) to the referenced semantic model object. Method invocations and variable accesses are examples. Each relationship is associated with the parse tree node of the expression that produced the reference.

Precise definition of metrics is necessary if results are to be interpreted correctly [17]. Even straightforward metrics such as the examples above require elaboration. When counting methods, for example, we must answer questions such as:

- Are constructors counted? If so, are compiler-generated default constructors also included?

- Are private, protected and/or package methods counted? What about static methods and abstract methods?

- Do overloaded methods count singly or multiply?

- Are inherited methods included? If so, are private methods considered as inherited? Should overridden methods count multiply or singly? Are method signatures in interfaces counted separately from their implementations? How should a single method signature inherited from multiple interfaces be counted?

Clearly, answering these questions in different ways could lead to very different metric values, which in the absence of a precise metric definition, would allow few safe conclusions to be drawn. Unfortunately, a lack of disclosure of metrics definitions and implementation techniques has been a hallmark of much metrics literature and tools.

We contend that one reason for this deficit has been the lack of reference models that supply terms by which metrics may be defined, compounded by the lack of available data consistent with the models. Our approach allows metrics to be defined and calculated in terms of the semantic concepts of the language and its underlying grammar. Questions like those above can be answered using the same semantic and syntactic terms, yielding metrics definitions and data with a degree of precision that has been lacking in many other approaches. The improved rigour of this approach enables reliable interpretation and comparison of results.

The actual mechanics of metrics calculation are unsurprising. XSLT stylesheets [59] can produce many direct measurements of model features. Alternatively, metrics can be calculated by a program that reads a JST model into memory and uses the *visitor* design pattern [35] to walk through the model and accumulate values. Figure 111 shows the number of methods metric (in its simplest form) implemented in Java by subclassing the `ModelVisitor` provided with JST. `ModelVisitor` provides a de-

```
public class NOMVisitor extends CompositionVisitor {

    protected Map counts;

    public NOMVisitor() {
        counts = new HashMap();
    }

    public void visitMethodDecl(MethodDecl methodDecl) {

        Scope owner = methodDecl.getOwnerScope();

        Integer count = (Integer) counts.get(owner);
        if (count == null)
            count = new Integer(1);
        else
            count = new Integer(count.intValue() + 1);

        counts.put(owner, count);
    }

    public Map getResult() {
        return counts;
    }
}
```

Figure 111: Number of methods visitor

fault navigation implementation that follows the containment hierarchy of the Model.

The inheritance structure of the JST model facilitates definition and calculation of many metrics. The hierarchy captures commonalities of concrete classes at successively more abstract levels. For example, `ClassType` and `InterfaceType` share a common superclass, `UserType`. Metrics that measure some feature shared by classes and interfaces—number of methods, for instance—need not distinguish between the concrete types, but can simply work with the `UserType` abstraction. Similarly, metrics can be defined for other abstractions such as scopes, declarations, operations (methods and constructors), variables (local, fields, parameters), and others without concern for their specialisations.

CFG-based parse trees also exhibit a hierarchical nature that aids metrics production. We can, for example, count instances of *statement* nonterminals without regard for the type of statement. We might alternatively count only *assignment*s or *switch_statement*s. This sort of syntactic generalisation, while useful, is less robust than the generalisation represented by object-oriented inheritance because it reflects only syntactic commonalities considered important by the grammar designer, rather than deeper semantic generalisations. In the case of the Java exposition grammar, a simple count of *equality_expression*s, for example, might produce a surprising result because the syntax is defined so that other expression types such as *additive_expression*s are reduced as *equality_expression*s and would also be counted.

6.3.1 CodeRank

So far, the metrics we have discussed require only traversal of a model while counting features. Of course, metrics thus produced might be averaged, aggregated, combined to find ratios, correlated and so on. A broad range of valuable metrics can be calculated in this way. Metrics, however, need not measure only program attributes directly represented in the model, but may depend on further static analysis and derived data structures. We have developed a new family of metrics known as *CodeRank* that demonstrates this more elaborate approach [77][4].

[4] This paper received the best research paper award at ASWEC06.

CodeRank is inspired by the idea behind Google's *pagerank* algorithm [81] and applies it to software structure. Pagerank models the World Wide Web as a graph in which nodes are web pages and edges are hyperlinks, and ascribes to each page a rank derived from the topology of the graph. Each page shares its own rank equally along outgoing edges to target pages. The algorithm propagates rank incrementally and uses a damping factor to ensure it converges on stable values. The result is a ranking of web pages by their importance in the structure of the graph. This information has proven very valuable in the context of internet search engines.

CodeRank takes a similar approach to ranking software components: the components are the graph nodes and their relationships the edges. Rankings are found iteratively. Unlike pagerank, which treats all web pages and all hyperlinks homogeneously, we broaden the concept to support heterogeneous types of nodes and relationships. A heterogeneous graph allows us to model the various component types found in software such as packages, classes, methods and attributes, and permits us to apply different weightings to the various relationship types. In this way we can ascribe greater importance to some relationships. For example, we might weight inheritance relationships more heavily than association relationships.

By choosing the types of nodes and edges that participate in a graph, we can produce a variety of rankings. *PackageRank*, *ClassRank* and *MethodRank* are members of the CodeRank family produced by restricting nodes to packages, classes and methods, respectively. (Or alternatively, by aggregating the ranks of sub-components—see [77] for details.)

Figure 112 is a screenshot of `CodeRanker`, our implementation of CodeRank[5]. The screen shows a tab for configuring ClassRank. The sliders allow weightings of relationship types to be individually adjusted. A checkbox allows method overriding to be incorporated in the calculation, by adding to the graph edges to all methods that override an invoked method. The resulting class ranks appear at the bottom of the screen.

Figure 113 uses parallel coordinates to show the results of calculating ClassRank for several successive versions of an open source Java project (ANTLR). Horizontal lines show

[5] `CodeRanker` was implemented primarily by Blair Neate under the author's supervision.

changes to a class' rank as the code evolves. Many lines show a downward trend, suggesting that classes have been refactored to more evenly distribute functionality around the system.

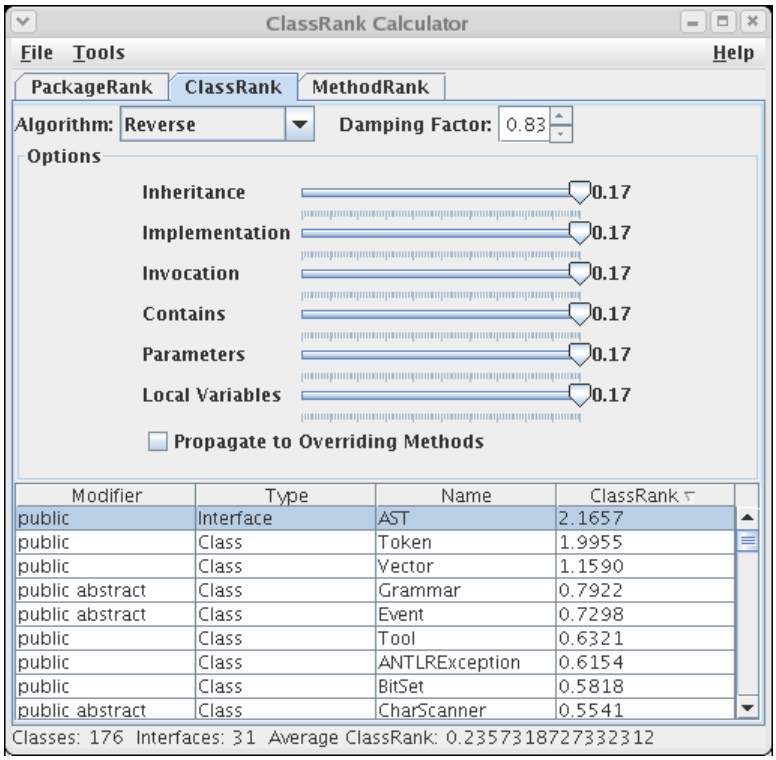

Figure 112: CodeRanker

CodeRank illuminates a dimension of software not shown by existing software metrics. Like its progenitor pagerank, it indicates the relative importance of components within the structure. Software engineers have conventionally been forced to rely on more circumstantial indicators of component importance, such as measures of size (often measured by LOC), complexity or coupling. CodeRank can be used in conjunction with these traditional metrics to better characterise components, so for example we might recognise components as small-but-important, or complex-

