Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

12-2007

Table driven prediction for recursive descent LL(k) parsers

William M. Leiserson

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Leiserson, William M., "Table driven prediction for recursive descent LL(k) parsers" (2007). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Table Driven Prediction for Recursive Descent LL(k) Parsers

by

William M Leiserson

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Supervised by

Professor James Heliotis, PhD
Department of Computer Science
Golisano, College of Computing and Information Science
Rochester Institute of Technology
Rochester, New York
December 2007

Approved By:

James Heliotis

James Heliotis, PhD
Professor of Computer Science, RIT
Primary Adviser

Axel Schreiner

Axel Schreiner, PhD
Professor of Computer Science, RIT

F. Kazemian 12/7/07

Fereydoun Kazemian, PhD
Professor of Computer Science, RIT

Thesis Release Permission Form

Rochester Institute of Technology

Golisano College of Computing and Information Science

Title: Table Driven Prediction for Recursive Descent LL(k) Parsers

I, William M Leiserson, hereby grant permission to the Wallace Memorial Library reporduce my thesis in whole or part.

William Leiserson

William M Leiserson

12/8/2007

Date

Acknowledgments

I am grateful to Prof. Jim Heliotis for his help in developing this work and especially for directing me to prior art in the area of design. Also, he has greatly encouraged me to step back and consider the work in the context of the big picture. I would also like to express my thanks to Prof. Axel Schreiner who guided me through building the project and writing the thesis. *Oops* v.2 is his brainchild and it was his interest in extending it that led to this particular topic. Thirdly, thanks to Prof. Fereydoun Kazemian who, among other things, fostered an interest in language processing during his course on *Programming Language Theory*. Finally, there are many other people inside and outside of the RIT Computer Science Dept. who directed me to research, helped me think through the content, and generally made this a better thesis, and I would be remiss for not thanking them.

Abstract

Programming languages are typically described in BNF or some extension of BNF, and the process of converting these descriptions into parsers is performed by parser generators. Some of the parser generators that convert LL grammars into parsers construct them to use recursive descent that gives them context during execution. The context is provided by the execution stack as the parser descends into the grammar and this is what allows the full expressiveness of LL grammars. Table driven parsers can be generated instead but restrictions are placed on the LL grammars that can be accepted. The benefit of tables is that they facilitate a separation of syntax analysis and semantic code written by a language designer. They are also faster and they simplify language implementation and modification. This paper proposes the possibility of a hybrid system that makes decisions using tables but once decisions are made recursive descent is employed to maintain context. The benefits of each system are maintained, and the drawbacks are mitigated. Also discussed are the modifications made to an existing parser generator, *oops* (version 2), so that it accepts LL(k) grammars and builds parsers using this system as proof-of-concept.

Contents

Chapter 1

Introduction

Programmers often develop their own domain-specific languages for projects and applications as evidenced by the abundance and variety of tools available to them. Parser generators, the tools used to convert high-level descriptions of languages into recognizers, generally fall into one of two classifications: LL and LR. Programmers who choose LL parser generators prefer the flexible forms of language descriptions that they offer. LR parser generators, on the other hand, are generally faster and the set of languages they can describe is larger. LR parsers have generally won out in industrial settings due to their efficiency and power.

LL parsers built by parser generators typically look very much like hand coded parsers because of the conceptual intuitiveness of LL. However, this evolution from hand coding to automated generation has maintained implicit design decisions that were practical early on but have become a hinderance to modern tools. In making predictions they show preference to certain alternatives and as a consequence they encourage modification that takes time and potentially introduces bugs. Many of these "vestigial" elements of generated LL parsers are not present in LR parsers because the popular solution to building them uses an entirely different, table-driven mechanism[1]. Although table-driven LL parser generators exist that overcome many of the difficulties plaguing recursive descent ones, they necessarily sacrifice the flexibility of input and accept a smaller set of grammars. This weaker set

[1]The scheme of parsing LR(k) grammars is defined by the conversion of grammars to states by Knuth in [9].

of "Strong LL grammars" comes about in table-driven LL parsers because there is no stack to act as context. If the same non-terminal is used in multiple rules when the state machine has completed the non-terminal it cannot distinguish which rule is the correct parent.

This paper is not primarily concerned with making fast LL parsers, but with advocating the automatic generation of robust parsers. Robust parser generation: 1. divides the work of the language designer between syntactic analysis and semantic actions, defines the production of the syntactic analyzer as the work of the tool, and 2. makes the tool produce parsers with deliberate design decisions that make modification unnecessary. An added benefit is that in separating the interface from the parser, itself, the parser generator designer is given flexibility to modify the way it builds parsers in whatever way without the modifications being visible to the programmers who already use it.

Chapter 2 (Background) deals specifically with the development of conventional LR and LL parser generators, the way they build parsers, their interfaces, and their benefits and limitations. It also discusses the differences between LL and LR parsing and the reasons why LR parsing has managed to avoid some of the problems from which LL parsing suffers. It defines the problem and the motivation for solving it. Chapter 3 (Implementation) describes the work that was done to the LL parser generator, *oops* v.2, as an example of a solution. The conclusions of the research are presented in Chapter 4. Finally, some possible avenues of exploration and additional development is suggested in Chapter 5 (Future Work).

Chapter 2

Background

2.1 Conventional Tools and Principles

Modern language processing theory has made programming language development conceptually simple. Recognizing a language has become a matter of describing the syntax of the language in a high-level form and defining actions on an input as it is processed (tree generation, immediate evaluation, conversion to another form, etc.) written in a programming language. The syntax description is used by a tool to build a parser and the appropriate sections of the action code are executed by the parser. The parser can be thought of as the syntactic part of the language processor and the action code can be considered the semantic part.

There is value for both the parser generator designer and the user in keeping the two parts well-separated and making the syntactic portion exclusively the domain of the tool. A tool that implements this in its interface has the following benefits:

- Extensibility: as parsing technology advances a tool can be updated without affecting existing users (language designers). Input that worked in the old version can be used with the new one. Also, it is easier to retarget a particular tool to another language. Users who want to retarget their language processors must update their semantic code but the (presumably debugged) syntactic input remains unchanged.

- Flexibility: the format for the syntactic description of a language can be optimized for

simplicity and intuition. This makes grammars easier to debug, maintain, and update. Also, the separation removes many of the particular difficulties such as namespace issues where a user has to avoid certain naming conventions to keep from adversely affecting the parser.

- Reliability: parsers that don't rely on user code but are built from tool-generated code (or pre-compiled classes) are more reliable than parsers that mix tool-generated code and user-generated code. If bugs are found in the parser, fixing the tool fixes the bugs in all of the parsers anybody generates. Also, parsers that are modified by users after they have been generated by a tool have to be modified every time the grammar is modified. This is difficult and it creates many opportunities for the introduction of bugs. It is best simply to have the tool generate the best parser the first time.

- Aesthetics: an input to a tool for which the syntactic and semantic parts of the code are separated leads to more attractive input. This is useful not only for the sake of flexibility but for communication. If an input is human-readable it can easily be taught to or understood by someone who is not necessarily familiar with the particular tool. Also, the tool itself has a smaller learning curve. Rather than learning an esoteric form of input, simple, standard grammar representations like the ones that appear in the literature can be used.

These benefits are a consequence of clear delineation between the parts of a language processor and providing them to end users through a tool's features is a good design goal. The delineation is in keeping with the Law of Demeter[11] that advocates minimal interaction between objects. The application of the Law of Demeter is more completely described in **Motivation** (section 2.2).

