A Computer Language Transformation System Capable of Generalized Context-Dependent Parsing

by

Adrian D. Thurston

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
December 2008

Abstract

Source transformation systems are special-purpose programming languages, or in some cases suites of languages, that are designed for the analysis and transformation of computer languages. They enable rapid prototyping of programming languages, source code renovation, language-to-language translation, design recovery, and other custom analysis techniques. With the emergence of these systems a serious problem is evident: expressing a parser for common computer languages is sometimes very difficult. Source transformation systems employ generalized parsing algorithms, and while these are well suited for the kind of agile parsing techniques in use by transformation practitioners, they are not well suited for parsing languages that are context-dependent. Traditional deterministic parser generators do not stumble in this area, but they sacrifice the generalized parsing abilities that transformation systems depend on. When it is hard to get the input into the system as a correct and accurate parse tree the utility of the unified transformation environment is degraded and more *ad hoc* approaches become attractive for processing input.

This thesis is about the design of a new computer language transformation system with a focus on enhancing the parsing system to support generalized context-dependent parsing. We argue for the use of backtracking LR as the generalized parsing algorithm. We present an enhancement to backtracking LR that allows us to control the parsing of an ambiguous grammar by ordering the productions of the grammar definitions. We add a

grammar-dependent lexical solution and integrate it with our ordered choice parsing strategy. We design a transformation language that is closer to general-purpose programming languages, yet enables common transformation techniques. We add semantic actions to our backtracking LR parsing engine and encourage the modification of global state in support of context-dependent parsing. We introduce semantic undo actions for reverting changes to global state during backtracking, thereby enabling generalized context-dependent parsing. Finally, we free the user from having to write undo actions by employing automatic reverse execution. The resulting system allows a wider variety of computer languages to be analyzed. By focusing on improving parsing abilities and moving to a transformation language that resembles general-purpose languages, we aim to extend the transformation paradigm to allow greater use by practitioners who face an immediate need to parse, analyze and transform computer languages.

Acknowledgements

I would like to thank my supervisor, James R. Cordy, for guidance during the production of this work. Jim was always available for discussions that left me with a fresh perspective. He has an amazing ability to relate new ideas to the huge and rather intimidating body of research that we have behind us. His ability to articulate my ideas in a manner that inspires confidence in my own work proved invaluable to me.

I would like to thank lab members Jeremy S. Bradbury and Chanchal K. Roy for many productive discussions about research and grad school life.

I am aware that in a lifetime it is possible to pursue interests that are easily shared with those you love. Like many grad students, this was not the case with me. My interests lie far from those of my family, yet my family always supported my personal obsessions and distractions. I would especially like to thank them for their love and support in my final year. Nobody is forced to write a thesis, and completing one takes a lot of hard work. Persistence in this kind of situation requires support from people who are willing to sacrifice while you chase after your own goals. My family did just that, and it is one of the many reasons why I am forever indebted to them.

Statement of Originality

I hereby certify that the research presented in this dissertation is my own, conducted under the supervision of James R. Cordy. Ideas and techniques that are not a product of my own work are cited, or, in cases where citations are not available, are presented using language that indicates they existed prior to this work.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Computer Language Transformation

A source transformation system is a programming language, or a suite of languages, designed for the analysis and manipulation of source code. A transformation system is characterized by language constructs for defining a grammar, and language constructs for defining a set of transformation rules. The user defines a grammar for the input language and the system derives a parser from it, then it uses the parser to build a syntax tree from the input. The user-defined transformation rules specify how the tree is to be manipulated. Rules contain a structured pattern and a structured replacement that both adhere to the grammar. The system also provides mechanisms for searching trees and applying the transformation rules. When the transformation is complete the result can be unparsed to the output stream.

Transformation systems are useful in core computer science activities and in the field of software engineering. Uses include programming language prototyping, programming language extension, program optimization, design recovery, program evolution, and language-to-language migration. Many other uses have been found over the years. Examples of transformation systems include TXL [25], ASF+SDF [97, 99], Stratego [107] and DMS [11].

A key requirement of a transformation system is that it automatically derive a parser

from the input language grammar. The parser is used for two purposes: to parse the input and to parse the structured patterns and replacements that make up the transformation rules. Using a single parser that has been derived from the grammar allows the structured patterns to be compared to fragments of the input. It also guarantees that the result of the transformation is in the language generated by the grammar, ensuring that the transformation is type-safe.

Though originally used for analysis and manipulation of source code, the utility of structured transformation is not exclusive to source code. Theoretically, a transformation system can be used to analyze or transform any data that has a structured tree representation. This includes data formats and communication protocols such as XML [17], HTML [81], DNS [70], SMTP [56], IMAP [26], HTTP [31], Internet Mail Messages [83] and log messages of various kinds. With the advent of the ubiquitous networked computer, the body of domain-specific languages and communication protocols that are needed to support our modern communication methods is growing. The Internet is built on protocols and data formats, where data is exchanged and stored in streams of bytes, but processed in structured representations. The need for tools that analyze and manipulate these data formats and communication protocols is as important as the need for tools that manipulate source code.

1.2 Motivation

1.2.1 Parsing

Transformation systems are not without problems, and perhaps the most serious of these is parsing ability. The difficulty of writing a grammar for some input languages can sometimes outweigh the benefits of using the transformation system. For example, C++ is very difficult to accurately parse using grammar-based techniques because its grammatical definition is simultaneously context-dependent and ambiguous. Since it is difficult to get a correct parse

tree on which the transformations will be based, the value of the transformation system is reduced.

In order to be useful for general-purpose transformation tasks, a parsing system must meet a number of requirements. The most important requirement is that the parsing algorithm be generalized. The system should not reject any grammar due to limitations of the parsing algorithm. In transformation tasks, it is common to use ambiguous grammars directly from language specifications, or to combine existing grammars when languages are mixed. These activities are possible when the parsing algorithm is generalized.

In some parsing problems, it is necessary to choose which lexical rules to apply based on surrounding grammatical context. For example, from within the value of a particular HTML tag attribute, perhaps the `onClick` attribute of a button, it may be desirable to parse JavaScript. In this specific context the JavaScript lexical rules must be applied, while using basic string parsing for other attribute values. One way to solve this problem is to use a scannerless generalized parsing system [84, 105]. Such a system includes lexical definitions at the leaf nodes of the grammar. In this capacity, the generated parser utilizes surrounding grammar context to decide which lexical rules to apply. The Stratego source transformation system employs this type of parser. As a result, languages that contain numerous regions with distinct lexical rules can be parsed naturally.

The final requirement, and that which causes transformation systems the most trouble, is that of context-dependent parsing. This refers to the ability to make decisions about the syntactic structure of the text being parsed based on the semantic interpretation of the text previously parsed. This typically involves analyzing the input text using logic custom to the input language and then using this information to assign extra meaning to the tokens being passed from the scanner to the parser. The extra information is used by the parser to direct its choices. This is a task that deterministic parsing systems such as the traditional Lex [61] plus Yacc [46] approach handle very well, but the generalized source transformation systems do not at all. The problem is that context-dependent parsing requires the maintenance of

global information, which is at odds with generalized parsing. The common approach when using a generalized parser is to defer the context-dependency restrictions to a post-parse disambiguation phase.

No existing grammar-based parsing system meets all three of these requirements. Worse, no system is capable of handling just the two requirements of context-dependent parsing and generalized parsing. The traditional Lex+Yacc approach can support grammar-dependent scanning and context-dependent parsing, but not generalized parsing. Source transformation systems easily support grammar-dependent scanning and generalized parsing, but lack support for context-dependent parsing. The primary goal of this work is to design a transformation system that meets all three of these requirements.

1.2.2 A New Transformation Language

In existing systems the transformation language depends on the parsing system. In our system this is true, but the reverse is also true. As our technique unfolds, we come to rely on the transformation language to enhance the parsing engine. Due to our design we must implement a new transformation system. We take this opportunity to also design a new transformation language and explore an approach to transformation language design that has not yet been taken.

Existing systems employ a rule-based programming paradigm that is different from general-purpose languages. Working in a transformation system requires shifting to this paradigm for all code that is written, including support libraries. While the transformation aspect is well taken care of, common programming tasks such as control flow and elementary algorithms sometimes have nontrivial implementations. One must take the time to figure out how to express solutions that seem easy to solve in general-purpose languages.

The problem becomes apparent when specialized programming problems creep into a transformation program as support code. Examples include the computation of a checksum specific to the application domain, the encryption/decryption of a sensitive data field,

the traversal of a directory structure, or the displaying of a dialog box. The more time a programmer must spend expressing otherwise straightforward algorithms as transformations, the more attractive it becomes to spend the time expressing the transformations in a general-purpose language.