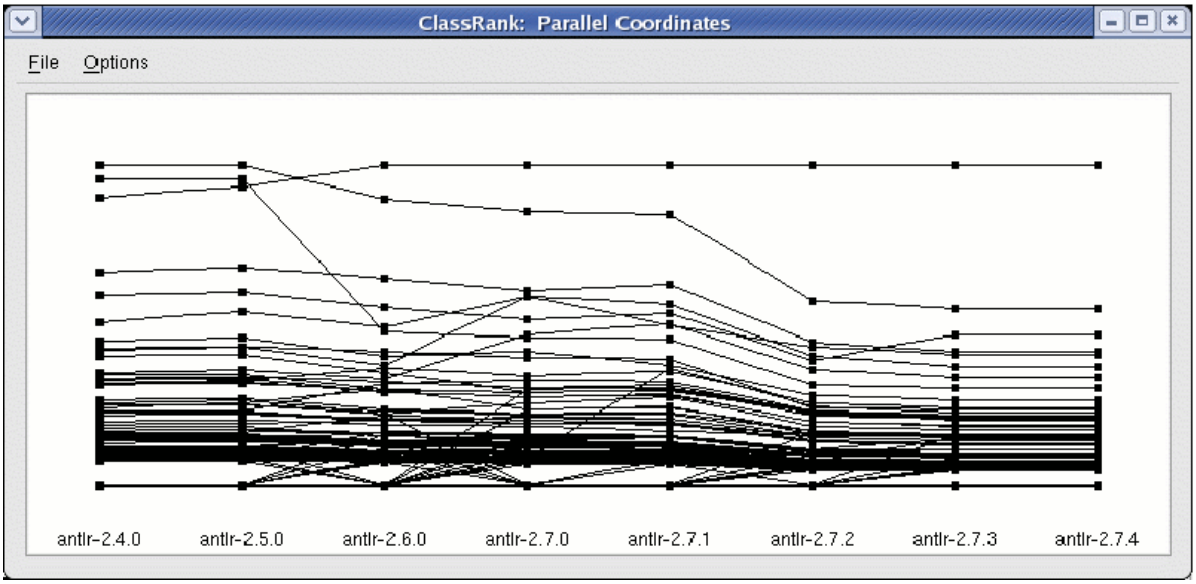

Figure 113: ClassRank parallel coordinates graph

but-less-important, and so on.

Figure 114 plots metrics for the classes in a small program called `Aliens` (a simulator of alien abductions, originally developed as a design patterns teaching resource). Parts (a) and (b) show relationships between conventional metrics: LOC and cyclomatic complexity in the first case, WMC and cyclomatic complexity in the second. The strong correlations with occasional outliers evident in the graphs are characteristic of most software, and indicate that the metrics reflect non-orthogonal dimensions of the program. Parts (c) and (d) of the figure plot our ClassRank metric against cyclomatic complexity and WMC, respectively. The met-

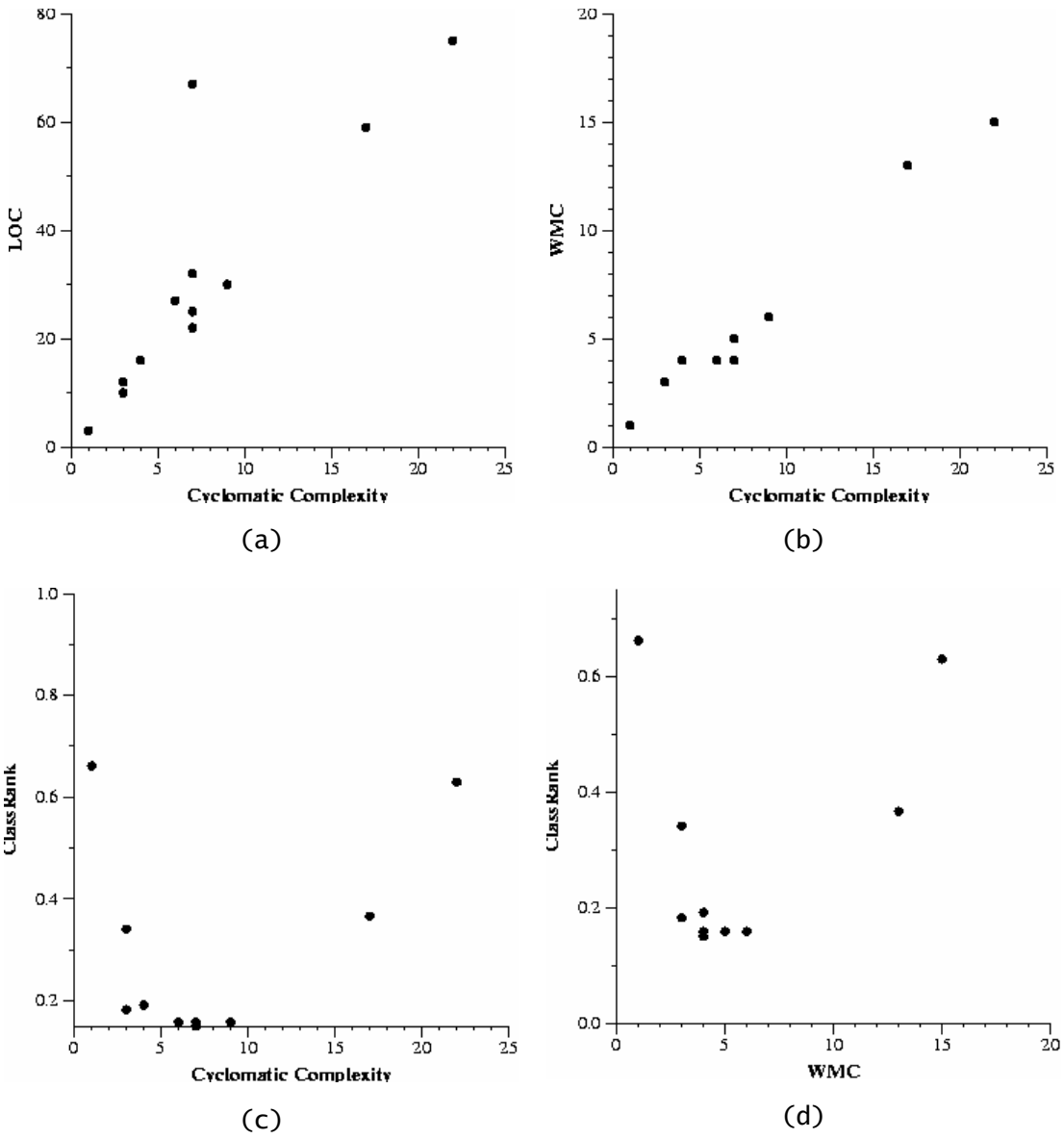

Figure 114: Aliens program metrics

rics are not correlated, indicating that the quality measured by ClassRank is distinct from these other metrics.

These graphs serve a purpose beyond merely showing that ClassRank measures something different. By combining metrics in this way, we can observe program features that might otherwise not be evident. We can distinguish regions of the graphs in (c) and (d) that contain classes with common characteristics. Classes at the top right (`Person` and `UFO`) have high functionality. They are highly ranked because they *do* a lot. Classes at the top left (`Experiment` and `Invader`) are small utilities and high-level abstractions. They are highly ranked because they are *used* a lot. The remaining classes in the graphs are of only modest importance and average size/complexity.

The central regions of the graphs are empty, indicating the absence of highly important classes with only moderate size/complexity. If classes were to appear in this region, they might be considered as candidates for refactoring by identifying missing abstractions.

We expect CodeRank to prove valuable in a range of software engineering situations. Highly ranked classes are those which are most pivotal to the design and provide the most widely used services. They are consequently good candidates for early attention when trying to understand a body of unfamiliar code, and provide important vocabulary for understanding the rest of the design. Similarly, high rank suggests that correspondingly high effort might be directed when maintaining, refactoring or extending code, and when developing unit tests.

CodeRank is also beneficial for understanding actual levels of software reuse. Reuse is a perennial theme of OO software development but has proven difficult to achieve in practice. Consequently, the ability to detect and measure reuse is an important capability. Existing metrics can capture simple, direct forms of reuse [89], but more subtle and indirect cases have remained challenging [106]. CodeRank can be configured to capture varieties of reuse, including reuse through inheritance. The transitive nature of the metric produces a holistic view of system-wide reuse, rather than showing just the proximate causes of reuse.