Some modern parser generators draw a reasonably clear line between the parser and the semantic interpreter, though many do not. The difference is frequently related to the language description format and the way in which the parser functions. However, the more flexible forms of input are generally the ones for which the line is less clear.

2.1.1 Language Description Formats

Backus Normal Form (or Backus Naur Form) (BNF)[5] is a notational system that expresses context-free grammars and is generally the system understood by language processing tools, today. Context-free grammars are represented in BNF in terms of rules with context free non-terminals on the left and the syntactic translations on the right.

SetOfStatements ::= SetOfStatements Statement | Statement ;
Statement ::= Declarative | Interrogative ;
Declarative ::= Subject verb Predicate "." ;
Interrogative ::= verb Subject Predicate "?" ;
etc.

Figure 2.1: Sample BNF grammar.

Figure 2.1 is somewhat simplified from BNF as described by J.W. Backus et al[5], where non-terminals were represented in angle-brackets, terminal keywords were bold, and terminal characters were strings. In this paper for simplicity non-terminals are capitalized, terminals (not just characters) are not, and string literals (a type of terminal) are quoted. Multiple symbols in a row (non-terminals, terminals, and string literals) indicate that the particular set of symbols appear in sequence in any legitimate input string. The "or" ("|") indicates an alternative right-hand side for a rule. For repetition, BNF permits recursion, as in **SetOfStatements** (this rule will resolve to one or more instances of **Statement**). Parentheses can be used for precedence. Each rule is terminated by a semicolon.

Niklaus Wirth developed Extended BNF (EBNF)[20] to standardize and simplify grammar representations. He shortened grammars by removing the need for many instances of recursion. This made grammars more readable and was useful for certain types of parsers. One type of EBNF (the initial form Wirth defined) uses brackets to signify repetition as in figure 2.2 (though, there are a number of generally accepted ways to represent EBNF[16]).

Curly braces indicate one or more repetitions of the enclosed symbols. Square brackets indicate zero or one repetition of the enclosed symbols. By combining them (e.g.

SetOfStatements = { Statement } ;

Figure 2.2: Sample EBNF grammar fragment.

`[{Statement}]`), one can represent zero or more repetitions[1]. Also, even as tools use some variation of EBNF alternations ("|") are not only applied to a complete right-hand side but can be used within the right-hand side of a rule by using brackets to enforce precedence. Thus, for language designers EBNF greatly simplifies grammar descriptions.

BNF and EBNF usually (but not always) correspond to LR and LL parsing, respectively. LR parsers consider the right-hand sides of rules by building a stack of states (roughly corresponding to input symbols) and reducing them to other states (roughly corresponding to left-hand sides) until the only thing left on the stack is the start symbol. Each reduction passes the symbols to be replaced to the user code for semantic evaluation. Consequently, LR parser generators more readily prefer BNF descriptions because they can use the right-hand sides of rules as the exact strings for reduction and can build tables to make deciding fast. With respect to user interfaces, LR parser generators typically divide the syntactic and semantic portions of their code well. At reduction time the semantic code can access the symbols on the stack by number (as in many tools) because the exact set of symbols to which the user has access is known ahead of time. There is no need for semantic code interspersed with the syntactic definition. The parsing process is typically fast and reliable. The tool's user never sees or directly interacts with the parsing tables.

Some of the older tools like *yacc*[7], a C targeted parser generator, use their own representations of BNF as syntactic input. Semantic routines are placed near their right-hand side syntactic descriptions.

The input is taken by *yacc* and converted to an output file written in C with the semantic code embedded in the parser code. This interface has potential namespace difficulties but *yacc* attempts to avoid this by prepending a "*yy*" to all of its internal variables.

[1]This shorthand for "zero or more" repetitions was not explicitly defined by Wirth in [20] but clearly it has the intended meaning.

```
SetOfStatements:  SetOfStatements Statement {
                  // C code snippet.
                  }
                  | Statement {
                  // C code snippet.
                  } ;
```

Figure 2.3: Input fragment to *yacc*.

With the development of object oriented patterns, however, some LR parser generators have divided the work of language processing even more completely. *SableCC*[6] is targeted to *Java* and it uses the visitor pattern to connect the syntactic and semantic portions of its processors. The grammar input includes some annotation to indicate the names of the classes it will generate and pass to the visitor. Although it incorporates some aspects of EBNF a language developer must treat the or operator ("|") strictly as defining right hand sides of rules.

LL parsers consider the left-hand sides of rules and expand into the right-hand sides as they read input. LL is more intuitive because this treatment of a grammar can be expressed in terms of recursive descent. In other words, one can "walk through" a grammar by looking at the start symbol, "descending" into the right-hand side, walking across terminal symbols, and descending further into non-terminals as they appear. This is the traditional method for constructing LL parsers and LL parsers are easy to hand-code.[19] This style of parser generator encourages input formats like EBNF that are easily converted to parser code.

The simplicity of conversion has led to the creation of many LL parser generators that convert grammars into human-readable recursive descent parser code. The direct conversion, however, does not produce the most efficient parsers. The code produced is often hindered by seemingly innocuous differences in grammars such as the order of alternatives. Consider the simple example presented in figure 2.4.

A human readable parser favors the first alternative ('a') in that a comparison of parser input is performed to 'a' first, and if that fails, a comparison is made to 'b'. In general,

R1 = a | b ;

Figure 2.4: Example grammar.

in an LL(1) grammar a parser with n alternatives will require $O(n)$ comparisons in order to make a decision. However, the complexity increases as more tokens of lookahead are required.

Lookahead in LL grammars is the number of tokens, k, a parser requires in order to make a parse decision without backtracking (backtracking is trying one solution and backing up if it fails). After a method of parsing is chosen a grammar is defined by its lookahead using that method. When measuring the lookahead in the grammar in fig. 2.4 using an LL parsing method k = 1 token because only the first token is needed to decide between the alternatives. Thus, the grammar is said to be LL(1). The set of grammars that can be parsed by LL parsing methods with an arbitrary k tokens of lookahead are called LL(k) grammars.

In general, if k tokens of lookahead are used, decisions take $O(n*k)$ time. It is possible to generate a minimal machine that uses $O(n + k)$ comparisons to make decisions for a parser with n alternatives and a lookahead depth of k but even the popular tools[14][17] use an algorithm with a complexity of $O(n * k)$. The particulars of these algorithms are discussed in **Motivation** (section 2.2).

An alternative is to build a purely table driven LL parser much like in LR parsers. These LL parsers can be constructed without recursive descent[18], but in doing this, they lose the contextual power of recursion. With LL recursive-descent parsers an implicit stack of rules is created on the program execution stack and the context is useful as rules are completed and control is subsequently returned to the caller. Parser generators like SLK[18] build purely table-driven parsers that are limited to strong LL grammars. Since LR parsers don't use this form of context purely table-driven operation is popular with them and they don't lose any power through this implementation.

2.2 Motivation

The present state of technology for predictive LL(k) parser generators that use recursive descent is a non-robust system that encourages modification of the generated code. The weakness of a modified/modifiable parser when compared with a robust parser (one that does not need modification) is that user modifications potentially introduce bugs. This is compounded as a language, itself, is altered and as the parser is regenerated and remodified by the user. It would be best to have a tool that produces correct parsers the first time. Thus, the desire emerges for a flexible tool that incorporates the full power of LL(k) prediction and produces robust parsers.