The secondary goal of this work is to design a transformation language that is closer to general-purpose languages. We do not wish to give up the rewriting paradigm, but rather integrate core transformation language concepts with general-purpose constructs to produce a transformation language that appears familiar to programmers. Examples of essential transformation features are parsing at both run-time and compile-time, pattern matching, variable binding, tree construction, and tree traversal. By doing this we hope to design a system that appeals to programmers who are not familiar with rewriting systems.

1.3 Thesis Outline

In Chapter 2 we review the components of a transformation system, with a strong focus on generalized parsing techniques. The capabilities of our system will depend upon the choice of parsing method, therefore it is important to consider the characteristics of these methods in some detail. We then survey some existing transformation systems. Chapter 2 will give us the background necessary to identify the limitations of current systems and to develop a new system that advances the state of transformation systems.

In Chapter 3 we settle on the backtracking LR algorithm as our parsing method. Before we can employ this method in a source transformation system we must solve the issue of controlling the parse of ambiguous grammars. We develop a parsing strategy for backtracking LR that approximates the ordered choice strategy found in generalized recursive-descent parsers. We then discuss its limitations and show how the algorithm affects common grammar forms such as left-recursion and sequences. Finally, we describe our solution for defining languages that have varying lexical rules.

In Chapter 4 we give the design of our new transformation language, which we call COLM. We first outline the features taken from general-purpose languages, then move to the transformation-related features, such as pattern matching, tree synthesis, and the generic tree traversal constructs. We show our language is suitable for source transformation tasks by giving examples of generic traversals and an example of a goto-elimination program. Following that we give notes on some of the implementation choices that we have made.

In Chapter 5 we describe what we have done to allow our system to parse context-dependent languages. Deterministic parsing systems such as Lex and Yacc allow the user to write code that programmatically generates tokens and manipulates global data structures during parsing. The global data can then be used for feedback to the token-generation code. This is the approach we have taken, but rather than use some host programming language such as C for semantic actions and token-generation functions, we use the transformation language. To make this work in the context of backtracking LR, we address the issue of undoing changes to global state during backtracking. We introduce undo actions and make use of reverse execution to automatically generate them. This completes our transformation system and gives us a generalized context-dependent parsing system.

In Chapter 6 we demonstrate the ability of our system to parse language features that are difficult for other systems. We give examples of parsing problems that illustrate the grammar-dependent scanning, generalized parsing and context-dependent parsing abilities of our system. In some cases, these languages cause other transformation systems difficulty because of grammar-dependent scanning or context-dependent parsing requirements. In other cases, these languages cause deterministic parsing systems difficulty due to inherent ambiguities in the language specifications. The most notable example of a single common language that is difficult to parse using grammar-based techniques is C++.

Chapter 2

Background

2.1 Introduction

In this chapter we give a background for source transformation systems and generalized parsing. We begin by breaking the transformation system down into the essential components. We look at lexical analysis, context-free parsing, tree representation, tree traversal, structured pattern matching, tree synthesis and unparsing. We then survey existing generalized parsing algorithms, which will help us choose a parsing method to move forward with, in pursuit of our goals. Following the survey we analyze the methods with respect to the properties that we deem important in a transformation system. We cover the limitations imposed on the accepted grammars, the run-time complexity, the ability to control the parse of ambiguous grammars and the ability to recover from errors. We then move the discussion to context-dependent parsing. It is in this section that the primary motivation for this work becomes clear: we find that the current systems are insufficient for handling context-dependent languages. In the final section of this chapter we survey existing source transformation systems and give an example of a common algorithm that appears unfamiliar when expressed in a transformation system. We close by looking at the recent trend of extending general-purpose languages with select transformation features.

2.2 Anatomy of a Transformation System

A transformation system can be described as a programming language with a few particular characteristics. Where a general-purpose programming language might have integers, records, arrays and classes as the building blocks of its type system, a transformation system employs context-free languages for data-structure definition. The system then provides a parsing algorithm that is used to automatically derive the context-free structure of input text from the type definitions.

With parse trees in memory, the user can analyze and manipulate the input by traversing and rewriting trees. The transformation system provides language constructs for expressing tree searching, pattern matching, and tree synthesis. These constructs are explicitly designed for the traversal and manipulation of tree-based data structures. The user combines these basic operations to yield rewrite rules. Following a successful transformation the parse tree can be serialized.

In a transformation system the grammar has two purposes. It serves as the description of the language that the system's parsing algorithm will use to guide parsing. It also serves as the description of the program's data structures, against which the transformation program is type-checked. This dual use is what makes a user of a transformation system incredibly productive when writing programs that analyze and manipulate computer languages. The grammar is exploited for generating parsing code and for guiding programmers in their transformations.

In more *ad hoc* approaches, such as the use of Lex [61] plus Yacc [46], these two processes are separate. The user defines a grammar but it is used for parsing only. During the course of parsing (in semantic actions) the user builds up data structures using general-purpose data structure constructs such as classes, and then manually traverses these data structures with hand-written code. Once the input is in the general-purpose data structures, there is no opportunity for exploiting grammar structure for traversal function generation, deep

pattern matching, or transformation correctness checking.

The first purpose of the grammar, the declarative parser description, causes this research to move towards the study of algorithms for parsing context-free languages. It is desirable to be able to parse any reasonably useful computer language using a transformation system, and for the grammar-based parsing to be completely automatic. Unfortunately, these two desired features are in conflict because existing formalisms are often inadequate for writing declarative solutions to complex parsing problems. This highlights a need for parsing-related research.

There are many systems that implement some transformation concepts, but there are only a handful of systems that target the full parse - transform - unparse cycle. These are TXL [25], Stratego [107], ASF+SDF [97, 99] and DMS [11]. It could be useful to consider the features of the many other transformation-related tools; however we focus on the systems designed for the full cycle.

2.2.1 Tokenization

The traditional approach to tokenization is to treat the input as a fixed sequence of tokens by matching it against a collection of regular expressions. This approach is fast and simple. Lexical analysis can be performed ahead of time or on demand. Computing a fixed sequence of tokens is the approach of TXL.

Within the generalized LR (GLR) community there has been interest in scannerless GLR parsing [84, 105]. This method is used by the Stratego language. Each character is treated as a terminal and the tokens are defined using the context-free grammar. In many cases this creates ambiguities that must be resolved by disambiguation rules. These are declarative descriptions of illegal syntax forms that are allowed by the syntax definition, but should not be accepted by the parser. For example, a follow restriction on an identifier definition would specify that the characters `[a-zA-Z_]` are not allowed to follow an identifier, forcing the longest possible identifier token to be found. The downside of this approach is

that it is costly to treat each individual character as a node in the parse tree.

Van Wyk describes a deterministic technique called context-aware scanning [103]. When a new token is needed the parser passes to the scanner a list of tokens that are legal in the context of the current parser state. This set is derived from the grammar.

Automatically choosing scanning rules based on the grammar context is useful in a number of situations. Scripting languages often have distinct lexical regions in which very different lexical rules apply. The Ruby programming language [33] is an example of a scripting language that requires scanning rules that are dependent on the grammar rules active in the parser. For example, this is valid Ruby code:

```
v = v / 2 if 'foo' =~ /[foo]b*r/;
```

A forward slash is used to denote both division and the start of a regular expression. The naive scanner-based approach of entering a regular expression when the first slash is seen will not work. We instead need to instruct the scanner to recognize regular expressions only when the parser is able to accept them.

Multi-lingual parsing tasks also require an ability to handle multiple lexical regions, as does the parsing of many text-based Internet protocols. For example, Internet Mail Messages [83], HTTP [31] and IMAP [26] all require a tokenizer that can vary the active set of lexical rules.

2.2.2 Generalized Parsing

Transformation systems normally use a generalized parsing method in order to be useful for general-purpose transformation tasks. The advantage of a generalized parser is that it does not reject any grammar, whether it be for common computer languages that have ambiguous grammar definitions, a combination of grammars that arises from multi-lingual parsing, or a grammar modification that arises from agile parsing [27]. The need for generalized parsers in software engineering is argued by Brand et al. [102] and Aycock [3].

The C++ programming language is an example of a common language that has an ambiguous grammar that causes trouble for non-generalized parsing algorithms. The C++ standard [44] describes four ambiguities in the definition of the language.

In other cases, a generalized parser is needed because several grammars must be mixed for multi-lingual parsing. There are many scenarios when it is useful to embed one language in another. Examples include SQL in Java and Javascript in HTML. Mixing grammars can easily cause the limitations of deterministic parsing methods to be reached, but mixing is not a problem for a generalized parser.

The island grammar paradigm [71, 92] is another application of generalized parsers. Island grammars are useful for robust parsing and for multi-lingual parsing. The idea is that many parsing problems can be modeled as an island of interesting structure in a sea of uninteresting text.