CodeRank is an example of a metric that cannot be calculated without a rich semantic model such as JST. It relies upon an assortment of semantic relationships, including resolved method invocations, which are not available from tools that lack a full set of features includ-

ing fully scoped name look-ups and the ability to determine the type of expressions used as parameters.

Although the `Aliens` example we have described is a "toy" system, CodeRank has been shown to perform well on a large corpus of real Java software.

6.4 Generating virtual world visualisations

Our approach to visualising software using virtual worlds has been documented in a number of papers [49], [50], [13]. We summarise the main points here (reproducing several figures) and refer the reader to the relevant papers for details.

Off the shelf technology for displaying virtual worlds is readily available. We make use of VRML to describe virtual worlds and web browser plug-ins to display them. Virtual worlds provide an opportunity to display large volumes of multivariate information in a form that allows intuitive perception and exploration across a range of scales [88]. By visualising software structures and metrics with virtual worlds, we hope to provide the viewer with perspectives that show many facets of software in concert, and so to encourage insights that are not prompted by conventional views.

Software lends itself naturally to graph-based visualisations, because the underlying semantic concepts inherently form a graph consisting of heterogeneous nodes. JST models such a graph, although without visual form. Humans are adept at perceiving and manipulating real 3D objects, and with the advent of virtual world technology, the potential of 3D graph-based (and other) software visualisations deserves research.

In the same way that our `SeeSoftLike` 2D visualisations allow metrics to be perceived on a broader, less detailed scale, our 3D virtual worlds can show high-level vistas of metrics data. Our virtual worlds take this idea further by combining a variety of structures and metrics into one view, whereas `SeeSoftLike` shows only one structure—lines of code—and requires separate views for different metrics. The result is a more holistic, integrated view that supports subjective judgements about aspects of software designs such as size, coupling, centrality, complexity, encapsulation and so on. For example, we might devise a visualisation to

convey insights into the balance of competing forces in a system such as the number of components, the size of their interfaces and the complexity of their implementations.

Figure 115 shows a more detailed example of the visualisation section of the pipeline. Two alternative pre-layout filters are shown, driven by different stylesheets. Pre-layout stylesheets specify the content of the visualisation, independent of its appearance. For example, we might produce a graph of classes and methods linked by inheritance, containment and invocation relationships, or a graph of the program scope structure. Features may be selected here for their role in layout, as well as in the final visualisation. For example, we often add relationships to ensure the graph is connected and will therefore be laid out in a contiguous space.

Layout of graphs ascribes an artificial geometry to conceptual entities and positions them in space. Layout of two dimensional graphs has been addressed by other researchers—[25], for example—but 3D layout presents an additional challenge. In Figure 115 layout is performed by ANGLE [11], which uses an original 3D inhomogeneous force-directed approach. A configuration file defines parameters for the layout algorithm, including setting the spring strengths for different relationship types [14].

Finally, Figure 115 shows several post-layout stylesheets being applied in order to map concepts in the 3D model to visual forms with shape, size, colour and orientation. Different post-layout mappings typically emphasise different features of the model or show metrics in different ways.

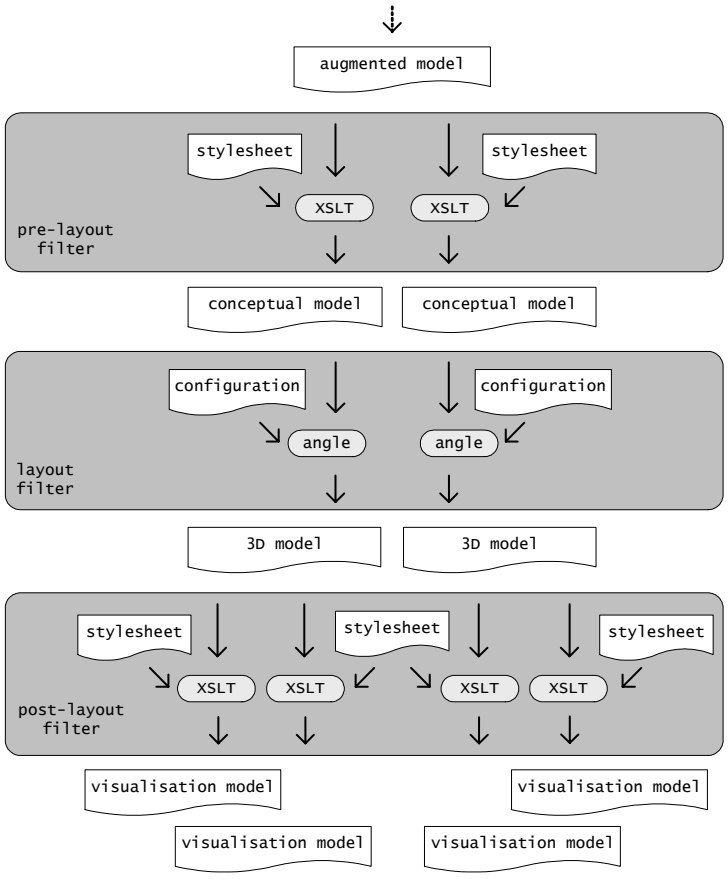

Figure 115: Visualisation filters in the pipeline

Figure 116 uses a simple example to illustrate this process, including the effects of selecting different sets of relationships to participate in a visualisation. The purpose of this type of visualisation is to show class cohesion, in the spirit of the LCOM metric, which is based on the idea that methods should tend to use multiple attributes of their own class. This idea is expressed by Riel as heuristic 4.6: "Most of the methods defined on a class should be using most of the data members most of the time." However, some controversy about the validity of the idea has arisen [42]. Our visualisation approach allows us to explore and clarify the issues.

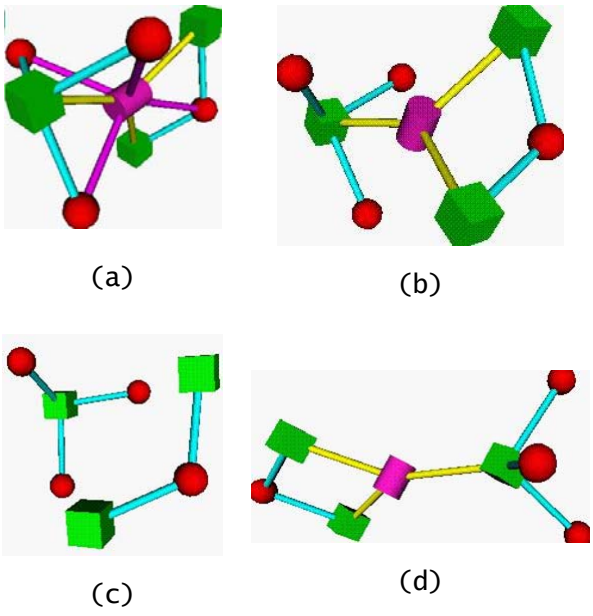

Figure 116: Class cohesion variations

Each of the four parts of the figure shows the same conceptual model containing one class with three attributes and three methods (a purple cylinder, three green cubes and three red spheres, respectively). In parts (a), (b) and (c) the pre-layout filter included all relationships, resulting in the same layout in each. The post-layout filter for (a) made all relationships visible. The post-layout filter for (b) omitted class-contains-method relationships, and for (c) omitted the class and all its relationships. Part (d) is laid out differently, because the pre-layout filter excluded class-contains-method relationships. Consequently methods have moved further from the centre. The post-layout filter for (d) is the same as for (a); that is, showing all relationships.