This notion of making a sturdy parser, independent of the end user, conforms to the Law of Demeter[11], wherein discrete parts of an application are maintained separately, and interaction between them is strictly regulated. The goal of Demeter is modularity. Although it is apparent that modern LL(k) parser generators construct their parsers along the same lines as hand-coded ones, the Law of Demeter suggests that this should not be the concern of the user. In other words, it should not be necessary for a tool's user to alter the underlying parser implementation, as this is more of a liability than a benefit. In practice this has actually been a hinderance.

The popular tools, ANTLR[14] and JavaCC[17], are used as a basis for discussion. Both tools consider grammatical decisions in terms of if-statements, and if there are n alternatives and k tokens of lookahead, then $O(n * k)$ comparisons are performed to make a descent prediction. Consider the LL(2) grammar fragment in figure 2.5 and the corresponding pseudocode in figure 2.6.

```
S  =  a a A | a b B | a c C ;
```

Figure 2.5: Example grammar.

When code similar to this is generated by a tool, six comparisons must be made before the third alternative can be executed. Preference is given to alternatives that appear earlier

```
RULE-S()
1  if t1 = a and t2 = a
       then ALT-1()
2  elseif t1 = a and t2 = b
       then ALT-2()
3  elseif t1 = a and t2 = c
       then ALT-3()
4  else ERROR()
```

Figure 2.6: Conventional prediction pseudocode.

in the grammar. Furthermore, a savvy user is likely to modify the code by condensing all of the *t1* comparisons into a single comparison. All of the if-statements with *t2* comparisons would then appear inside of that condensed statement bringing the number of comparisons necessary to settle on the third alternative to 4. If it is later decided that the third option is the most common the user has a choice between modifying the grammar so that the third alternative is placed first and remaking the optimizations on the regenerated parser or making further optimizations on this parser. All of this modification of the parser code is the hurdle because it must be made every time the parser is regenerated and the sections of code that are changed are likely to become a source of bugs. Optimally, the tools would produce parsers with no preference for any alternative and use an algorithm that doesn't require or even permit modification.

A different lookahead system that discourages modification, while providing good service to the end programmer, uses tables to perform lookups. This is different from a truly table-driven parser in that once a decision has been made, recursive descent is still used. However, rather than taking $O(n * k)$ operations to make a decision, the tables need only take $O(k)$ time. A table corresponding to the grammar fragment is in figure 2.7.

This is more solid because the time it takes to make a decision is based on the grammar rather than on any preference of the tool. In this case, any decision is made with two lookups, though it is apparent that for other LL(2) grammars, many of the decisions could

Table	Symbol	Decision
t1	a	t2
	b	*error*
	c	*error*
t2	a	alt-1
	b	alt-2
	c	alt-3

Figure 2.7: Lookahead table.

be made with a single lookup. As an added benefit, error discovery occurs much more quickly. In the conventional system, an error is discovered only after all possibilities have been exhausted. With tables, errors are discovered with no more than k lookups, the same as any prediction. This solution is preferable and can be implemented in any tool that strictly regulates interaction between the user and the generated parser.

The input formats to conventional LL parser generator tools, however, frequently conflate the syntactic and semantic parts of the language processor. *ANTLR*[14], for example, mixes syntactic predicates with its EBNF.

```
expr  :  <<isvar(LATEX(1))>>?  ID "\(" expr_list "\)"
         <<array_ref_action>>
      |  <<isfunc(LATEX(1))>>?  ID "\(" expr_list "\)"
         <<fn_call_action>>
      ;
```

Figure 2.8: ANTLR Semantic Predicate

The code in figure 2.8 is an example of a semantic predicate that decides whether the input is a function call or an array reference. The *isvar* and *isfunc* elements are user-defined functions that return boolean values to help the parser decide which subrule it should choose. This means that user code is executed as part of the decision-making process which potentially introduces bugs into the parser, itself.

JavaCC[17], another Java targeted parser generator has the same limitations because

its input format expresses the non-terminals as the actual function calls that are present in the parser that is built. In both of these cases much of the implementation of the parser is exposed in the parser generator input and because of parser generation preferences modification of the output is encouraged.

2.3 The Oops Observer Interface

The *oops*[2] parser generator[10] provides the expressiveness of full EBNF and separates generated parsers from the semantic interpreters written by the users and from the users themselves. *oops* takes pre-written classes, instantiates them, and configures those instantiations to form a parser rather than generating code for the user to modify and compile. The instantiated classes that form the parser are then serialized and can be stored to disk. Interaction with the parser is regulated through an observer interface. The observer interface was adapted from the JAXP[13] model of a parser that invokes methods on a handler. In JAXP a single handler is passed to the parser before it is run and data is passed to that handler until execution is complete. In the observer interface implemented by *oops* a user can provide a new rule-specific handler for each grammar rule instantiation. Each handler is called an observer.

When an *oops*-generated language processor is executed the serialized parser is converted back into a data structure and is given control. Each time the parser descends into a rule the method `init` is invoked on the current observer, along with the rule name and number. At that time the observer has the opportunity to return a new observer for that rule instantiation. Thus, as the parser descends through the grammar a stack of observers is created (assuming the observers are returning new observers each time rules are instantiated). As symbols are processed by the parser it gives them to the current observer through `shift`

[2]All discussion of *oops* relates to *oops* v.2, the version that was modified for this paper. The newer version, v.3, was not available when research began.

methods. When the rule is complete a `reduce` method is called on the observer, signaling it that it will receive no more input. The previous observer becomes the handler again and the reduced observer is now treated as a completed non-terminal from a grammatical standpoint. If the method `value` is invoked on it, it is expected to return whatever meaning it derived from its execution. The reduced observer object is now passed to the current observer through another `shift` method. When writing an observer for a rule each terminal comes through as the scanned string corresponding to that token and each non-terminal comes through as a reduced observer object.

Through the observer interface, the semantic code is almost entirely removed from the syntactic description. *Oops*' input formats are simple and clear to anyone who knows EBNF. Modifications to the tool can be made with little regard to the observers written by users, and new parser theory can be implemented without making input obsolete. If the designers of *oops* want to implement parsers in another language like C++ or C# new runtime classes need to be implemented for parser generation, but the end-users can use the same input – albeit with new observers.

Observers are also flexible for users. If a visitor pattern (as in SableCC[6] and others) is desired, a trivial observer can be written to produce an abstract syntax tree that can be visited. If, instead, a simple grammar-checker is desired, or something to trace the parsing of a program, an observer can be written (and has already been written and is included in the *oops* package) to do that. A user can write multiple observers for a single grammar – e.g. a user can write a compiler and an interpreter using a single parser for a language. Importantly, *oops* discourages modification of its parsers. Although it expresses preference for certain alternatives, the aforementioned observer interface permits easy modification.

Figure2.9 shows a language processor built with the *oops* architecture. A file in some form of EBNF is converted into a parser, and a set of observers, written in *Java* are compiled and used by the parser to process input.