The multi-lingual and island grammar paradigms stress the need for all features of the parsing method to work in the presence of ambiguities. For example, any facilities for handling grammar-dependent scanning rules or context-dependent name lookups must be usable in the presence of ambiguities, otherwise we may not be able to freely combine existing grammars without some potentially drastic modifications to the grammars.

There are a number of generalized parsing algorithms available. The choice of parsing algorithm has a direct impact on the type of parsing problem that can be specified, and by extension the number of languages the transformation system can be applied to. Since we aim to improve the parsing capabilities of existing transformation systems, we study generalized parsing methods in depth. A large section of this chapter (2.3) deals with this topic.

2.2.3 Representing Trees

Transformation languages have type systems built upon context-free grammars. A type is defined as a terminal with an associated text pattern, or a nonterminal with an associated

set of productions. In the term-rewriting community these are known as sorts.

Most systems use a value-semantics for the representation of trees at run-time. This raises the question of how to efficiently store trees in memory. It is common to use some form of tree sharing. Trees can be shared maximally using hash functions on trees [98], or they can be shared when it is convenient to do so, and it results in a measurable gain in performance. TXL uses the latter approach.

Another design decision is whether or not to store and operate on concrete syntax. The TXL programming language does this, whereas the languages based on term-rewriting generally operate on abstract syntax. Languages that operate on abstract syntax often have facilities for decorating trees with concrete syntax information such as layout and comments.

There is also the possibility of storing attributes in trees. A pure and simple tree-based transformation language would have just context-free right-hand sides and no attributes since only the context-free data is visible in the serialized output. It is convenient, however, to be able to decorate tree nodes with attributes and these may be added to the type definitions.

2.2.4 Tree Traversal

Tree traversal is a very important aspect of a transformation system. The Stratego language makes use of strategies for describing how trees are to be traversed. It allows one to specify traversals without mentioning the types that are traversed and therefore achieves a very high level of traversal abstraction. In other systems trees are explicitly traversed using program logic. In TXL there is some choice between the two. A functional style can be adopted and trees can be explicitly traversed. Alternatively, rules with a searching behaviour can be used to abstract traversal.

Some method for moving about trees must be devised and it must be made to work with the value semantics. Ideally, it allows the programmer to avoid duplicating common traversals for each type that is searched.

2.2.5 Tree Pattern Matching

In addition to being apt for searching for interesting types of trees, a transformation system must be suitable for testing the forms of trees.

The system should be able to take some string composed of types and concrete syntax and parse it into a partial parse tree, treating the literal types as terminal nodes. It should be able to test the parsed form against some input parse tree. In the process of testing it should capture the input trees that are matched to the types given in the pattern. These captured trees should become available as local variables.

Tree pattern matching gives us the ability to perform two tasks at once; test for particular forms of trees and capture the constituents of a tree without explicitly navigating parse trees. All existing source transformation systems are capable of this.

2.2.6 Tree Construction

Tree construction refers to the ability to synthesize new types from constituent trees. A transformation system should not require that the user explicitly specify the syntax of the trees constructed. Just as it can do with the parsing of patterns, it should be able to infer the structure by parsing a string composed of tree variables and concrete syntax, and use this partial parse tree to guide the construction of trees at run-time. This allows many intermediate steps of the tree building process to be conveniently skipped by the user because they are performed automatically, following the guidance of the system's parser.

2.2.7 Unparsing

The final step in the transformation process is unparsing. This would be a trivial task, save for the issue of laying out the tree with whitespace. If one wishes to preserve whitespace from the original program then the problem becomes harder.

A straightforward solution is to impose a computed layout on the entire output, thus

normalizing it and guaranteeing a self-consistent layout. However, the user can easily become unhappy about the fact that portions of the program that were untouched by the transformation system suddenly look different. A more ambitious approach is to attempt to preserve input formatting and integrate the layout of synthesized trees with the layout of the untouched trees [65, 59, 109].

2.3 Generalized Parsing

Generalized parsing algorithms allow the specification of parsers using arbitrary context-free grammars. In contrast with parsing algorithms that restrict the permitted language class, such as LALR(1) [28] and LL(1) [62], generalized parsing algorithms take ambiguous language definitions in stride. There has been a recent resurgence in the study of generalized parsing algorithms. This is partly due to the fact that modern hardware can support super-linear parsing methods. It is also due to the fact that current applications warrant their use.

Generalized parsing algorithms are important in the field of natural language processing, where the languages to be parsed are heavily ambiguous. They are also important for the parsing of computer languages. For example, many modern programming languages such as C++ contain ambiguities, albeit usually fewer than the natural languages. It is the computer language application of generalized parsing that we are concerned with.

The problem of generalized parsing has been solved by a number of parsing algorithms. Backtracking recursive descent, backtracking LR, generalized LR (GLR) [94] and the Earley algorithm [29] are the most prevalent approaches. Many of these methods have numerous variations, especially GLR, and some of these methods have disguised incarnations. For example, many parsing methods are, in fact, an implementation of generalized recursive descent. It is also possible to take the extreme position that there are just two types of parsing methods, top-down and bottom-up, and all other variations are just implementation

details.

This section will examine the research in generalized parsing techniques. First we introduce the parsing methods with a discussion about the two main types, top-down and bottom-up. In Sections 2.3.2 through 2.3.7 we describe the research in generalized parsing as we work through all the major algorithms. In Sections 2.3.8 through 2.3.11 we evaluate the techniques with respect to practical considerations. In the end we should have a better understanding of the space of generalized parsing methods as well as have gained some insight into which parsing methods suit the needs of the transformation system.

2.3.1 Classification of Parsing Strategies

Different parsing methods often have characteristics in common. One of those characteristics is the method's approach in constructing the parse tree. In the study of parsing methods it is useful to consider whether a method is top-down or bottom-up. A top-down method attempts to match the input to the grammar by considering the derivations of a grammar nonterminal. It asks the question which production right-hand sides can be derived from this left-hand side? It starts with the goal symbol and moves down to the leaf nodes. A bottom-up method starts with the leaf nodes and considers reductions to nonterminals. It asks the question which nonterminals do these right-hand-side elements reduce to? It proceeds by moving from leaf nodes upwards to the goal symbol.

Of all the parsing methods discussed in this chapter, every one except Earley parsing can be considered strictly top-down or bottom-up. Earley parsers are unusual in that they contain properties of both strategies.

It is also useful to consider how the method consumes the input. The most common case is left-to-right online parsing. This is due to the fact that in many parsing tasks the input is not available all at once, and it usually arrives in sequence from left to right. Though this is the most common case, other methods of consuming the input are discussed in the literature. Non-directional methods, which consider the input as a whole, and right-to-left

parsers are also possible.

Top-Down Parsing

In top-down parsing the possible grammar derivations are recursively explored beginning at the root production. Possible productions of a nonterminal are selected and the right-hand sides are explored. As the right-hand sides are traversed, any nonterminals that are found are recursively explored. Leaf nodes are compared to the tokens in the input stream. When a match is made the parser shifts over that input item and proceeds to match further on down a production's right-hand side.

Top-down parsers often require that grammars be free of left-recursion because it causes the parser to enter into an infinitely recursive exploration of the grammar that does not consume any input, only going deeper and deeper into exploring the grammar. Consider the following example grammar for arithmetic expressions.

```
E → T + E          T → F * T          F → ( E )
  | T - E            | F / T            | id          (Grammar 2.1)
  | T                | F                | num
```

Right-recursion is used instead of left recursion. Use of right recursion solves the infinite exploration problem, however it means that if the resulting parse trees are recursively evaluated with a bottom-up evaluation then sequences of addition and subtraction operations or sequences of multiplication and division operations will be evaluated from right to left. Though inconvenient, this is not a fatality. It is quite simple to rewrite the resulting parse trees into a left-recursive form or to adapt the evaluation.

As an alternative approach, it is possible to modify a top-down parsing method to support left-recursion [37]. The TXL programming language employs a recursive-descent parser. The run-time system of the parser is able to recognize when it is attempting to parse a left-recursive construct and it modifies its behaviour accordingly. In simplified terms, it temporarily emulates a bottom-up parser.

It is also possible to transform a grammar to factor out left recursion, leaving a grammar that a top-down system can use naturally. Lohmann et al. [63] give a modern treatment of this technique. This is less desirable in the context of transformation systems because it modifies the user's types.

Figure 2.1 illustrates a top-down parsing approach. In the left column the stack of explored productions is shown, beginning with the root. A level of indentation is used to indicate that a group of lines have a common parent. The dot symbol inserted in the right-hand side of a production is used to indicate how far into the production the parser has explored. If the dot immediately precedes a nonterminal, then the next level explores this nonterminal.

In the right column the unparsed stream of input tokens is shown. As the parser consumes an input token the dot in the production on the top of the stack is advanced. When a production is fully matched it is popped from the top of the stack. Square brackets are used to indicate the input symbols that have been matched to a fully parsed nonterminal.