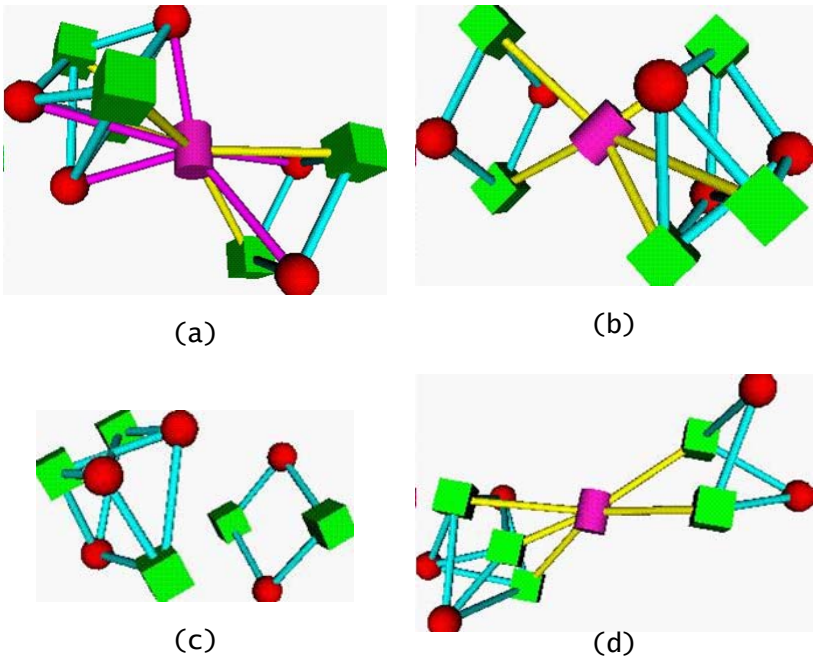

Figure 117: Separable class

In Figure 117 we apply the same sets of filters to another class, this one contrived to exhibit low cohesion, as it might if it should be refactored into two classes. Strategies (c) and (d) in both figures are more promising for revealing low cohesion. Examples of a real class visualised with these two strategies appear in Figure 118. Both variants readily show that the methods tend to cluster around individual attributes, rather than using a substantial fraction of the available attributes. This characteristic proves to be very common, and perhaps raises questions about the assumptions behind LCOM and Riel's heuristic 4.6; or at least provides data with which to calibrate its interpretation.

Static images of virtual worlds communicate 3D structure much less effectively than interactive 3D browsers that support movement of (or through) the model. We typically view our virtual world visualisations using a VRML web browser plug-in such as Cortona (www.parallelgraphics.com), shown in Figure 119.

Virtual worlds are well suited to more immersive platforms, such as the Magic Book™ [4], shown displaying a cohesion model[6] in Figure 120, CAVE environments

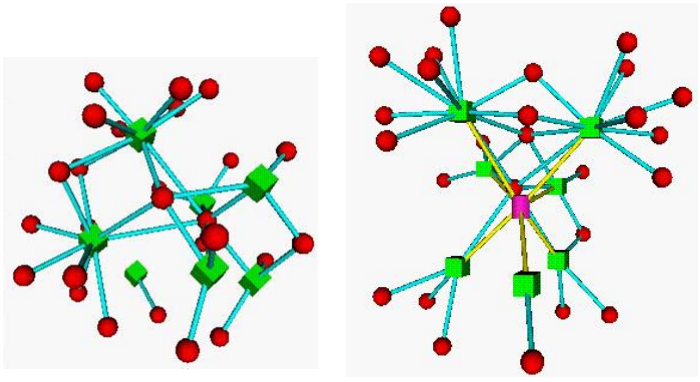

Figure 118: Real class cohesion

[6] Photograph by Eric Woods (HIT Lab NZ).

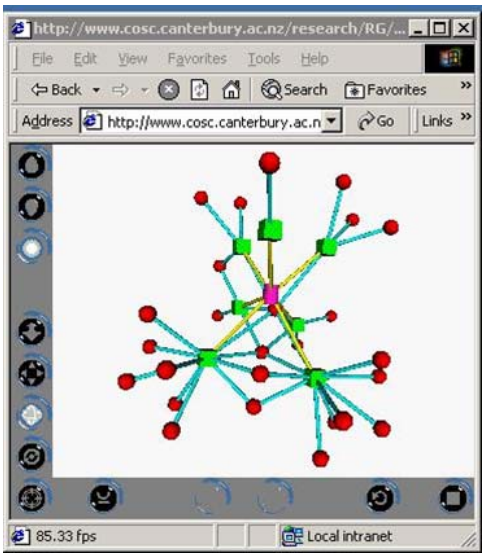

Figure 119: VRML browser

Figure 120: Magic book

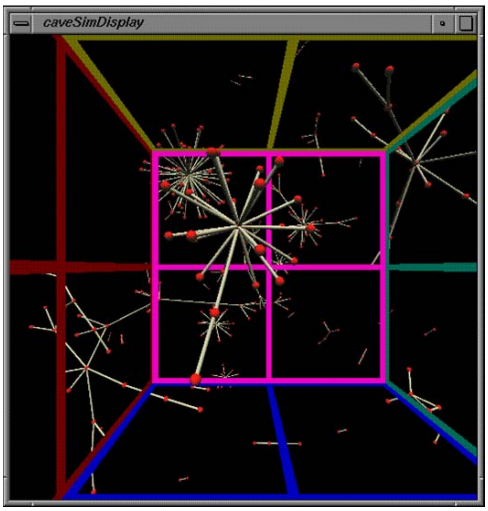

Figure 121: VT CAVE (console)

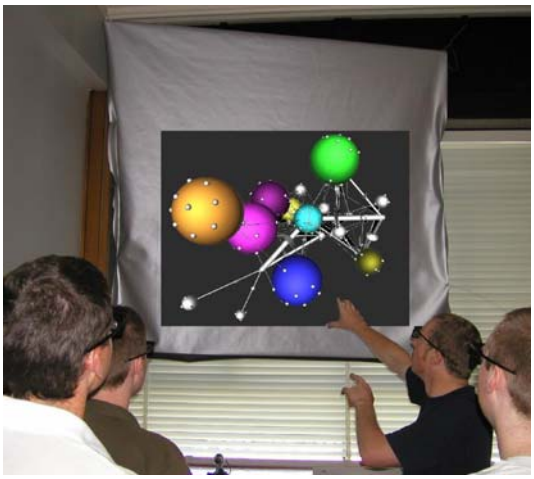

Figure 122: UC GeoWall

like that of Virginia Tech (http://www.cave.vt.edu) in Figure 121, and the UC GeoWall in Figure 122. The CAVE is displaying a galaxy of class cohesion models and allows 'flying' around them. The GeoWall shows a *class cluster* visualisation described in the next section.

These more ambitious virtual environments help to overcome limitations of conventional visualisation approaches.

6.4.1 Class Clusters

We have extended our 3D graph-based visualisation approach with a new visualisation called class clusters, intended to depict coupling between classes. Figure 123 shows an example *class cluster* virtual world visualisation [49] produced by the pipeline. The software being visualised is JST itself. The large spheres are classes, with diameter proportional to a class size metric. Inheritance relationships are shown as (red) cylindrical rods with a cone indicating direction of the superclass. The diameter of each inheritance cylinder is proportional to another metric: the number of subclasses directly or indirectly inheriting via that relationship. This helps to convey the relative importance of these relationships in the inheritance structure. 'Pinheads' on the surface of classes represent public methods. They are connected by (yellow) invocation lines.

The overall shape of the class cluster reflects the net forces of the relationships, with inheritance more rigid than invocations. Heavily coupled classes are drawn together and classes that are connected to many others are pulled into the centre of the graph. In the figure, the