The existing observer interface has some shortcomings that tie it unnecessarily to a particular parser implementation, however. Although observers in *oops* generally hold to

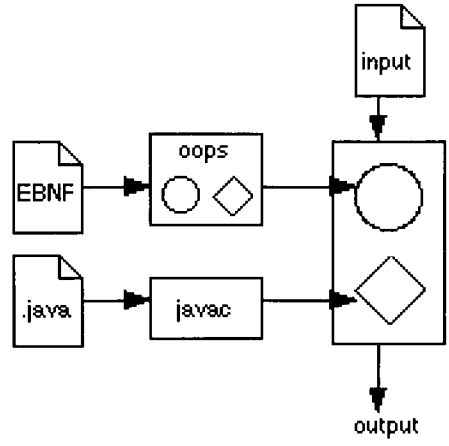

Figure 2.9: The *oops* framework for parser generation.

```
R1  :  `a` 'b' "c" R2 ;
```

Figure 2.10: Grammar input to *oops*.

a handler pattern of data transfer in that data is pushed from the parser to the observers, and data does not go the other way, the exception *oops* makes is in the retrieval of parsed data that it has to request directly from the scanner. This is an obstacle to extending *oops* to accept LL(k) grammars in its current form. In keeping with the Law of Demeter, language designers should not have direct access to the scanner, but only to the information an observer requires at a particular point in its execution.

In its present form, *oops* distinguishes three types of terminals, and the profile for `shift` methods only specifies the type. An example of input to *oops* is presented in figure 2.10. The three types of quotes correspond to the three types of terminals. Those terminal types – back-quoted (BTerminal), single-quoted (Terminal), and double-quoted (DTerminal) – and the non-terminal comprise the four `shift` methods implemented by observers:

```
public void shift (BTerminal bterminal);
```

```
public void shift (Terminal terminal);
public void shift (DTerminal dterminal);
public void shift (Observer sender);
```

Except for the observer object none of these methods contain any information about what was processed except for some indication of where it appeared in the rule. If `"c"`, for example, represented identifiers the `DTerminal` shows (by looking at the context) that an identifier has been read but the `DTerminal` object does not contain any information about the identifier (such as the text that was read). To retrieve that information the observer must request access to the scanner and query it for the last value read. Optimally, an observer would have no knowledge of the state of the parser (including the scanner) but only what the parser sends it.

Strictly, the interface should fit the following criteria:

1. There can be a different observer class for each individual rule and a different observer object for each rule instantiation.

2. Neither the parser nor any observers have knowledge of the inner workings of the other but the parser only uses predefined callback methods on the current observer.

3. The parser passes all relevant information to the observers during execution.

This division fits in with the required constraints and ensures modularity. Either side, the parser or the observer, should be sufficiently modular to develop without significantly influencing the other and neither should have to reach into the inner workings of the other to function. *Oops* observers as developed from handlers are duly suited to this.

In a modified version of *oops* (as demonstrated in Chapter 3: **Implementation**) observers strictly divide the syntax and semantics of a language processor by restricting the flow of information. Information moves from the parser to user-defined *observers* in the form of parsed terminal and non-terminal symbols. No information is passed from the observers to the parser during execution. This restriction divides the syntactic and semantic

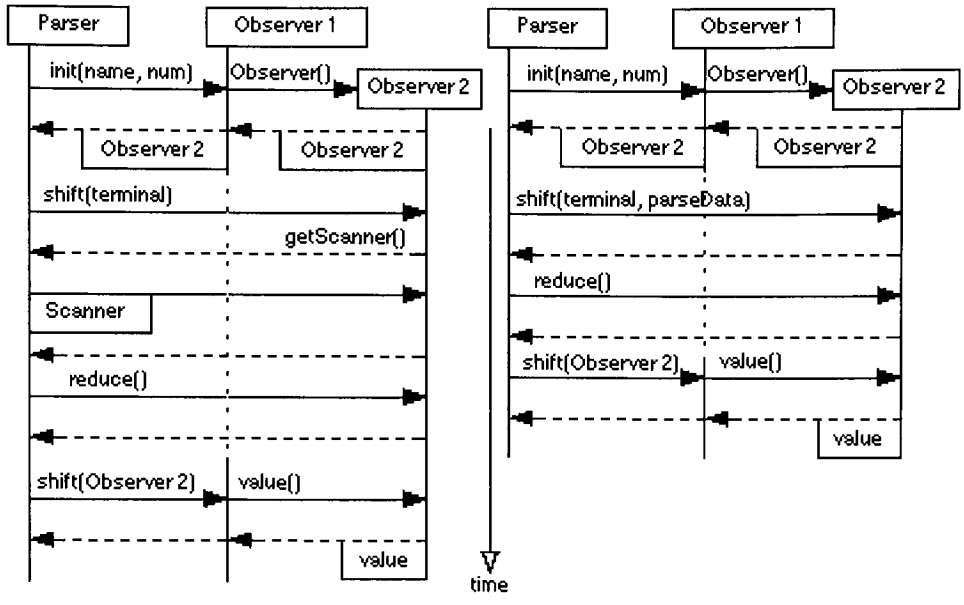

Figure 2.11: Flow of control diagrams for *oops* observers before and after modification.

parts of the language processor and it permits the design of a simple modular user interface. A flow control diagram demonstrating the change is given in figure 2.11.

Communication between the parser generator designer and the language designer is facilitated by the guarantee that the user interface does not have to be altered on account of changes to the parser. It may be that the user interface changes but the changes should not correspond to particular ways in which the parser functions. Thus, the designer can make any changes to the parser generator necessary and can completely redesign the way parsers function without forcing the language designer (the user) to modify a single line of code. Similarly, the input to the parser generator is not influenced by changes to the way parsers are generated. Under these constraints *oops* is fit to be modified to implement robust LL(k) parsers.

Chapter 3

Implementation

3.1 Purpose

Converting *oops* from an LL(1) to an LL(k) parser generator was intended as a proof-of-concept that demonstrated a way to construct LL(k) parsers that adhere to a strict division between syntax and semantics especially for the reasons described in Chapter 2. *Oops* was the preferred subject of this extension because in its old form it already had a number of the intended properties. The grammar used to construct the parser and the observers used to interpret the input are entirely distinct. Furthermore, it builds its parsers from prewritten Java classes rather than writing Java code. Finally, it flattens the parser for storage using Java's `Serializable` interface, further isolating it from modification. Although in its old form its parsers' decisions were made using an algorithm with the same complexity as that of a traditional handwritten parser, the algorithm was encapsulated well within the parsers.

3.2 Oops Development

Oops was significantly modified in how it performed lookahead prediction and the observers were altered to conform more strictly to the division between syntax and semantics in keeping with the Law of Demeter. The result was an extension that permitted *oops* to produce LL(k) parsers. LL(k) grammars are those for which corresponding parsers must look ahead k tokens of input in order to make a parsing decision.

Oops was an LL(1) parser generator that was designed with LL(1) grammars in mind. As a result, when upgrading it to an LL(k) parser generator (among the other functional extensions) some of the original design decisions had to be altered. After modifying the the observer interface to reduce direct interaction between parser and observer the significance of these changes combined with the lack of impact on the end user was a testament to the interface's flexibility. The new observers functioned independently of the modifications to their parsers and required no further alterations themselves.

Oops was altered in the following ways: the observer interface was updated, a lookahead buffer was introduced to replace direct access to the scanner, the new table-driven system of acquiring and using lookahead was designed, and *JLex* replaced the old scanner generation system. This chapter discusses these modifications.

3.3 The Oops Observer Interface

The observer interface of *oops* (prior to the changes) divided well the work of syntax recognition from the work of application of semantic meaning. However, there was a shortcoming in that a user was still required to query the scanner in order to access l-values associated with terminals as they were shifted. When instantiated an observer would ask the parser for direct access to the scanner and whenever `shift` was called for a terminal the observer would ask the scanner for its most recent value. This was inconsistent with the design goals in two related ways: (1) it provided the user with direct access to elements of the parsing process, and consequently (2) such an interface would not scale when transitioning from LL(1) to LL(k).