Top-down parsing methods differ by how they decide which production to attempt to parse and by the action they take when a parse error is encountered. For example LL(1) [62] parsers, which are not general, look one token ahead to try and decide which production to attempt to parse. Non-general methods will usually give up when an error is encountered or implement some limited form of backtracking. They attempt to parse some subset of all possible derivations of the grammar, whereas a generalized top-down method will try all possible derivations.

Bottom-Up Parsing

In bottom-up parsing the stream of input is analyzed for possible instances of production right-hand sides. When a right-hand side is found it is replaced with its corresponding nonterminal. The parser then uses the found nonterminal in its search for more matches of production right-hand sides.

Stack	Input
E→.T+E ? T→.F ? F→.id	x + (3 + 4) * y - 2
E→.T+E ? T→.F ? F→id[x].	+ (3 + 4) * y - 2
E→.T+E ? T→F[x].	+ (3 + 4) * y - 2
E→T[x].+E	+ (3 + 4) * y - 2
E→T[x]+.E	(3 + 4) * y - 2
E→T[x]+.E ?	
E→.T-E ?	
T→.F*T ?	
F→.(E)	(3 + 4) * y - 2
F→(.E)	3 + 4) * y - 2
F→(.E) ?	
E→.T+E ?	
T→.F ? F→.num	3 + 4) * y - 2
T→.F ? F→num[3].	+ 4) * y - 2
T→F[3].	+ 4) * y - 2
E→T[3].+E	+ 4) * y - 2
E→T[3]+.E	4) * y - 2
E→T[3]+E[4].	) * y - 2
F→(E[3+4].)	) * y - 2
F→(E[3+4]).	* y - 2
T→F[(3+4)].*T	* y - 2
T→F[(3+4)]*.T	y - 2
T→F[(3+4)]*.T ? T→.F ? F→.id	y - 2
T→F[(3+4)]*.T ? T→.F ? F→id[y].	- 2
T→F[(3+4)]*.T ? T→F[y].	- 2
T→F[(3+4)]*T[y].	- 2
E→T[(3+4)*y].-E	- 2
E→T[(3+4)*y]-.E	2
E→T[(3+4)*y]-.E ? E→.T ? T→.F ? F→.num	2
E→T[(3+4)*y]-.E ? E→.T ? T→.F ? F→num[2].	
E→T[(3+4)*y]-.E ? E→.T ? T→F[2].	
E→T[(3+4)*y]-.E ? E→T[2].	
E→T[(3+4)*y]-E[2].	
E→T[x]+E[(3+4)*y-2].	
E[x+(3+4)*y-2].	

Figure 2.1: A visualization of a top-down parsing approach used to parse the string `x+(3+4)*y-2` according to Grammar 2.1. The left column contains the stack of explored productions. The right column contains the unparsed input.

Consider the following grammar, which is identical to Grammar 2.1 except the productions have been made left-recursive and the resulting parse trees can therefore be recursively

Stack	Input
	x + (3 + 4) * y - 2
E[x]	+ (3 + 4) * y - 2
E[x]	+ (3 + 4) * y - 2
E[x] +	(3 + 4) * y - 2
E[x] + (	3 + 4) * y - 2
E[x] + (E[3]	+ 4) * y - 2
E[x] + (E[3] +	4) * y - 2
E[x] + (E[3] + T[4]	) * y - 2
E[x] + (E[3+4]	) * y - 2
E[x] + (E[3+4])	* y - 2
E[x] + F[(3+4)]	* y - 2
E[x] + T[(3+4)]	* y - 2
E[x] + T[(3+4)] *	y - 2
E[x] + T[(3+4)] * F[y]	- 2
E[x] + T[(3+4)*y]	- 2
E[x+(3+4)*y]	- 2
E[x+(3+4)*y] -	2
E[x+(3+4)*y] - T[2]	
E[x+(3+4)*y-2]	

Figure 2.2: A visualization of a bottom-up parsing approach used to parse the string `x+(3+4)*y-2` according to Grammar 2.2.

evaluated in their natural form. Figure 2.2 illustrates how a bottom-up parser would proceed in parsing the string `x+(3+4)*y-2`. The left column contains elements that have been parsed. This particular bottom-up approach is called LR [58]. It is looking for rightmost derivations among the matched input symbols and nonterminals in the left column.

```
E → E + T        T → T * F       F → ( E )
  | E - T           | T / F         | id          (Grammar 2.2)
  | T               | F             | num
```

LR style bottom-up parsing has enjoyed a considerable amount of exposure over the years. Notably, the Yacc [46] program and its clones are in wide use today. Bottom-up parsing permits one to construct a deterministic automaton for the purpose of driving the parse. The exact nature of the automaton depends on the variant of the table construction method that is used. Some possibilities are SLR(1) [28], LALR(1) [28], LR(1) [58]. These can vary in table size and in the amount of lookahead encoded in the parse tables, but

for the most part they all have the same basic structure. They are driven by two types of actions encoded in the transitions of the automaton: shift and reduce actions.

If the grammar meets the criteria of the automaton construction method (for example LALR(1)) then the actions will be free of conflicts. However, if the grammar does not, which is always true for ambiguous grammars, then the automaton will contain shift-reduce or reduce-reduce conflicts. Some parser generators may choose one action by default and produce code that runs, but does not necessarily work as intended. Other generators may simply fail to proceed. Some allow one to manually resolve conflicts by assigning priorities to grammar constructs, or explicitly stating the correct action to take in the event of a particular conflict. The general methods, on the other hand, will emit a parse table regardless of conflicts. The problem is handled at run-time either by backtracking, or by concurrently pursuing multiple possible parses.

In the next section we begin discussing the generalized parsing methods, starting with one of the earliest top-down methods, Unger's Method.

2.3.2 Unger's Method

The first parsing method we discuss is Unger's Method [95]. It is a greedy, non-directional, top-down parsing method. It works by searching for partitionings of the input that match the right-hand side of production rules. When a plausible partitioning is found it recurses on the nonterminals and the corresponding matched text. If all nonterminals return a match then the partitioning succeeds. The parsing problem then becomes an issue of matching the entire input string against the root production.

For example, the input string `a b a b` has three partitionings (not including those with empty bins) when checked against the production `S` → `AB b`. The last of these succeeds. To verify that the production matches the string, the method must now recurse on the string `a b a`, checking all partitionings of it against all productions of `AB`.

For small examples this method works quite well. However, the amount of required

E			
E	-	T	
x	+	(3+4)*y-2	
x	+(	3+4)*y-2	
x	+(3	+4)*y-2	
x	+(3+	4)*y-2	
⋮	⋮	⋮	
x+	(	3+4)*y-2	
x+	(3	+4)*y-2	
⋮	⋮	⋮	
x+(3+4)*y	-	2	*

E			
E	+	T	
x	+	(3+4)*y	*
x	+(	3+4)*y	
⋮	⋮	⋮	
x+(3	+	4)*y	*
⋮	⋮	⋮	
x+(3 +4)	*	y	

Figure 2.3: The partitioning of an arithmetic expression for matching against an expression grammar in Unger's method.

processing time quickly grows for larger, more practical parsing problems. Consider the partitionings shown in Figure 2.3. The input string `x + (3+4)*y - 2` has 165 partitionings when checked against the production `E` → `E - T` and it is the last one that succeeds as a possible match (indicated by an asterisk in the last column). The method then must recurse on the text matched to `E`. It is only 2 elements shorter and the check has 84 partitionings. With this example it is easy to see that Unger's method quickly becomes a very impractical parsing method.

To improve the situation Unger proposed a number of optimizations. These are in the form of checks that rule out partitions that can never match. An obvious optimization is to pursue a partition only when the terminals in the partition match the terminals in the production being compared against. It is interesting to note that Unger's method can be expressed as a search problem. The search can be attempted in a depth-first manner or a breadth-first manner. Unger suggests depth-first. Improving performance can then be viewed as pruning the search space.

The description of the algorithm so far does not include support for productions that derive the empty string. Unfortunately, a naive implementation of Unger's method has

problems with such epsilon productions. The difficulty is that if a production derives the empty string, the entire string may be passed to a neighbouring production. If that neighbouring production recurses on the original production without ever shortening the string, an infinite exploration occurs. This is similar to what happens when a top-down method explores a left-recursive grammar, only the problem is more general. It can occur with other forms of recursion. The solution is to add in checks that detect infinite recursion and cut off the search. This is described in detail by Grune [40].