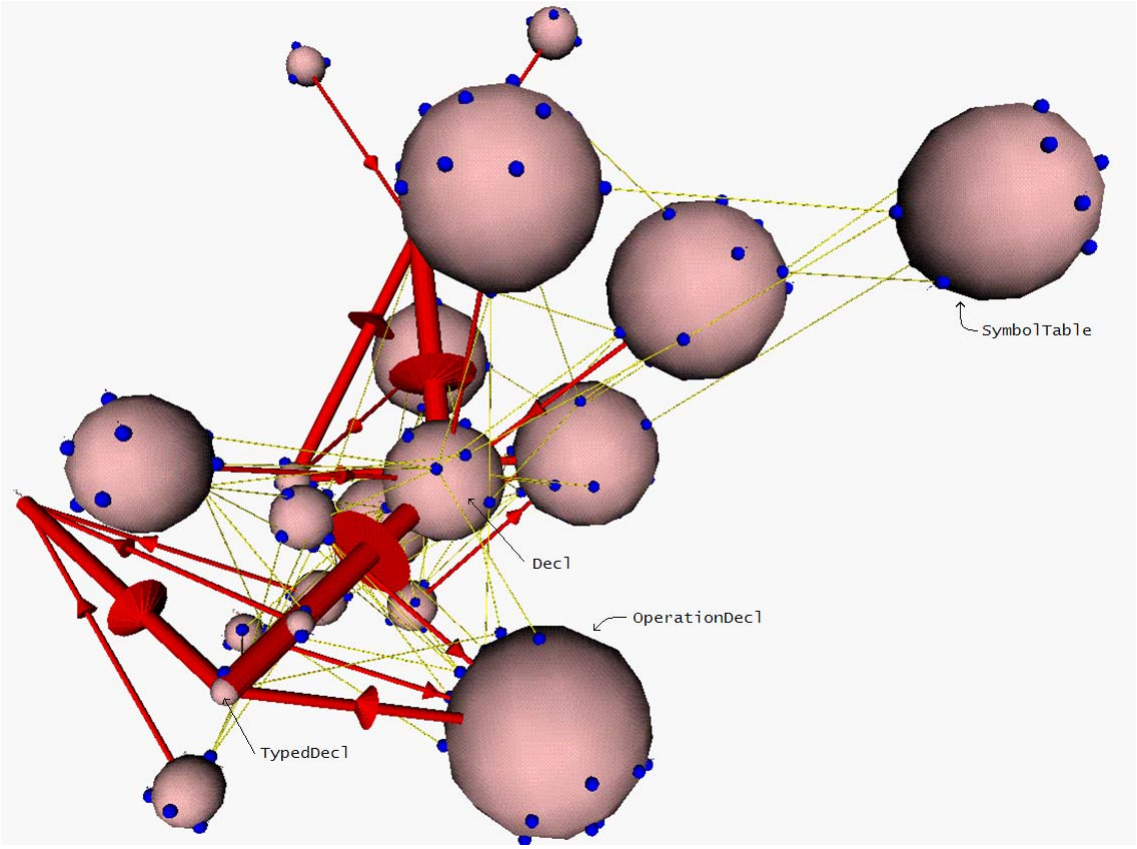

Figure 123: Class cluster

`Decl` class occupies a central place, indicating its centrality to the design. (Unsurprisingly, `Decl` also ranks highly using our `CodeRank` metric.) A counter-example is provided by `SymbolTable`, which is outside the main inheritance hierarchy and has a more peripheral role in the design.

In the foreground of the figure, `TypedDecl` and `OperationDecl` are notably contrasting in form. `TypedDecl` is small and little used—it provides little functionality—but it plays an important role as a supertype of many classes. `OperationDecl`, on the other hand plays a lesser role as a supertype, but is much more substantial and more heavily coupled.

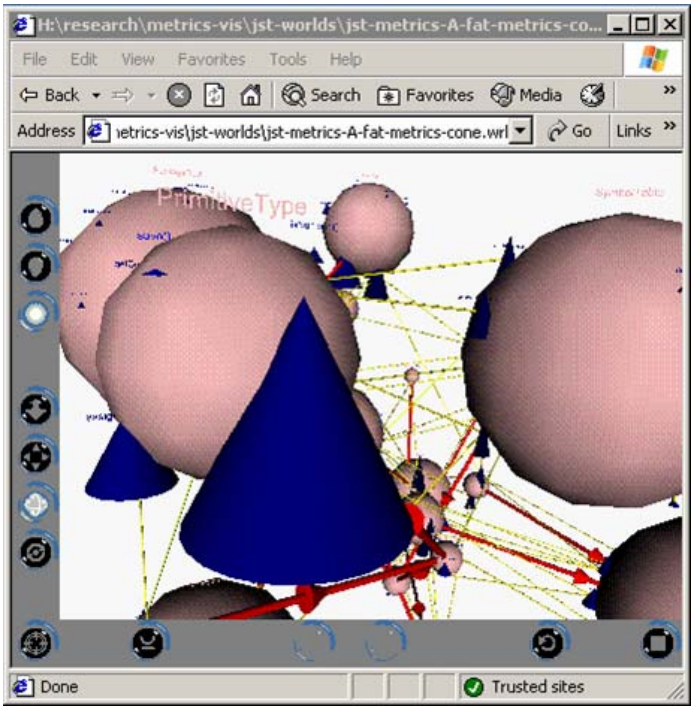

Figure 124: Class cluster with method cones

Figure 124 and Figure 125 show class clusters with additional metric decorations. Method 'pinheads' have been replaced with cones, whose heights correspond to method length and widths to method complexity. The first figure orients all method cones vertically, while the second orients them radially around their class. Although the metric mappings are essentially the same, the appearance is quite different and there is less occlusion in the latter figure.

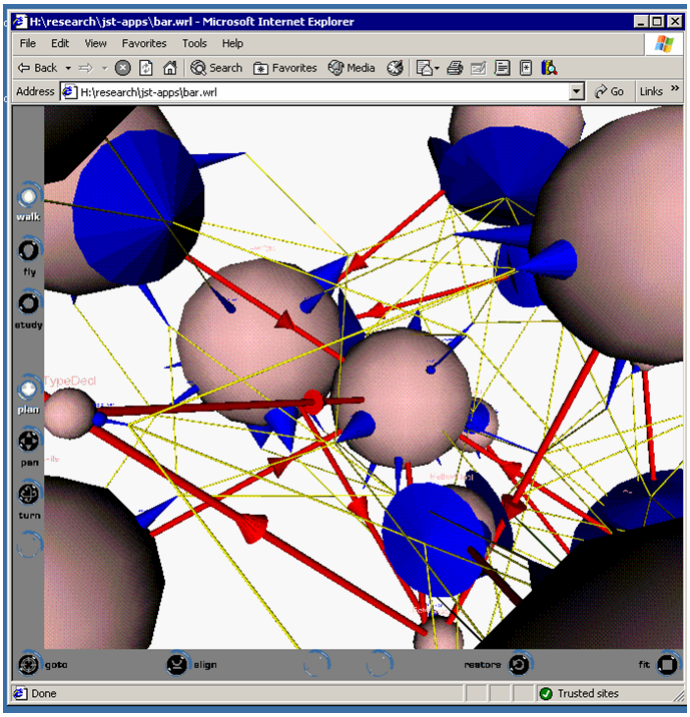

Figure 125: Class cluster with method spikes

As we have noted, the focus of this thesis is on facilitating software

engineering tools such as the metrics and visualisation applications described above, rather than on evaluating metrics and visualisations themselves. We do not discuss application details further here, but the interested reader is referred to our visualisation papers such as [49], [50], [13] and [15].

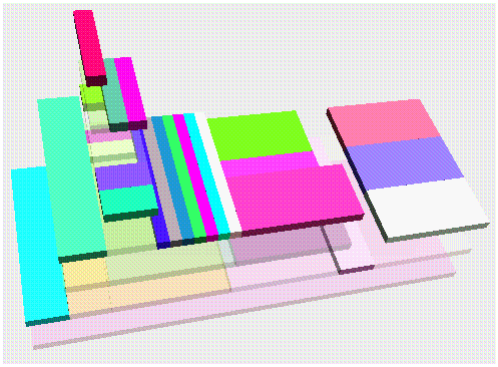

Figure 126: NOC 3D TreeMap

6.4.2 Other applications

The examples described above use the pipeline to produce 3D graph-based visualisations delivered by VRML. We do not mean to suggest, however, that the pipeline is constrained to these types of visualisation or delivery technology. Members of SEVG have used the visualisation pipeline to generate—from the same semantic model type—a variety of visualisation styles. These include conventional 2D graphs, 2D TreeMap visualisations [53], a new 3D (VRML) TreeMap [15], alternative 3D metaphors, and Java 3D presentation in place of VRML. Figure 126 shows a 3D TreeMap of the Number Of Children (NOC) metric for classes in a small system. Figure 127 shows a *Debugs* visualisation developed by Sarah Frater, which presents each class as a metaphorical bug, whose characteristics are derived from several metrics.