Lack of scalability meant that a simple extension of *oops* would provide bad data to its observers. Grammars that at any point required multiple tokens of lookahead would not give the correct l-values corresponding to their terminals at those points. A parser that requires more than one token of lookahead in order to make decisions will treat the syntactic and semantic portions of the language processor asynchronously. In other words,

during execution the syntactic and semantic portions of the program may be at different places in the grammar. There are times when the scanner is advanced through the input by the parser beyond the current point of action on the part of the observer. In the case of *oops* this meant that an observer querying the scanner might get incorrect information.

```
R1 = R2 | R3 ;
R2 = a b ;
R3 = a c ;
```

Figure 3.1: Example grammar.

Consider the LL(2) grammar in figure 3.1. When the parser examines the lookahead for **R1** it must scan two tokens ahead in order to decide between the alternatives. If the scanner reads an 'a' followed by a 'b' the parser instantiates an observer for **R2** and invokes its `shift` method, saying that a value has been read. Although the observer should receive an 'a' when it queries the scanner it will receive a 'b' because the scanner has already moved ahead in the input. In fact, before scanning further, the parser will probably invoke the observer's `shift` method again, and it will receive the same 'b' it did before. Even this cannot be guaranteed depending upon the implementation of the parser (e.g. one implementation might tokenize all of the input before the first observer is instantiated). Hence, any interface that requires direct user interaction with the scanner is unnecessarily tied to the parser implementation.

This necessitated a modification to the *oops* observer interface such that observers would be passed all of the relevant data by the parser and the user would no longer require access to any elements of the parser itself during execution. Each `shift` method corresponding to the reading of a terminal was modified so that it was passed a `ParseData` object in addition to the type of terminal shifted. `ParseData` is a class that was written as a wrapper for the value from the scanner, the line number, and a character counter. Thus, in the example above the value 'a' is now wrapped in a `ParseData` object and passed through the `shift` method. Furthermore, if the user wants data regarding the location of

the terminal in the input that information is available and is guaranteed to be correct.

For the purposes of this chapter only one `Terminal` type will be treated and any discussion will be assumed to apply to the other two. Thus, in the original version of *oops* the profile for the terminal `shift` method was:

```
public void shift (Parser.Terminal sender);
```

In the modified version of *oops* that conforms to the observer pattern the `shift` method is now:

```
public void shift (Parser.Terminal sender, ParseData pd);
```

3.4 The Lookahead Buffer

Oops initially only required access to the scanner which would store the most recent token and value. In an LL(1) grammar only one token of lookahead is required to make a decision so the scanner had an internal buffer for a single token. Extending *oops* to resolve LL(k) grammars, however, required a corresponding extension to the buffer. A queue was introduced to wrap the scanner and push tokens onto one end as they were required for search and pop them off the other as they were passed to the user.

The queue was implemented as the `LookaheadBuffer` class and is now the only class with access to the scanner (except for deprecated methods in other classes for backwards compatibility). Now when any new parser has to make a decision it iterates through the buffer's contents as far as is necessary. If the `LookaheadBuffer` has reached the end of tokenized input it moves the scanner ahead, pushes the read value onto itself and continues to return values to the parser. When a `shift` method is called a value is popped off the queue and sent to the observer.

3.5 Lookahead Architecture

3.5.1 Framework

The initial design of *oops* parsers corresponded roughly to the design described by Metsker[12]. Parsers were constructed from three main types of classes that corresponded to `Sequence` (a b), `Alternation` (a | b), and `Repetition` (a{*min, max*}). Rules were expressed in hierarchies of objects corresponding to these structures. `Alternation` was expanded to form `Or` and `And` constructs (in addition to its implicit `Xor` meaning) that permitted/required multiple alternates to be executed (e.g. an `And` node expressed by "a & b" recognized either permutation of "a" and "b"). Additionally, other classes were used to represent the remaining parts of a functioning parser corresponding to a grammar. Leaves in a parser tree were represented by `Terminal`[1] and `Nonterminal` nodes, and the root of each tree - corresponding to a grammar rule - was a `Rule` node.

The product that *oops* delivered was a simplified abstract syntax tree of the input grammar. *Oops* itself was a parser with observers. Its observers produced an AST on which methods could be called to minimize the tree, optimize it, generate lookahead, and any other functions its designers wished. The final visitor method was a generate function called `gen` that recursively worked its way through the tree building a simplified data structure that only included the data and methods needed for execution. The simplified structure was the language parser that was returned and serialized.

This framework did not change when *oops* was modified. Parsers are still constructed from these basic types. The `Or` and `And` nodes remain limited to LL(1), however, for reasons that will be discussed below.

[1] As mentioned above, terminals were represented by three classes but they will be treated together in this chapter.

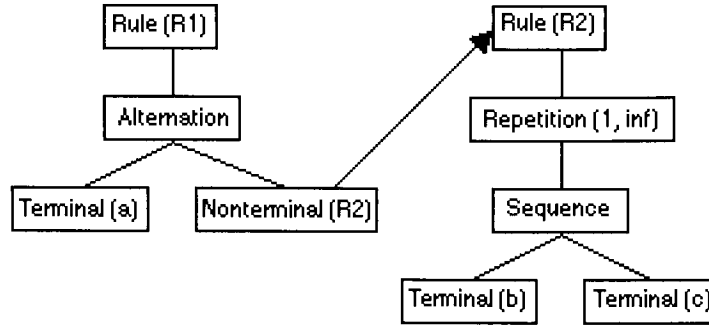

Figure 3.2: Sample *oops* parser hierarchy.

3.5.2 Using Lookahead

Prior to modification lookahead was represented by sets of tokens. Each node had a "first-set". Any node that could accept no input (i.e. it could be skipped during execution) also had a "follow-set" of tokens that might appear immediately after it in a grammar. In an LL grammar the first-set is comprised of a set of strings (or individual terminal symbols for an LL(1) grammar) that a parser can expect to see when it is run. The set is used by the parser to make decisions between alternatives. The follow-set is used when the first-set contains the empty string ("ϵ") and does not have to consume any input. It includes the set of strings that can appear as input after the parser has completed execution. The purposes of first and follow-sets in predictive parsing are described more fully by Appel[1]. Prior to modification, *oops* used the sets in the following ways by the different types of nodes:

- `Terminal`: its first-set was its own token. When executed it would call *shift* on the current observer and return control to its parent. No follow set was ever necessary because no `Terminal` would ever accept no input.

- `Sequence`: the first-set was comprised of the union of the first sets of all of its children in order, up until the first child that could not accept no input. When executed, for each child in sequence it would compare the most recent symbol read by the scanner to the child's first-set. If it was in the set the child was run and it would move on

to the next child. If the symbol was not in the child's lookahead, it would verify that the child could accept no input and move to the next child. If the child required input it would report an error. After control was returned to the `Sequence` node it would proceed through the rest of the children in the same way. When there were no more children it would return to its own parent.

- `Alternation`: an `Alternation` node's first-set was the union of the first sets of all of its children. On execution, it would iterate through its children until it found one that could accept the most recent symbol read by the scanner, execute it, and return. `Or` or `And` would continue until no children would accept the current symbol or all children who required input, respectively, had been run.

- `Repetition`: its first-set was equivalent to its child's first-set. It would ensure its child's first-set contained the current symbol read by the scanner and execute it until the maximum number of repetitions was reached or its child wouldn't accept it. When the symbol was not present in its child's first-set, if the minimum number of executions had been reached it returned. Otherwise it reported an error.