The following ambiguous grammar will serve as our running example through all the parsing methods. It does contain epsilon productions; however it does not give Unger's Method any trouble. In future sections it will sometimes be expressed as right recursive to aid in the discussion of other top-down methods.

```
S → AB b              AB → AB a
  | a A a AB              | AB b        (Grammar 2.3)
A → A a                   | ϵ
  | ϵ
```

The input string in the running example will be: `a a b a`. Though the grammar is ambiguous, there is only one parse of this input string. There are two benefits of employing generalized parsing methods. The first is that it allows us to work with ambiguous grammars. The second is that it allows us to parse strings that require infinite lookahead, irrespective of whether or not we are dealing with an ambiguous grammar. This input string illustrates the latter case. It is an example of the type of input that requires infinite lookahead to parse properly. To grammar programmers this represents freedom from parse table conflicts and is a huge gain whether or not they are dealing with an ambiguous language definition.

Figure 2.4 illustrates Unger's Method using our example grammar and input string. Again, an asterisk in the last column is used to indicate a possible match of a partition. Following these matches, the algorithm then explores the nonterminals in the partition and the corresponding matched text to determine if the partitioning is a real match.

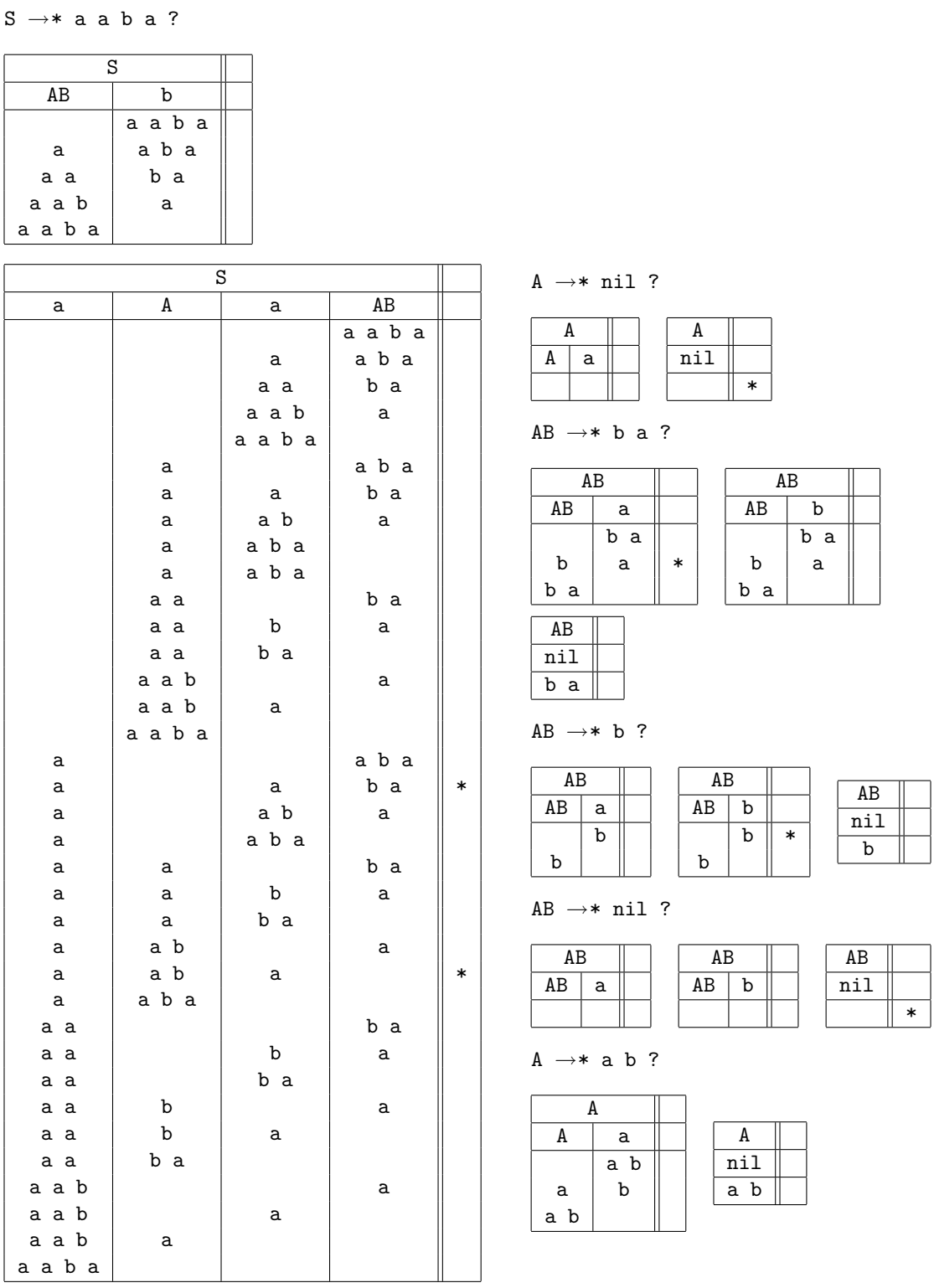

Figure 2.4: A visualization of Unger's Method parsing `a a b a` using Grammar 2.3.

In the next section we discuss another historically significant algorithm, the CYK method. It is significant because it gives us a polynomial algorithm for parsing arbitrary context-free grammars. Where Unger's Method has an exponential upper bound, CYK gives us a cubic upper bound.

2.3.3 CYK Method

The CYK parsing method is a bottom-up parsing method discovered independently by Cocke [22], Younger [112], and Kasami [55]. Grune [40] provides a modern treatment of CYK parsing. CYK parsers consider which nonterminals can be used to derive substrings of the input, beginning with shorter strings and moving up to longer strings.

The algorithm starts with strings of length one, matching the single characters in the input string against unit productions in the grammar. It then considers all substrings of length two, looking for productions with right-hand-side elements that match the two characters of the substring. This process continues up to longer strings. At each length of substring, it searches through all the levels below, that is, all the matches of the substrings of the substring.

The CYK algorithm assumes the grammar to be in Chomsky Normal Form (CNF), which simplifies things. What follows is Grammar 2.3 transformed into CNF.

```
S → AB    BL            ALAL → AL   AL          AB → b
  | AL    AL            ALA  → AL   A           BL → b
  | ALAL  AB            ALAB → AL   AB          S  → b      (Grammar 2.4)
  | ALA   AL            A    → A    AL          AB → a
  | ALA   ALAB          AB   → AB   AL          AL → a
                              | AB   BL          A  → a
```

CNF restricts each production to either having exactly two nonterminal elements on its right-hand side, or to being a unit production deriving a terminal. Epsilon productions are not allowed, except when derived immediately from the start symbol. The unit productions are used in the first round of matching where substrings are of length one. The two-element productions are then used in all following rounds. Because each production has only two

elements we need only consider partitionings of the substrings into two further substrings.

The complete CYK algorithm is given in Figure 2.5. As the algorithm proceeds, it enters values into a two-dimensional array. Each element in the array contains sets of nonterminals. These sets indicate which nonterminals can derive the substring of the input described by the location in the array. The column indicates the position of the substring in the input and the row indicates the length of the substring. In the example that follows in Figure 2.6, the shortest substring is positioned at the bottom row.

The body of the algorithm consists of three levels of loops. The outer loop iterates over the lengths of substrings. This takes us up in the array. The next loop iterates over offsets of substrings. This takes us across the array. The innermost loop iterates over the partitions of the substrings. Each substring is partitioned into two parts. The value used in the partitioning loop gives the length of the first half of the partition. Then within this innermost loop, the CYK algorithm considers all productions of the form `N` $\rightarrow$ `T1 T2`. If the current partitioning of the current substring has T1 in its first half and T2 in its second half then N is added to the set in array cell $P[o, l]$.

The algorithm given in Figure 2.5 is the greedy, non-directional version. It starts by considering the entire input string at shorter substrings, working up to longer substrings. It is possible to implement CYK as an online version, where instead of iterating over the columns beginning on the bottom row (the shorter substrings) the traversal runs diagonally beginning in the bottom left corner and going up to the left, progressively moving more to the right (and hence consuming input in an online manner).