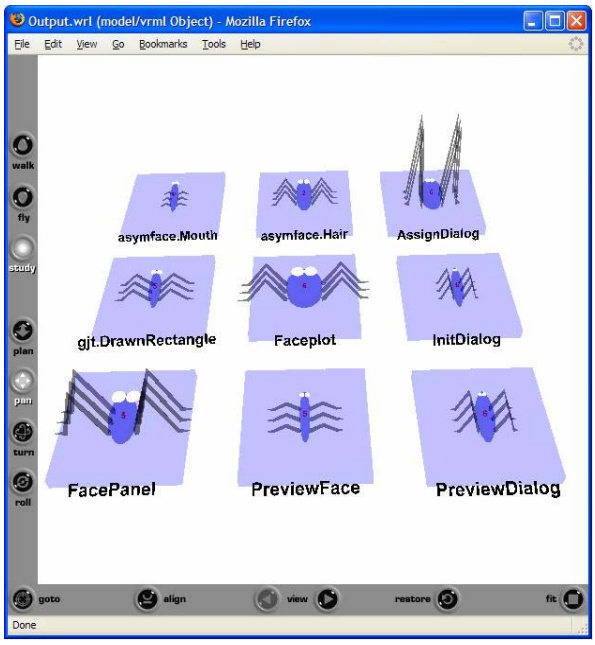

Figure 127: Debugs

While a pipeline is a useful vehicle for our modelling tools in many applications, it is not mandatory. Cook's IDE (discussed earlier) uses the model directly as a library. The IDE is a substantial application that places heavy demands on the model. Its ability to withstand that load provides evidence the implementation is sound.

We continue to find new applications for our modelling technology and expect to initiate many more projects based on the static analysis foundation we have built.

Chapter 7

Conclusions and future work

7.1 Conclusions

As software continues to increase in size and complexity, software engineers need better tools to help them understand and develop their products. Static analysis is a key component of software engineering tools: it is the primary means by which they acquire information about software structure. This thesis describes improvements to static analysis technology and demonstrates these improvements in a range of example applications.

Today, any software engineering researcher or practitioner who wishes to build a new tool is confronted with an immediate obstacle: it is difficult to acquire complete, high quality data sets capable of representing the complex multifaceted nature of software. Existing sources of software structure information, such as reflection and IDE repositories, usually were not designed for the single purpose of producing high-fidelity representations of the complete set of software features, and so are likely to impose compromises on developers of new tools. Tool builders may choose instead to develop their own models by parsing and semantically analysing source code, but this too presents difficulties if current approaches are used.

Parsing is a fundamental static analysis activity for all software represented as source code. Although parsing theory is mature and largely complete, the application of parsing theory to

practical software engineering problems is less so. Conventional parsing wisdom holds that LALR(1) is the most suitable algorithm for generating parsers of programming languages. We find that in applications where conformance to a standard grammar is important, use of LALR(1) should not be automatic, and that a range of LR parsing algorithms up to and including GLR can be employed to advantage.

We have developed a new approach for generating LR parsers, and implemented it in yakyacc. The approach uses an extended version of the GLR algorithm *within the parser generator itself* to explore and modify the parsing automaton. This enables an escalating approach to parser construction, in which progressively more powerful parsing algorithms are applied to individual states until they become adequate or cannot be improved further. The final increment in this escalation involves state splitting to produce LR(k) states, but only where they actually improve the recognition power of the automaton. The approach also generates heterogeneous lookahead depths. The resulting automaton is a hybrid of the various parsing classes used.

The use of hybrid parsing algorithms and heterogeneous k mitigates combinatorial explosions that would otherwise make higher values of k in general and LR(k) for $k > 1$ impractical. When even these more powerful deterministic parser classes are inadequate for a given grammar, we use GLR parsing to accommodate any CFG, including ambiguous ones, without sacrificing linear performance for real languages.

Our approach integrates a number of previously separate LR parsing innovations, and yields a parser generator with significant practical advantages over current technology. It enables a fundamental change in the way parsers are developed: rather than modifying a grammar to accommodate a parsing algorithm, the parsing algorithm adapts to the grammar. This change eliminates the need for manual intervention in parser development, and allows parsing to conform to standard grammars. Benefits include improved rigour of static analysis, as well as easier parser development.

Parsing, nevertheless, does not expose the deep structure of software and for this semantic analysis is required. We have developed a semantic modeller, JST, for the Java language. JST models the ways in which elements of the Java type system (packages, classes, methods and so on) are used in programs. It takes advantage of our parser's conformance to the Java

Language Specification to produce a model that has high fidelity to the specification, and this in turn enables us to build precise tools for purposes such as metrics and visualisation. JST records all relationships between semantic model entities, and also relates them to their syntactic representations. The resulting model is a complete and accurate representation of static software structure, exposed in a form that makes it easily accessible, open for extension and able to be applied in many ways .

JST and derived versions of it have been used in several experimental applications, including a collaborative IDE, OO design heuristics auditor, and metrics and visualisation pipelines. These applications demonstrate that JST can be used in real tools and can process programs of realistic size and complexity, and can even meet the performance demands of a real-time collaborative software engineering setting. They establish the efficacy of the approach in a range of roles, and provide positive indicators of its robustness and scalability.

The applications we have developed using JST share a common theme: they provide information to software engineers (and their tools) in order to enhance understanding of software and encourage insights into how it might be improved. This goal will continue to motivate research for the foreseeable future, and our static analysis technology can facilitate progress by providing a sound basis for the acquisition of static software structure information.

7.2 Continuing and future work

Our semantic analysis technology is only a beginning. We are continuing to refine existing tools and expand the range of applications based on it.

As software engineering tools mature, static analysis will increase in importance and breadth of application. This trend is evident in higher levels of static information—notably generics—being introduced to languages like Java and C#, in the growing use of automated tools for refactoring and code auditing, and in increasingly ambitious software tools and IDEs. Another software engineering trend is the growth of multi-language software, and this raises challenges and opportunities for static analysis technology.

We anticipate continued advances in the fields of software metrics and visualisation, which despite many years of research, are still in their infancy. Few metrics and visualisations for software designers are, as yet, sufficiently enlightening to be adopted as mainstream development practices. Our work in this area will continue to investigate metrics and visualisations that shed light on software characteristics that are relevant to designers' decision-making. We have conducted a number of experiments that test metrics and visualisations, but long term studies using real software development projects are needed to clarify the value of static analysis information to designers.

Frater [77] has taken some initial steps toward using our semantic modelling approach for evaluating software in the light of design heuristics and maxims. Early signs are encouraging, but a larger set of heuristic tests is needed, as well as effective means of communicating results (such as the "wiggly green underline" interface proposed in Chapter 6). The system should be tested with real users.

Some collaborators have made valuable enhancements to JST, including Huynh's Java 1.5 extensions [76], and Cook's support for incomplete models [19]. These improvements should be integrated into the main version. Cook also achieved useful results by tracking model changes over time, but the method needs further development and the potential for new metrics and visualisations afforded by the extra dimension of time has barely been tapped.

JST models only the Java language, but Neate [75] has derived from it a model capable of representing the Common Type System of .NET. Huynh has added a Java-.NET mapping. The availability of a common semantic model mapped to language-specific models raises the possibility of metrics and visualisations that can reliably be calculated and compared across a range of OO languages, yet can still be communicated in terms of the original languages.

Our static analysis work has so far been limited to modelling type systems. Further analysis can build on this base. For instance, we might trace the flow of objects through methods and expressions to determine how object interfaces are actually used. Our model also provides a useful framework to which dynamic analysis information might be attached, yielding a more complete picture.

Yakyacc needs to undergo industrial hardening to cope with the range of conditions encountered by general-purpose parser generators. Better error handling will greatly expand the range of settings in which it can be applied, by allowing it to work with code that has not necessarily already been compiled. Experiments are needed to clarify the effects of longer lookaheads and hybrid algorithms when using different grammars.

7.3 Final words

The outcome of this thesis is superior static analysis technology that facilitates development of new software engineering tools. Several papers have been published to report the findings—and many more will follow. The main contributions are:

- An elegant new LR parser generator algorithm that employs an enhanced GLR automaton *in the generator*. The algorithm escalates parsing power on a per-state basis to produce hybrid automata and heterogeneous lookahead depth. This integrates several previously separate approaches and delivers a broader range of parsers than are used in current practice, including practical versions of powerful parsers such as LR(*k*) and GLR . This leads to a fundamental change in how parsers are developed: standard grammars can be used.

- A tool, yakyacc, that implements our parser generation approach. It also employs XML file formats and external code generation to gain a level of flexibility that is not found in existing parser generators.

- An approach to semantic modelling that takes advantage of standard-conformant parsing to produce a complete, accurate representation of the way a program uses a language's type system, and to disseminate it to other tools.

- An implementation of our semantic modelling approach for the Java language, JST. The model is comprehensive and conforms more faithfully to the Java type system than existing alternatives.