- `Rule`: the first-set was equivalent to its child's first-set. It instantiated an observer corresponding to itself as the new handler and passed control to its child. When control was returned to it, it called `reduce` on the observer and returned to its own parent.

- `Nonterminal`: the first-set was equivalent to that of its corresponding `Rule` which it immediately called.

The modified version of *oops* has a single lookahead tree that combines the functions of the first and follow-sets. A node's tree consists of all legal sequences of symbols - of a length necessary to make a decision - that may appear at its particular point of execution. Leaves in a tree return signals for their nodes indicating which child to call, whether the current node should return control to its parent, or whether the current sequence of symbols

in the `LookaheadBuffer` is illegal (thereby causing an error). Lookahead trees are minimal so that if, for example, a grammar is LL(3) but a particular decision can be made with one symbol strings corresponding to that decision use only one symbol. This maximizes space efficiency and runtime speed efficiency. An example is provided in figure 3.3.

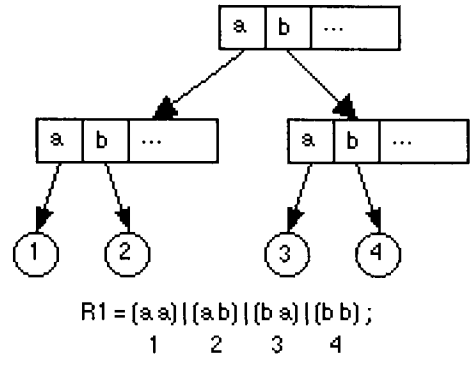

Figure 3.3: Sample lookahead tree for the `Alternation` node in the grammar.

- `Terminal`: has no lookahead tree. As before it `shifts` itself to the current observer when called.

- `Sequence`: a node has as many trees as children. When called it checks the tree corresponding to the first child and calls the child indicated by the returned signal. When the `Sequence` is given control again it moves on to the child immediately following the child just called, checks its lookahead tree, and repeats until there are no more children or it receives a signal to return (or there is an error).

- `Alternation`: the tree indicates which child should be called. When run, it checks the tree against the contents of the `LookaheadBuffer`, runs the appropriate child, and returns. `Or` and `And` nodes each have one lookahead tree with a depth of 1. The same tree is used as many times as there are children to be run. When a reduce signal is received an `Or` node immediately returns. If the `And` node has run all of its children it returns. Otherwise it produces an error.

- `Repetition`: it has one lookahead tree that indicates whether it should call its child or return (or report an error). As before, it decides whether a return signal is legitimate based on its minimum and maximum number of executions.

- `Rule`: a `Rule` has no lookahead tree of its own but simply instantiates a new observer and calls its child as before. When it receives control again it `reduces` the observer and returns.

- `Nonterminal`: it has no lookahead tree but immediately calls the corresponding `Rule`.

3.5.3 `And` and `Or` Nodes

It was decided not to extend the functionality of the special structures designed for XBNF, `Or` and `And`, on account of the incredible complexity of lookahead. The complexity arises from certain cases in which the lookahead for these nodes can use an overwhelming amount of space. In order for an `And` or `Or` parser node to be predictive each time a child is called a different lookahead tree is required to proceed.

R1 = And b ;
And = (a b) & a ;

Figure 3.4: A grammar using `And` constructs.

In figure 3.4 making a decision for the rule **And** requires knowledge of the state and what has already been executed. In a sample run, if the lookahead buffer contains the sequence "a b" there must be a distinction between options 1 and 2 (1: a b, 2: a). The grammar can be decided with 2 tokens of lookahead but the parser has to know what options have already been executed in order to return a correct signal. In this case, there are 3 relevant possibilities: 1. no children have been run, 2. the first child has been run but not the second, or 3. the second child has been run but not the first (the possibility where both

children have been run is not necessary because the `And` node would have already returned to its parent). Reading the sequence "a b" and making a decision requires knowledge of what has already been done. Each of the possibilities can be treated as states, and each state requires its own lookahead tree. In the general case, if there are n options there are $2^n - 1$ states (the state that corresponds to all options having been executed is not necessary, as mentioned in the example). This is a "correct" solution in that for any grammar it will make the correct choice and it does not place constraints on what grammars are considered legal beyond those discussed in section 3.5.4.

There are some alternatives, however. A simpler solution to the grammar in fig. 3.4 is one that makes a lookahead tree for the `And` node and potentially solves the ambiguity in the same way as a first/follow conflict (discussed in section 3.5.4). Whatever resolution, though, new constraints are placed on input grammars and those constraints are beyond the scope of this thesis. Most of the code that would be used to extend `And` and `Or` nodes has been written and is available for someone who wants to remove the restraining code and make such decisions.

But within the scope of this thesis it was decided that correct solutions were preferred for simplicity. The correct solution to predicting descent in `And` and `Or` nodes had an extremely large space complexity. This was decided to be unreasonable so the `Or` and `And` nodes were not extended in functionality (though both were modified to use lookahead trees of depth = 1 instead of sets and may still be used in their limited forms in the new architecture).

3.5.4 Checking Grammars

In predictive LL parsers certain constraints are placed on grammars: no left recursion and no unlimited recursion. There are also limitations on the relationship between first and follow-sets. The following is a brief description of each.

Left recursion means that when a rule is called it can recursively descend and be reached

at a deeper level without consuming any input symbols. Figure 3.5 demonstrates this principle. The algorithm was not altered in the updated version of *oops*.

R1 ::= R1 a | b ;

Figure 3.5: Left recursive grammar.

Unlimited recursion happens when a rule cannot terminate. If there is no way out of a rule once it is entered it will continue to recursively call itself even if it is processing input symbols. Figure 3.6 gives an example of a grammar with unlimited recursion. Again, the algorithm for detection was not altered.

R1 ::= a R1 ;

Figure 3.6: Unlimited recursive grammar.

There are 2 primary constraints on first and follow sets that ensure that for any input there is only one legal path through the parser. The first set of a rule (or subrule) is the set of all legitimate sequences that can be produced by a rule. The follow set is the set of all strings that can legitimately follow a rule (given its context in the rest of the grammar). In certain grammars some strings can be produced in more than one way leading to ambiguities. A parser that processes the input for such a grammar has a choice as to how it will "interpret" that input. The constraints on a grammar ensure that a parser produced by the grammar is deterministic - ie. there is only one correct interpretation for any given input.

1. $\forall s \in FIRST$, s corresponds to exactly one decision.
2. $\epsilon \in FIRST \Rightarrow FIRST \bigcap FOLLOW \equiv \phi$

The first constraint means that no two alternatives can produce identical strings. In the case of the original LL(1) implementation of *oops*, no two alternatives could start with the same symbol (or they would not be LL(1)). This is a shift/shift conflict and cannot be

resolved (comparable to a reduce/reduce conflict in an LR parser). The second constraint restricts a string from being produced both from one of the first-set alternatives and from the follow-set, if the node is not required to process input. If ϵ the empty string is in the first set then any string that appears in both sets is a potential ambiguity. This is the LL version of a shift/reduce conflict and it can be resolved if the constraint is relaxed so that the node will process input from the first-set if it is able. This greedy algorithm solves the *dangling-else* ambiguity: one frequent occurrence in programming languages. An example grammar is provided in figure 3.7.

Stmt ::= Ifstmt | Otherstmt | ...;
Ifstmt ::= if Expr then Stmt (else Stmt)?

Figure 3.7: Dangling else grammar.