We now begin to discuss Earley parsing. Earley parsing has stood the test of time. It is still in use today and modern implementations can be found. An advantage that Earley parsing has over CYK is that it does not require the grammar to be in CNF.

```
σ1...σn is the input string.
P[n,n] is an array of sets of nonterminal items.  These are initially empty.
for each o = 1 ..  n
  for every nonterminal N such that N -> σo add N to P[o,1]
for l = 2 ..  n
  for o = 1 ..  n-l+1
    for p = 1 ..  l-1
      for all N -> T1 T2
      where T1 member of P[o,p] && T2 member of P[o+p,l-p]
      add N to P[o,l]
if P[1,n] contains the start symbol then parsing succeeds
```

Figure 2.5: The CYK parsing algorithm.

```
length = 2
    o: 1 p: 1 adding: S -> AL AL
    o: 1 p: 1 adding: ALAL -> AL AL
    o: 1 p: 1 adding: ALA -> AL A
    o: 1 p: 1 adding: ALAB -> AL AB
    o: 1 p: 1 adding: A -> A AL
    o: 1 p: 1 adding: AB -> AB AL
    o: 2 p: 1 adding: S -> AB BL
    o: 2 p: 1 adding: ALAB -> AL AB
    o: 2 p: 1 adding: AB -> AB BL
    o: 3 p: 1 adding: AB -> AB AL
```

```
length = 3
    o: 1 p: 1 adding: ALAB -> AL AB
    o: 1 p: 2 adding: S -> AB BL
    o: 1 p: 2 adding: S -> ALAL AB
    o: 1 p: 2 adding: AB -> AB BL
    o: 2 p: 1 adding: ALAB -> AL AB
    o: 2 p: 2 adding: AB -> AB AL
length = 4
    o: 1 p: 1 adding: ALAB -> AL AB
    o: 1 p: 2 adding: S -> ALAL AB
    o: 1 p: 3 adding: AB -> AB AL
```

↑ substring length	AB ALAB S			
	AB ALAB S	AB		
	A AB ALA ALAB ALAL S	AB ALAB S	AB	
	A AB AL	A AB AL	AB BL S	A AB AL
	a	a	b	a
		substring offset →		

Figure 2.6: The CYK algorithm applied to Grammar 2.4 and the input string `a a b a`.

2.3.4 Earley Parsing

Earley parsing [29] is a bottom-up method that bears a strong resemblance to LR parsing. Earley parsers are somewhat more dynamic, however. Rather than represent the state of the parser with a finite automaton and a stack, the entire state is represented by a list of sets of parser state items.

Earley parsers can also be thought of as a top-down method due to the fact that during parsing they consider which production right-hand sides to attempt to match by expanding nonterminals in the current item set. In a sense an Earley parser uses a top-down method to predict what should be matched in a bottom-up way. The Earley parsing process is very similar to the process of constructing an LR automaton and then running it, only the Earley parser constructs the automaton as states are needed, whereas an LR parser generator constructs the entire automaton ahead of time, then it is used to drive the parse.

Earley parsers work with what are called Earley items. An Earley item is a production augmented with a marker inserted at some point in the production's right-hand side and a number to indicate where in the input the matching of the production began. The marker indicates how much of the production has been recognized already. To the right of the marker are the terminals and nonterminals that must still be recognized in order for the production to be accepted.

$\texttt{P} \rightarrow \alpha \bullet \beta \ @n$

At run-time, an Earley parser creates a list of Earley item sets. Each set in the list corresponds to a character in the input stream. A movement from one set to the next represents a movement over an item in the input stream.

Earley item sets are constructed by applying three operations to the current list of Earley item sets. In the simple version of the algorithm, these operators will only ever need to modify the set at the head of the list, but will need to consult sets further back in the list when adding to the set at the head. We call the set at the head S_p and the next set back S_{p-1}.

The first of these operators is the scanner. It is responsible for advancing the parser over literal tokens in the input. We refer to the token in the input stream that corresponds to the item set at the head of the list as σ_p. The scanner searches the items in S_{p-1} and for any item with the dot immediately preceding the current token $\texttt{P} \rightarrow \alpha \bullet \sigma_p \beta \ @n$, we

advance the marker over σ_p and add the item to S_p.

The second operator is the predictor. It searches S_p for items where the marker is immediately ahead of a nonterminal $\texttt{P} \rightarrow \alpha\bullet\texttt{N}\beta$. The predictor then adds all productions $\texttt{N} \rightarrow \bullet\alpha$ @p to S_p. The predictor is responsible for setting the pointer p.

Where the scanner traverses the grammar from left to right, and the predictor traverses the grammar top-down, the completor is the operation that traverses the grammar bottom-up. For every item in S_p with the marker at the end to indicate the production has been fully matched $\texttt{P} \rightarrow \alpha\ \bullet$ @n, the completor searches the itemset S_n for items with the marker immediately preceding an instance of the nonterminal $\texttt{Q} \rightarrow \alpha\bullet\texttt{P}\beta$ @m. It advances the marker over the nonterminal and adds the item to S_p.

Figure 2.7 shows the item sets an Earley parser would construct as it parses the string `a a b a` using Grammar 2.3. In the final set the item $\texttt{S'} \rightarrow \texttt{S}\bullet$ @0 is present, indicating a successful parse.

The version of Earley's algorithm described thus far does not include any lookahead. The performance of Earley parsers can be improved by adding lookahead to the items. This reduces the number of items added by the completor. Lookahead requires that we be able to modify item sets further back from the head of the list and revisit the computation of those sets.

Like Unger's Method and the CYK algorithm, the simple version of Earley's algorithm is only a recognizer. It answers the question of whether or not a particular string is in the language. Further work must be done in order to produce a parse tree. Pointers can be maintained while items are added to keep track of which items derive which. These links can then be used to yield a traversal of the parse tree with predictors going down, scanners going across and completors going up.

Earley parsing can be made fast with a number of compile-time optimizations. McLean and Horspool [67] fuse an Earley parser with an LR parser to obtain what is called LRE

```
set S0:
    S' -> . S          @ 0  // start
    S -> . AB b        @ 0  // predictor
    S -> . a A a AB    @ 0  // predictor
    AB -> . AB a       @ 0  // predictor
    AB -> . AB b       @ 0  // predictor
    AB -> .            @ 0  // predictor
    S -> AB . b        @ 0  // completor
    AB -> AB . a       @ 0  // completor
    AB -> AB . b       @ 0  // completor

character a, set S1:
    S -> a . A a AB    @ 0  // scanner
    AB -> AB a .       @ 0  // scanner
    A -> . A a         @ 1  // predictor
    A -> .             @ 1  // predictor
    S -> AB . b        @ 0  // completor
    AB -> AB . a       @ 0  // completor
    AB -> AB . b       @ 0  // completor
    S -> a A . a AB    @ 0  // completor
    A -> A . a         @ 1  // completor

character a, set S2:
    AB -> AB a .       @ 0  // scanner
    S -> a A a . AB    @ 0  // scanner
    A -> A a .         @ 1  // scanner
    S -> AB . b        @ 0  // completor
    AB -> AB . a       @ 0  // completor
    AB -> AB . b       @ 0  // completor
    AB -> . AB a       @ 2  // predictor
    AB -> . AB b       @ 2  // predictor
    AB -> .            @ 2  // predictor
    AB -> AB . a       @ 2  // completor
    AB -> AB . b       @ 2  // completor
    S -> a A a AB .    @ 0  // completor
    S -> a A . a AB    @ 0  // completor
    A -> A . a         @ 1  // completor
    S' -> S .          @ 0  // completor

character b, set S3:
    S -> AB b .        @ 0  // scanner
    AB -> AB b .       @ 0  // scanner
    AB -> AB b .       @ 2  // scanner
    S' -> S .          @ 0  // completor
    S -> AB . b        @ 0  // completor
    AB -> AB . a       @ 0  // completor
    AB -> AB . b       @ 0  // completor
    S -> a A a AB .    @ 0  // completor
    AB -> AB . a       @ 2  // completor
    AB -> AB . b       @ 2  // completor

character a, set S4:
    AB -> AB a .       @ 0  // scanner
    AB -> AB a .       @ 2  // scanner
    S -> AB . b        @ 0  // completor
    AB -> AB . a       @ 0  // completor
    AB -> AB . b       @ 0  // completor
    S -> a A a AB .    @ 0  // completor
    AB -> AB . a       @ 2  // completor
    AB -> AB . b       @ 2  // completor
    S' -> S .          @ 0  // completor
```

Figure 2.7: A trace of an Earley parser as it parses the string `a a b a` using Grammar 2.3.

parsing. This uses the states of an LR table in place of the production and marker component of Earley items. The backpointer is retained as a dynamic component. Each state in the automaton represents a collection of dot items from an Earley parser. Using these LR style tables eliminates the predictor component from the dynamic computation. Predictions are instead pre-computed as they are inherent in the tables. LRE does not use a

single stack and can therefore still be used to parse ambiguous grammars.

Aycock and Horspool [6] give a technique for producing directly executable Earley parsers. Backpointers remain as dynamic entities. However, the actions to take on each state are static and can be generated as directly executable code rather than as table-based data and a driver. Earley items can then be represented as a pointer to a block of code that implements the action for the state, and a corresponding backpointer into the stream of input. Earley sets are encoded as lists of these pairs. This technique relies on the target language supporting first class labels. That is, it must be possible to assign a label to a variable, then later jump to the destination pointed to by the variable using a goto statement, the same way a function pointer can be invoked.

Aycock and Horspool [5] give a further refinement of the LRE parsing method. It is observed that some states in the LR automaton constructed for Earley parsing will always appear together due to epsilon productions requiring immediate completion. The states that always appear together can be merged, producing a new type of automaton that yields very high performance for Earley parsers. The extra run-time work that epsilon productions cause for a standard Earley parser is eliminated by enhancing the predictor with an instantaneous complete for epsilon productions. With this new type of automaton as much work as possible is moved to the static computation. This work can therefore be considered the last word on improving the speed of Earley parsers.