- Several applications that demonstrate the value of our static analysis contributions, including an enhanced metrics and visualisation pipeline. A very promising new

family of metrics, CodeRank, and several new visualisations have been developed. Our technology has also served as the basis for a successful Collaborative Software Engineering research project.

These contributions advance the state of the art of software tool building, and ultimately will help software engineers to understand and improve their products.

Acknowledgments

I am indebted to my family, colleagues and friends for their support throughout my research. I gratefully acknowledge the contributions of:

- My supervisor, Dr. Neville Churcher, who consistently provided encouragement, advice and warped humour throughout this research. His ideas are woven inextricably into this document.

- My colleagues in the department of Computer Science and Software Engineering at Canterbury, and especially Dr. Tim Bell, who initiated my employment and so made this work possible.

- All of the members of SEVG whose ideas and energy helped spark mine.

- The Foundation for Research Science and Technology, for funding the formative stages of this research with a GRIF grant.

- My delightful sons Isaac and Riley, who have never known a father free from the obsession of writing a thesis.

- My wonderful wife Liz, above all, who allowed me to direct my time and energy away from her for so long.

Thank you.

References

1. Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

2. Aho, A.V. and Ullman, J.D. *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

3. Arthorne, J. and Laffra, C. *Official Eclipse 3.0 FAQs (Eclipse Series)*. Addison-Wesley Professional, 2004.

4. Billinghurst, M., Kato, H. and Poupyrev, I. The MagicBook-Moving Seamlessly between Reality and Virtuality. *IEEE Computer Graphics and Applications*, *21* (3). 6-8.

5. Bloch, J. and Gafter, N. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Addison-Wesley Professional, 2005.

6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

7. Card, S.K., Mackinlay, J.D. and Shneiderman, B. *Readings in Information Visualisation: Using Vision to Think*. Morgan Kaufman, San Francisco, 1999.

8. Carey, R. and Bell, G. *The Annotated VRML 2.0 Reference manual*. Addison Wesley Professional, 1997.

9. Chidamber, S. and Kemerer, C. Towards a Metric Suite for Object Oriented Design. *ACM SIGPLAN Notices: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, *26* (11). 197-211.

10. Chomsky, N. On Certain Formal Properties of Grammars. *Information and Control*, *2*. 137-167.

11. Churcher, N.I. and Creek, A., Building Virtual Worlds with the Big-Bang Model. in *Australian Symposium on Information Visualisation, (invis.au 2001)*, (Sydney, Australia, 2001), ACS, 87-94.

12. Churcher, N.I., Frater, S., Huynh, C.P. and Irwin, W., Supporting OO Design Heuristics. in *ASWEC2007: Proceedings of the 2007 Australian Software Engineering Conference*, (Melbourne, 2007), IEEE Computer Society, 101-110.

13. Churcher, N.I. and Irwin, W., Informing the Design of Pipeline-Based Software Visualisations. in *APVIS2005: Asia-Pacific Symposium on Information Visualisation*, (Sydney, Australia, 2005), ACS, 59-68.

14. Churcher, N.I., Irwin, W. and Cook, C., Inhomogeneous Force-Directed Layout Algorithms in the Visualisation Pipeline: From Layouts to Visualisations. in *InVis.au2004 Australasian Symposium on Information Visualisation*, (Christchurch, New Zealand, 2004), 43-51.

15. Churcher, N.I., Keown, L.M. and Irwin, W., Virtual Worlds for Software Visualisation. in *SoftVis99 Software Visualisation Workshop*, (Sydney, Australia, 1999), 9-16.

16. Churcher, N.I. and Shepperd, M. Comment on "A Metrics Suite for Object Oriented Design". *IEEE Transactions on Software Engineering, 21* (3). 263-265.

17. Churcher, N.I. and Shepperd, M. Towards a Conceptual Framework for OO Software Metrics. *ACM SIGSOFT Software Engineering Notes, 20* (2). 69-75.

18. Conte, S.D., Dunsmore, H.E. and Shen, V.Y. *Software Engineering Metrics and Models*. Benjamin Cummings, Redwood City, CA, 1986.

19. Cook, C. Ph.D. Thesis: Towards Computer Supported Collaborative Software Engineering *Department of Computer Science and Software Engineering*, University of Canterbury, Christchurch, 2006.

20. Cook, C. and Churcher, N.I., Modelling and Measuring Collaborative Software Engineering. in *Proc. ACSC2005: Twenty-Eighth Australasian Computer Science Conference*, (Newcastle, Australia, 2005), ACS, 267-277.

21. Cook, C., Irwin, W. and Churcher, N.I., Towards Synchronous Collaborative Software Engineering. in *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*, (Busan, Korea, 2004), IEEE Computer Society Press, 230-239.

22. DeRemer, F.L. Ph.D. Thesis: Practical Translators for LR(k) Languages, MIT, Cambridge, MA, 1969.

23. DeRemer, F.L. Simple LR(k) Grammars. *Communications of the ACM, 14* (7). 453-460.

24. DeRemer, F.L. and Penello, J.P., Efficient Computation of LALR(1) Look-Ahead Sets. in *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*, (Denver, CO, USA, 1979), ACM Press, 176-187.

25. Di Battista, G., Eades, P., Tamassia, R. and Tollis, I.G. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

26. Dijkstra, E.W. (ed.), *On the role of scientific thought*. Springer-Verlag, New York., 1982.

27. Donnelley, C. and Stallman, R. The Yacc-compatible parser generator, Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, 1995.

28. Earley, J. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM, 13* (2). 94-102.

29. Eclipse Foundation. Eclipse web site, http://www.eclipse.org, 2006.

30. Eick, S.C., Steffen, J.L. and Sumner, E.E. Seesoft-a Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering, 18* (11). 957-968.

31. Fenton, N.E. and Pfleeger, S.L. *Software Metrics : A Rigorous and Practical Approach*. International Thomson Computer Press, London, 1997.

32. Fischer, G. Ph.D. Thesis: Incremental LR(1) Parser Construction as an Aid To Syntactical Extensibility *Department of Computer Science*, University of Dortmund, Dortmund, Germany, 1980.

33. Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003.

34. Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999.

35. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., USA, 1995.

36. Gosling, J. *The Java language specification*. Sun Microsystems, Palo Alto, CA, USA, 2000.

37. Gough, K.J. *Syntax Analysis and Software Tools*. Addison-Wesley, 1988.

38. Griffith, A. *GCC: The Complete Reference*. McGraw-Hill, 2002.

39. Grune, D. and Jacobs, C.J.H. *Parsing Techniques : A Practical Guide*. Ellis Horwood, New York, 1990.

40. Halstead, M.H. *Elements of Software Science*. Elsevier North-Holland, New York, 1977.

41. Harold, E.R. *Processing XML with Java: A Guide to SAX, DOM, JAXP, and TrAX*. Addison Wesley, 2003.

42. Henderson-Sellers, B. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

43. Holmes, J. *Object-Oriented Compiler Construction*. Prentice Hall, 1994.

44. Holser, P. Limitations of Reflective Method Lookup. *Java Report, Vol. 6, no. 8 (Aug. 2001).*

45. Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass., 1979.

46. Huynh, C.P. Honours Project: Modelling Programming Language Semantics Using a Common Semantic Model, University of Canterbury, 2006.

47. IEEE. Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE, 1990.

48. Irwin, W. and Churcher, N.I. A Generated Parser of C++. *N.Z. Journal of Computing, 8* (3). 26-37.

49. Irwin, W. and Churcher, N.I., Object Oriented Metrics: Precision Tools and Configurable Visualisations. in *In METRICS2003: 9th IEEE Symposium on Software Metrics*, (Sydney, Australia, 2003), 112-123.

50. Irwin, W. and Churcher, N.I., XML in the Visualisation Pipeline. in *Visualisation 2001. Workshop on Visual Information Processing.*, (Sydney, Australia, 2001), Australian Computer Society Inc., 59-67.

51. Irwin, W., Cook, C. and Churcher, N.I., Parsing and semantic modelling for software engineering applications. in *Australian Software Engineering Conference*, (Brisbane, Australia, 2005), IEEE Press, 180-189.

52. ISO. ISO/IEC 14882:1998(E) Programming Languages--C++, American National Standards Institute, 1998.

53. Johnson, B. and Schneiderman, B., Tree-maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. in *Proc. Visualization '91*, (Los Alamitos, CA, 1991), IEEE Computer Society Press, 284-291.