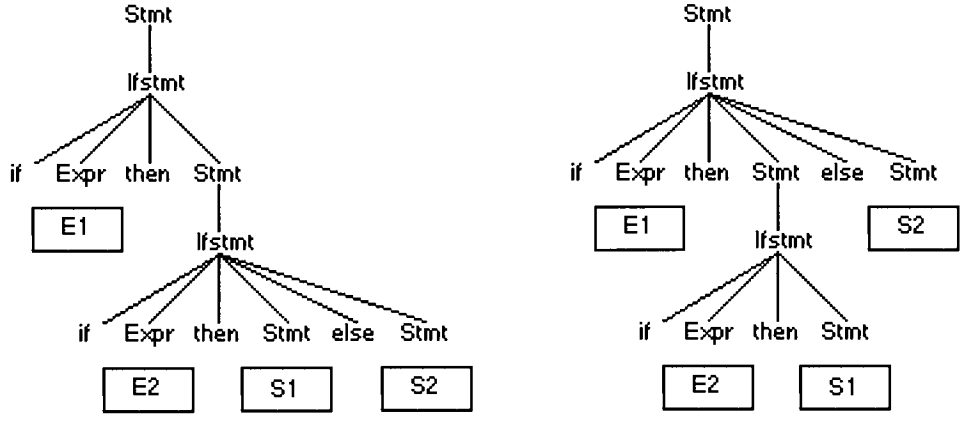

Figure 3.8: Two interpretations of an input to the grammar in figure 3.7.

The grammar does not obey the second clause because an **Ifstmt** within an **Ifstmt** and a single "else" can be parsed in two different ways such that it belongs to either **Ifstmt**. The relaxed second constraint will bind the **Elseclause** with the innermost **Ifstmt** possible (this example and fig. 3.8 were adapted from those presented by Aho, Sethi and Ullman in their

book, *Compilers Principles, Techniques, and Tools*[4]). *Oops* has relaxed this constraint but will still produce a warning if the constraint is broken.

3.6 Computing First Sets

Originally, *oops* nodes built their first-sets by querying their children and received their follow tokens through back-propagation. The back-propagation system was primarily facilitated through `Sequence` nodes that passed follow sets from one child to the previous child.

In order not to search any deeper than necessary the present system uses iterative deepening in its search to resolve lookahead. If a node requires only one token of lookahead in the worst case it will not search any deeper. The old system of back-propagating symbols (or strings in this case) would not have scaled well for this design. If a node required two tokens of lookahead (and accepted empty input) but its successor only required one, anything that was back-propagated would be too short to decide whether there was a conflict.

Instead of back-propagating follow symbols from the nodes that follow, *oops* now back-propagates the nodes themselves and each node maintains a list of its successors that it uses to build its follow-set and extend its first-set when necessary. Since the lookahead tree architecture means first and follow-sets only differ in the signals they return the generation of each happens in a single pass over the tree using the method `lookahead`. In each node the first-set routine is executed and the required lookahead is deepened until the tree is resolved or a maximum is reached. A default maximum is provided to prevent infinite recursion for ambiguous grammars. If two alternatives can generate the same infinite string then the recursive search will continue infinitely, too.

Once the first-set is resolved if the node accepts empty input the follow-set is added to the tree and lookahead is deepened until the grammar is resolved or a maximum is reached. If the maximum lookahead depth is reached and the tree is not resolved because of a shift/shift conflict an error is reported and no parser is generated. If it is not resolved

because of a shift/reduce conflict a warning is reported but a parser will be generated.

The `first` routine receives a set of lookahead tree nodes on which it works. In a call to the `lookahead` method a node generates its lookahead tree (or trees) by calling `first` on its children. The lookahead set it passes contains only the root of the tree (depth of zero). Whenever a node is added to a lookahead set with a depth that is equal to or exceeds the maximum depth of search the set does not retain the symbol. Thus, as strings are lengthened they are gradually removed from the set. An empty lookahead set indicates that the maximum depth of search has been reached. A call to build the first-set differs for each type of node:

- `Terminal`: for each tree node in the lookahead set add a child corresponding to the `Terminal` symbol and return all resulting nodes as a set.
- `Sequence`: call `first` on the first child with the existing lookahead set and get the returned set. If the set is not empty call the second child with it. Continue until the set is empty or until all children have been searched. Return the final lookahead set.
- `Alternation`: return a union of all of the lookahead sets generated by calls to the `first` methods of the children using the original lookahead set.
- `Repetition`: call `first` on the child and replace the lookahead set with the one returned until the minimum number of repetitions has been reached. Then call `first` on the child and union the lookahead set with the one returned until the maximum number of repetitions has been reached or the lookahead set ceases to differ from the one returned. Return that lookahead set.
- `Nonterminal`: call `first` on the corresponding `Rule`.
- `Rule`: call `first` on the child.

If a node that is calculating its own lookahead is returned a non-empty set from `first` then it extends the strings by invoking `first` on its successor nodes (and their successor

nodes and so on) until its required depth is reached. `follow` relies on the `first` methods of a node's successors, too. It is called in the same way that `first` is, and it is passed a lookahead set containing only the root of the lookahead tree.

The lookahead trees that are generated correspond to a complete set of strings for the necessary depth for decision-making up to the predefined maximum. In other words, the solution is correct and every grammar that is within the specified maximum depth of lookahead will be resolved and any illegal grammar will not be resolved. This solution was preferred to a linear approximate solution because it was correct and for most typical LL grammars little slowdown is detected. Even when slowdown is significant on account of a dangling-else it often can be minimized by finding offending rules and specifying lower lookahead for them.

3.7 Scanner Generation

Oops originally had a scanner interface and an implementation of a factory that configured a pre-written scanner class, the `TokenizerScanner` that was adapted from Java's built-in `StringTokenizer`. It was not table-driven and would try to match each alternative in sequence. Therefore, a faster scanner was desired.

The new system incorporates *JLex*[2] which builds table-driven scanners in Java. *JLex* was chosen over the newer *JFlex*[8] because it was simpler to generate the input files automatically (though *JFlex* could be introduced). For the purposes of backwards compatibility, the same command-line options were kept, and regular expressions were developed that corresponded to them. With the updated *oops*, a unique scanner is constructed, compiled, instantiated, and serialized along with each parser.

Unfortunately, this removes one of the aesthetic properties of the old version of *oops* wherein it would compile its own input format and prove that the result was equivalent to itself. The property is removed in that the scanner classes have different names and will

cause a minor difference in the output because of it. This may be an insurmountable difficulty given the use of a code-generating scanner generator as there is ambiguity when two potentially distinct classes have the same name. In the interests of retaining the principle a command-line option is provided to use the old scanner instead of creating a new one with *JLex*.

Chapter 4

Conclusions

Prior to the modifications *oops* used a decision algorithm with the same complexity as other conventional LL parsers. Furthermore, preference was given to whichever alternate was considered first. Its more general architecture, however, made it ideal for this proof-of-concept experiment. Since the parsers *oops* generated were built from prefabricated classes and were serialized they were not available for modification by end users.

In its upgraded form *oops* is able to perform lookahead of arbitrary depth making it a true LL(k) parser generator. It uses table lookup for parse decisions using the contents of the lookahead buffer. Therefore no alternative is preferred over another and unlike parsers that iterate through their decisions the worst case for a lookup is O(k) where k is the maximum depth of lookahead necessary to make a decision. In practice most decisions are made in constant time because even in grammars with high lookahead few decisions actually require it. These are the advantage of table lookups: greater runtime efficiency and equal treatment of alternatives. The typical disadvantage is a loss of expressibility. Most table driven LL parser generators accept a weaker form of input because of the loss of context provided by recursive descent. *Oops* does not suffer this weakness because once a decision is made it still uses recursive descent to execute that decision.