We now turn our focus to generalized top-down parsing and its many variants and associated implementations.

2.3.5 Generalized Top-Down

The generalized top-down parsing method consists of a recursive interpretation of a grammar coupled with some method for handling nondeterminism. The most common approach is to search the possible parses using backtracking. This yields a single parse. It is also possible to implement the list-of-successes approach, which yields all parses.

Generalized Top-Down with Full Backtracking

Generalized top-down parsing with full backtracking is a very flexible parsing method because it makes it easy to control the parsing of ambiguous grammars. When provisions are made for handling left recursion a wide range of parsing tasks can be implemented. The parsing algorithm used in TXL is a generalized top-down parsing method with full backtracking and support for left-recursion. The success of TXL shows that this method is very useful in language design and software renovation tasks.

In this parsing method there are two modes. The parser will attempt to parse in the forward direction until a parse error occurs. At this point it will locate an alternative decision to make and begin to unparse until it reaches that choice point. It will then resume forward parsing.

Forward parsing consists of recursively interpreting the grammar in the manner described in Section 2.3.1. When a nonterminal is found, the first associated production is fetched and the parser begins to move over its right-hand-side elements. This portion of the algorithm is somewhat straightforward and easy to implement. The most important part of the algorithm is in deciding what to do when an error occurs and it is time to backtrack. A proper implementation will yield an ordered choice interpretation of the grammar.

When it is time to backtrack the parser will search the current parse tree for the rightmost and innermost choice point with respect to the parse tree obtained thus far. The parser must then undo parsing until the choice point is at the head of the search space. It then chooses the next alternative and resumes forward parsing.

Grammar 2.5 shows our running example grammar in right-recursive form. Figure 2.8 shows how a generalized top-down parser would proceed to parse the string `a a b a`. It first exhausts all the possibilities in the first production of `S` then afterwards attempts the second production of `S`.

Parse Tree	Input
forward	
S	⊥
AB . b	
a AB	
a AB	
b AB	
a AB	
ϵ	
error, backup	
S	a ⊥
AB . b	
a AB	
a AB	
b AB	
ϵ	
error, backup	
S	b a ⊥
AB . b	
a AB	
a AB	

Parse Tree	Input
forward	
S	a ⊥
AB b .	
a AB	
a AB	
ϵ	
error, backup	
S	a b a ⊥
AB . b	
a AB	
ϵ	
error, backup	
S	a a b a ⊥
AB . b	
ϵ	
error, backup	
S	a b a ⊥
a . A a AB	

Parse Tree	Input
forward	
S	b a ⊥
a A . a AB	
a A	
ϵ	
error, backup	
S	a b a ⊥
a A . a AB	
ϵ	
forward	
S	b a ⊥
a A a . AB	
ϵ	
forward	
S .	⊥
a A a AB	
ϵ b AB	
a AB	
ϵ	

Figure 2.8: Generalized recursive-descent parsing of the string `a a b a`, according to Grammar 2.5.

```
S → AB b          AB → a AB
  | a A a AB           | b AB      (Grammar 2.5)
A → a A               | ϵ
  | ϵ
```

A key advantage of this method is that the ordered choice interpretation of the grammar puts the user in control of the parsing strategy when the grammar is ambiguous. The preferred order in which to attempt to parse mutually ambiguous alternatives can be specified locally. This is useful for grammar composition and specialization tasks in source code analysis [27]. The innermost backtracking strategy makes it easy for the user to predict the result of the parse. The primary disadvantage of this parsing approach is that it can result in very long parse times. Full backtracking often induces redundant reparsing when grammar alternatives contain common prefixes [18].

Definite Clause Grammars

A definite clause grammar (DCG) [78] is a syntactic shorthand for producing parsers with Prolog clauses. Prolog-based parsing is a very expressive parsing technique that can be considered a generalized top-down parsing method. In the DCG paradigm, the Prolog parsing clauses represent the input with difference lists: two lists with the first containing the input to parse (a suffix of the entire input string) and the second containing the string remaining after a successful parse. Removing the suffix indicated by the second list from the first list yields the tokens that a clause successfully parsed. These two lists correspond to the input and output variables of the clauses. Each clause corresponds to a nonterminal in the grammar. Prolog's backtracking guarantees that all possible grammar derivations are tested. Prolog clauses may be embedded in grammar productions acting as syntactic or semantic predicates.

Combinatorial Parsing and the List of Successes

Combinatorial parsing is a method of producing generalized recursive-descent parsers in functional languages. The idea was originally discussed by Burge [19] as an interesting example of a functional programming application. The method employs functions in a recursive-descent style and gets its name from the fact that function combinators are used to compose parsers. The two basic function combinators are the sequencing operator and the choice operator.

The list of successes method from Wadler [110] is used. Each parsing function, again corresponding to a nonterminal in the grammar, returns a list of pairs. Each pair represents a successful parse. It consists of a parse tree and a pointer into the input indicating where the remaining input begins. Applied in a recursive manner, this returns all possible parses without any explicit backtracking. If done in a lazy functional language, and at the end of the parse the head of the list is requested, then the lazy functional language will take

a backtracking approach to evaluating the first parse on-demand. Combinatorial parsing closely resembles generalized top-down parsing with full backtracking.

Hutton [43] gives an accessible description of the method. It is shown that combinatorial parsing is a very powerful method because parsing is tightly integrated into the host functional language and it is therefore easy to extend parsing with arbitrary program logic. This is a feature that combinatorial parsing shares with Prolog-based parsing.

Combinatorial grammars have also been integrated with other backend parsing methods. The method described above has exponential behavior and in some applications this is crippling. Vijay-Shanker and Weir [104] give a polynomial-time parsing algorithm for combinatorial grammars based on the CYK parsing algorithm. As a reminder, CYK parsers ask the question which nonterminals accept this substring? They answer it by considering which nonterminals accept the substrings of the substring. This connection between functional-style parsing and a bottom-up parsing method illustrates an interesting point: bottom-up methods can be considered top-down methods with the recursion statically eliminated.

Getting back to the list of successes approach, it is interesting to note that it has also been applied outside of a functional programming setting. In the GRDP method described by Johnstone [47], all possible parses are returned by an invocation of a parsing function. This eliminates the need to backtrack. Johnstone cites this method as appropriate for prototyping parsers. He supports the argument that generalized parsing methods are ideal for the parsing of computer languages. However, in practical terms work is needed to improve performance. When it comes time to produce a production parser, the GRDP algorithm can be optimized by taking advantage of a property of some grammars called follow-determinism. When a grammar exhibits this property, improved performance is attained by a pruning of the search space. However, it is at the expense of generality.

Recursive Descent with Memoization

Packrat parsing [35] attempts to solve the problem of redundant reparsing that arises in generalized top-down parsers with full backtracking. Packrat parsing can be classified as generalized top-down parsing with memoization and ordered choice in a lazy functional language setting.

As the algorithm parses in the forward direction, each time a nonterminal is successfully parsed the parse tree is recorded along with the position in the input where the parse occurred. Should the backtracking system be invoked and the question of whether or not nonterminal N can be parsed at position p be asked again, then the answer is readily available and reparsing is avoided.

In the Packrat literature there are claims that the underlying formalism on which Packrat parsing is based, the parsing expression grammar (PEG) [36], is a distinct formalism for representing languages. The PEG is, however, very similar to a sugared CFG with an ordered choice interpretation strategy. Since applying ordered choice to yield a single parse instead of acquiring all parses is a property of the parsing algorithm and not of the language formalism, it is difficult to agree that PEGs are a new kind of formalism.

The Packrat parsing literature also claims that linear time parsing is achieved for any grammar. This is quite a strong claim, and it relies on large amounts of memory usage. The memory usage is so large in fact that it can quickly defeat the speed claim in small practical cases. For example, a 40 KB Java source file requires upwards of 20 MB of heap space to parse. Experimental research suggests that memoization actually hurts measured performance [12].

ANTLR

ANTLR [77] is a parsing and rewriting framework based on top-down parsing technology. It is a notable tool because it has a long history and a large user base. Like other top-down

tools, it also focuses on managing the tradeoff between parsing power and performance in generalized top-down parsing. Parsers generated by ANTLR normally have the LL(k) property, with k > 1. An upper bound on the amount of lookahead can be explicitly set or k can be allowed to roam. When k roams this is referred to as LL(*) parsing. This allows many grammars that arise in practice to be parsed and is often sufficient. For cases in which it is not, ANTLR is able to revert to a generalized top-down method. This is a full backtracking mode with memoization.

This concludes our discussion of generalized top-down parsing. We now shift our attention to bottom-up parsing that is generalized due to backtracking.