54. Johnson, S.C. *Yacc: yet another compiler-compiler*. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

55. Johnston, K. and Smith, B. Javasrc, http://javasrc.sourceforge.net, 2002.

56. Johnstone, A., Scott, E. and Economopoulos, G. Evaluating GLR Parsing Algorithms. *Science of Computer Programming, 61* (3). 228-244.

57. Johnstone, A., Scott, E. and Economopoulos, G., Generalised Parsing: Some costs. in *Compiler Construction, 13th Intnl. Conf, CC'04*, (2004), Springer, 89-103.

58. Johnstone, A., Scott, E. and Economopoulos, G. The Grammar Tool Box: A Case Study Comparing GLR Parsing Algorithms. *Electronic Notes in Theoretical Computer Science, 110*. v97-113.

59. Kay, M. *XSLT Programmer's Reference*. Wrox, 2001.

60. Kipps, J.R. GLR Parsing in Time $O(n^3)$. in Tomita, M. ed. *Generalized LR Parsing*, Kluwer, Boston, 1991, 43-60.

61. Knuth, D.E. On the Translation of Languages from Left to Right. *Information and Control, 8*. 607-639.

62. Korenjak, A.J. A Practical Method for Constructing LR(k) Processors. *Communications of the ACM, 12* (11). 613-623.

63. Kyburg, H.E. *Theory and Measurement*. Cambridge University Press, Cambridge, 1984.

64. Lang, B. Deterministic Techniques for Efficient Non-Deterministic Parsers. *Automata, Languages and Programming: Lecture Notes in Computer Science, 14*. 255-269.

65. Lieberherr, K., Holland, I. and Riel, A., Object-oriented programming: an objective sense of style. in *Conference on Object Oriented Programming Systems Languages and Applications*, (San Diego, California, United States, 1988), 323 - 333.

66. Lilley, J. PCCTS-Based C++ Parser Page, http://www.empathy.com/pccts, 1997.

67. Liskov, B. and Wing, J. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems, 16* (6). 1811–1841.

68. Lorenz, M. and Kidd, J. *Object-Oriented Software Metrics*. Prentice Hall, 1994.

69. Martin, R.C. Granularity. *C++ Report, 8* (10). 57-62.

70. McCabe, T.J., A Complexity Measure. in *International Conference on Software Engineering*, (San Francisco, California, United States, 1976), IEEE Computer Society Press, 308-320.

71. McCall, J., Richards, P. and Walters, G. Factors in Software Quality, Rome Air Development Center, United States Air Force, Hanscom AFB, MA, 1977.

72. McPeak, S. and Necula, G.C., Elkhound: A Fast, Practical GLR Parser Generator. in *CC04: Proceedings of Compiler Construction*, (2004), 73-88.

73. Melton, H. and Tempero, E., JooJ: Real-Time Support For Avoiding Cyclic Dependencies. in *Proc. Thirtieth Australasian Computer Science Conference (ACSC2007)*, (Ballarat Australia, 2007), CRPIT, ACS, 87-95.

74. Miller, B. Catalog of Free Compilers and Interpreters, http://www.idiom.com/free-compilers, 2006.

75. Naur, P. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM, 3* (5). 299-314.

76. Neate, B. Honours Project: An Object-Oriented Semantic Model for .NET, University of Canterbury, 2005.

77. Neate, B., Irwin, W. and Churcher, N.I., CodeRank: A New Family of Software Metrics. in *ASWEC2006: Australian Software Engineering Conference*, (Sydney, 2005), IEEE, 369-378.

78. Nejmeh, B.A. NPATH: A Measure of Execution Path Complexity and its Applications. *Communications of the ACM, 31* (2). 188-200.

79. Nicol, G.T. *Flex, The Lexical Scanner Generator*. Free Software Foundation, 1993.

80. Nozohoor-Farshi, R. GLR Parsing for E-Grammars. in Tomita, M. ed. *Generalized LR parsing*, Kluwer, Amsterdam, 1991, 60-75.

81. Page, L., Brin, S., Motwani, R. and Winograd, T. The PageRank Citation Ranking: Bringing Order to the Web, Stanford Digital Library Technologies Project, 1998.

82. Pager, D., The Lane Tracing Algorithm for Constructing LR(k) Parsers. in *Annual ACM Symposium on Theory of Computing*, (Austin, Texas, United States, 1973), ACM Press, 172-181.

83. Pager, D. A Practical General Method for Constructing LR(k) Parsers. *Acta Informatica*, *7* (3). 249-268.

84. Parnas, D.L., Designing Software for Ease of Extension and Contraction. in *ICSE 78: Proceedings of the 3rd international conference on Software engineering*, (Piscataway, NJ, USA, 1978), IEEE Press, 264–277.

85. Parnas, D.L. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, *15* (12). 1053 - 1058.

86. Parr, T.J. Ph.D. Thesis: Obtaining Practical Variants of LL (K) and LR (K) for K Greater Than 1 by Splitting the Atomic K-tuple, Purdue University, Layfatette, IN, 1993.

87. Parr, T.J. and Quong, R.W. ANTLR - a predicated-LL(k) parser generator. *Software Practice and Experience*, *25* (7). 789--810.

88. Potanin, A., Noble, J., Frean, M. and Biddle, R. Scale-Free Geometry in OO Programs. *Communications of the ACM*, *48* (5). 99-103.

89. Poulin, J. *Measuring Software Reuse: Principles, Practices and Economic Models*. Addison Wesley, 1997.

90. Purai, S. and Vaishnavi, V. Product Metrics for Object-Oriented Systems. *ACM Computing Surveys*, *35* (2). 191-221.

91. Rekers, J. Ph.D. Thesis: Parser Generation for Interactive Environments, University of Amsterdam, Amsterdam, 1992.

92. Resenkrantz, D.J. and Hunt, H.B. Efficient Algorithms for Automatic Construction and Compactification of Parsing Grammars. *ACM Transactions on Programming Languages and Systems*, *9* (4). 543-566.

93. Riel, A.J. *Object-Oriented Design Heuristics*. Addison-Wesley, Reading, Mass., 1996.

94. Schroeder, W., Martin, K. and Lorensen, B. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall, 1996.

95. Spector, D. Efficient Full LR(1) Parser Generation. *ACM SIGPLAN Notices*, *23* (12). 143-150.

96. Spector, D. Full LR(1) Parser Generation. *ACM SIGPLAN Notices, 16* (8). 58-66.

97. Spence, R. *Information Visualisation*. Addison Wesley, 2001.

98. Stallman, R. *Using the GNU Compiler Collection*. Free Software Foundation, Inc., Cambridge, Massachusetts, 2003.

99. Stanchfield, S. and Parr, T. Parsers, Part IV: A Java Cross-Reference Tool, Mage-Lang Institute, Java Developers Connection, 1997.

100. Tomita, M. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986.

101. Tomita, M., LR Parsers for Natural Languages. in *COLING84: Proceedings of the 10th international conference on Computational Linguistics*, (Stanford, California, USA, 1984), Association for Computational Linguistics, 354-357.

102. Tucker, A. and Noonan, R. *Programming Languages: Principles and Paradigms*. McGraw-Hill, New York, 2002.

103. Unger, C.F. A Global Parser for Context-Free Phrase Structure Grammars. *Communications of the ACM, 11* (4). 240-247.

104. van den Brand, M.G.J., Heering, J., Klint, P. and Olivier, P.A. Compiling Language Definitions: the ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems, 24* (4). 334-368.

105. van den Brand, M.G.J., Sellink, M.P.A. and Verhoef, C., Current Parsing Techniques in Software Renovation Considered Harmful. in *Proc. Sixth International Workshop on Program Comprehension*, (1998), IEEE Computer Society, 108-117.

106. Yang, H.Y., Tempero, E.D. and Berrigan, R., Detecting Indirect Coupling. in *Australian Software Engineering Conference*, (Brisbane, Australia, 2005), IEEE Computer Society, 212-221.

107. Younger, D.H. Recognition of Context-Free Languages in Time n^3. *Information and Control, 10* (2). 189-206.

108. Yourdon, E. and Constantine, L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, Englewood Cliffs, N.J., 1979.

109. Zuse, H. *Software Complexity: Measures and Methods*. de Gruyter, Berlin, 1991.