A new *JLex* scanner builder was introduced to make *oops* more practical as a parser generator. This kept with the general theme of upgrading *oops* even though it had the unfortunate consequence of removing interesting properties of self-compiled *oops* parsers. An option was provided, however, to use the old scanner generation system.

In spite of the volume of changes that were made to the inner workings of *oops* and the parsers it generates few alterations had to be made to the observer interface. In practice the interface changed a few times in trying to develop the most practical solution to receiving data. In the end the interface was altered to prohibit direct access to the parser and scanner from an observer. Instead *oops* observers now have all of the relevant parse data pushed to them. But in the context of the Law of Demeter these changes were entirely independent of the extensions to the functionality of *oops* itself. Modifications to the interface were minimal and most of the prior interface was maintained, albeit in a deprecated form.

The substantial changes were those made to the software that actually produces the parsers and the prefabricated classes that are used in the parsers themselves. The consequence is that the upgraded *oops* looks very similar to the original on the surface except that there is extended functionality.

It is hoped that this lookahead system and the broader design goals of the Law of Demeter will be applied to recursive descent LL parser generators. When the user is discouraged from modifying the parser generator output time is saved whenever the grammar or semantic code is altered and there is less opportunity for bugs to be introduced into the parser.

Chapter 5

Future Work

Implementation of the observer pattern in *oops* requires further development in three key areas:

1. Terminal symbols and their corresponding meanings should be included in the grammar.

2. Error detection and recovery requires callback options in the observers.

3. The scanner system requires better integration.

At present *oops* requires command-line input to decide what particular tokens represent. For example, the terminal symbol `"Identifier"` in a grammar is accompanied by the java parameter:

```
> java -Doops.pg.tokenizer.word=Identifier ...
```

This does not constrain the definition of the grammar to the grammar file and it would be better to include the meaning of a terminal symbol within the grammar itself. Work has been done on this in *oops3*[15] wherein one associates a regular expression with a terminal symbol. However, *oops3* is an LL(1) parser generator and it needs to be extended in order to provide the same functionality as *oops*.

The built-in error system also requires update both in itself and with respect to the interface. *Oops* currently detects errors but does not handle them very well. More importantly

the user has no control over how errors are reported nor does the observer interface have any means of learning that an error has occurred. Although the user should have no control over how errors are handled by the parser it is important that the user have control over the output generated by the language processor.

There are good error-handling systems available to LL parsers. Burke and Fisher developed a relatively non-intrusive LR method for error recovery that was convertible and applicable to LL parsers.[3] In their work an input text is parsed by a primary and secondary parser. The primary parser ensures the text is correct for the secondary (the secondary is used to decide what semantic actions are taken). In the primary parser there are three levels of recovery: (1) token addition, removal, replacement, or combining multiple tokens into a single one, (2) a segment of text is added or removed to close open scopes (parenthetical or otherwise bracketed statements), and (3) discarding text that precedes, follows, or surrounds an error token.

For *oops* to incorporate such a system, it would be necessary for the parser not to report the errors (as it does now) but to pass them to the observers. In other words, the observer interface should be expanded to include error-handling methods called by the parsers alerting them to what has been done. The interface would have to be sufficiently generic enough to work with any error-handling scheme. One possibility is the following callback methods:

```
void error (Terminal term, ParseData pd);
void removed (ParseData pd);
```

The `error` method would be called in place of the `shift` method so that it is known that what is passed is not necessarily equivalent to what was read by the scanner. The default `error` method (to be extended and overwritten by a user's observers) would report the error in the way the parser reports it now. It is similar for the `removed` method except that it does not replace a call to `shift` and only reports text that has been removed. Of course, a particular implementation that does not require addition, removal, or replacement of text need not call one or either of these methods. For example, the current system would

only ever require `removed`. But the possible means by which the observers can learn about an error are available.

Finally, *JLex* as a scanner generator is not well integrated with *oops* and probably will never be. One of the design principles of *oops* is as a self-contained language processing system that does not generate code but builds its parsers out of pre-existing classes. *JLex*, however, generates code and this means that *oops* has to give it proper input, produce the scanner code, compile it, load the class, instantiate it, and then serialize it, whereas with the rest of the parser the classes are available and it has only to instantiate and configure an object before serialization. Equally significantly *JLex* produces its scanners outside a compiler's jarfile making them difficult to load. Even when one is loaded it requires a class name distinct from that of the scanner being used by the existing parser generator or there is ambiguity. As mentioned in the **Implementation** section this removes the property that an input format can compile itself and produce a parser that is provably equivalent to itself.

Another possibility is a scanner generator much like the original one that configures table-driven scanners from a pre-written class on the fly. Such a scanner generator would be better suited to *oops*. Construction of the scanner would be more consistent with construction of the parser and *oops* would regain the self-compiling proof.

Bibliography

[1] Andrew W. Appel. *Modern Compiler Implementation In Java.* Cambridge University Press, second edition, 2002.

[2] Elliot Berk. JLex: A Lexical Analyzer Generator for Java. Available at http://www.cs.princeton.edu/appel/modern/java/JLex/ Retrieved Apr. 10, 2007, February 2003.

[3] Michael G. Burke and Gerald A. Fisher. A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery. *ACM*, pages 164–197, April 1987.

[4] Alfred V. Aho et al. *Compilers - Principles, Techniques, and Tools.* Addison Wesley, March 1988.

[5] J.W. Backus et al. Revised Report on the Algorithmic Language ALGOL 60. *Numerische Mathematik*, pages 420–453, December 1962.

[6] Etienne M. Gagnon and Laurie J. Hendren. SableCC, An Object-Oriented Compiler Framework. *TOOLS*, page unknown, August 1998.

[7] Stephen C. Johnson. *Yacc: Yet Anther Compiler-Compiler.* Bell Laboratories, Murray Hill, July 1975.

[8] Gerwin Klein. JFlex User Manual. Available at http://www.jflex.de/manual.html Retrieved Apr. 10, 2007, July 2005.

[9] Donald E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, pages 607–639, June 1965.

[10] Bernd Koehl and Axel-Tobias Schreiner. An Object-Oriented LL(1) Parser Generator. *SIGPLAN Notices*, December 2000.

[11] Karl J. Lieberherr and Ian M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE*, pages 38–48, 1989.

[12] Steven J. Metsker. *Building Parsers With Java*. Addison-Wesley, March 2001.

[13] Sun Microsystems. Java API for XML Processing (JAXP). Available at http://java.sun.com/webservices/jaxp/ Retrieved Apr. 10, 2007.

[14] T.J. Parr and R.W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software Practice and Experience*, pages 789–810, July 1995.

[15] Axel T. Schreiner and James E. Heliotis. Design Patterns in Parsing. In *Proceedings to EuroPLoP*, March 2007.

[16] ISO Standards. ISO/IEC 14977, 1996.

[17] Unknown. JavaCC Home. Available at https://javacc.dev.java.net/ Retrieved Apr. 1, 2007, April 2007.

[18] Unknown. SLK Parser Generator. Available at http://home.earthlink.net/ slkpg/ Retrieved Apr. 10, 2007, April 2007.

[19] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, Inc., 1976.

[20] Niklaus Wirth. What Can We Do About the Unnecessary Diversity of Notation for Syntax Definitions. *ACM*, November 1977.