2.3.6 Backtracking LR

Backtracking LR parsing is a bottom-up method that extends deterministic linear time LR-based parsers with the ability to handle any context-free grammar. While a deterministic LR parser will encounter conflicting shift and reduce actions and emit an error when a grammar does not meet the LR criterion, a backtracking LR parser will generate a functioning parser regardless. Upon encountering a conflict, the run-time system will try an action, remember the choice that was made, and continue parsing in a single thread. Later on, should a parse error be encountered it will undo its parsing up to the most recent choice point, then try the next possibility. Such a strategy will eventually exhaust all possible parses.

While a standard top-down parser with full backtracking will revert to the rightmost, innermost choice point with respect to the parse tree acquired so far, a backtracking LR parser will revert to the rightmost, topmost choice point with respect to the LR stack. This algorithm on its own cannot guarantee an ordered choice as is the case with generalized top-down. In Chapter 3 we present a solution to this problem.

The primary advantage of backtracking LR parsers is that they retain the speed of deterministic LR parsers in places where the grammar is deterministic. Furthermore, if backtracking can be kept to a minimum, the cost of processing nondeterministic grammars

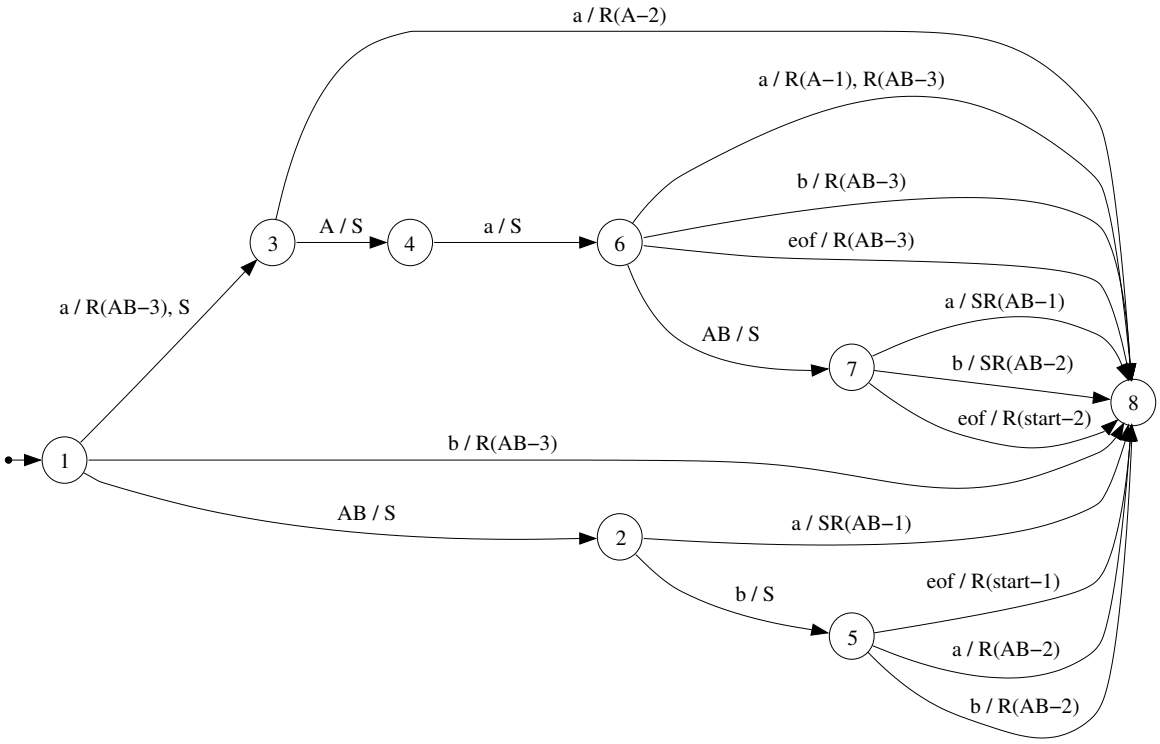

Figure 2.9: LALR(1) parse tables generated for Grammar 2.3. These are used in our backtracking LR and GLR parsing examples.

need not be prohibitive. Merrill [69] describes changes made to Yacc to support backtracking for the purpose of parsing C++. Power and Malloy [79] also suggest using a backtracking LR parser for the initial deployment of a C++ grammar due to the language's incompatibility with standard LR parsing.

Figure 2.9 shows the LALR(1) parse tables generated for Grammar 2.3. There are two conflict points. Figure 2.10 shows the behaviour of the backtracking LR parser on the string `a a b a`. It first pursues the possibility of matching the first production of S, eventually fails, then backs up to the beginning of the string and moves forward with the second production of S, which it accepts.

Action	Stack	Input
		a a b a ⊥
reduce	AB(n:2) ϵ	a a b a ⊥
shift	AB(n:2) a ϵ	a b a ⊥
reduce	AB AB(n:2) a ϵ	a b a ⊥
shift	AB a AB(n:2) a ϵ	b a ⊥
reduce	AB AB a AB(n:2) a ϵ	b a ⊥
shift	AB b AB a AB(n:2) a ϵ	a ⊥
reduce	AB AB b AB a AB(n:2) a ϵ	a ⊥
shift	AB a AB b AB a AB(n:2) a ϵ	⊥
ERROR		
unshift	AB AB b AB a AB(n:2) a ϵ	a ⊥

Action	Stack	Input
unreduce	AB b AB a AB(n:2) a ϵ	a ⊥
unshift	AB AB a AB(n:2) a ϵ	b a ⊥
unreduce	AB a AB(n:2) a ϵ	b a ⊥
unshift	AB AB(n:2) a ϵ	a b a ⊥
unreduce	AB(n:2) a ϵ	a b a ⊥
unshift	AB(n:2) ϵ	a a b a ⊥
unreduce		(next:2) a a b a ⊥
shift	a	a b a ⊥
reduce	a A ϵ	a b a ⊥
shift	a A a ϵ	b a ⊥
reduce	a A a AB ϵ ϵ	b a ⊥
shift	a A a AB b ϵ ϵ	a ⊥
reduce	a A a AB ϵ AB b ϵ	a ⊥
shift	a A a AB a ϵ AB b ϵ	⊥
reduce	a A a AB ϵ AB a AB b ϵ	⊥

Figure 2.10: The complete parsing of the string a a b a, according to Grammar 2.3 using a backtracking LR approach.

In the next section we address the final parsing method covered in this chapter, generalized LR parsing. This large and interesting topic has been the subject of a considerable amount of research in both natural language processing and computer language processing.

2.3.7 Generalized LR

Like backtracking LR parser generators, a generalized LR (GLR) parser generator will accept any grammar and will always emit a working parser. The generated parser will handle conflicts at run-time by forking its state and pursuing all parses in lockstep. Since parsing in lockstep requires multiple stack instances, much research has gone into managing a shared stack that conserves space and computation time, making the approach much more practical.

The underpinnings of generalized bottom-up parsing were given by Lang [60]. Theory was developed, though it was not shown to work in a practical sense. Tomita introduced GLR as we know it today [94]. In this work the concept of a graph-structured stack (GSS) was described and an algorithm that can be implemented with a reasonable amount of effort was given.

The critical innovation in GLR parsing is the GSS. Without it, the forking of parsing threads leads to exponential behaviour. The GSS is constructed and managed at run-time. As parsing proceeds, parse table conflicts induce splits in the stack. This causes the stack to branch out into a tree form. Each unique path that starts at the head of the stack and leads back to the root constitutes a unique stack. Later on, paths may converge to the same LR state. When the paths converge to the same state, they can be physically merged at their head. This reduces the number of unique stack heads, even though distinct paths through the GSS still exist and therefore distinct parse threads still exist.

Merging reduces the number of live stack heads and therefore improves performance, though the gain may be temporary. Merging sometimes needs to be undone. This is because each reduction in LR parsing requires that a number of elements be popped from the stack.

Action	Path Event	Stacks	Input
red AB, shift		AB 2 a 8	
	split	1	a b a ⊥
shift		a 3	
red AB, shift		AB 2 a 8	
		1	b a ⊥
red A, shift		a 3 A 4 a 6	
red AB, shift		AB 2 b 5	
		1	a ⊥
red AB, shift		a 3 A 4 a 6 AB 7 b 8	
red AB, shift		AB 2 a	
	merge	1 8	⊥
red AB, shift		a 3 A 4 a 6 AB 7 a	
die			
	separate		
red AB, shift		1 a 3 A 4 a 6 AB 7 ⊥ 8	

Figure 2.11: Visualization of a GLR parser parsing `a a b a` using Grammar 2.3.

If the elements popped include a merge point, then the stacks must first be split and the reduction performed once for each distinct sequence of elements that need to be reduced.

For our GLR parsing example we again use the LALR(1) parsing tables in Figure 2.9, which are built for Grammar 2.3. Figure 2.11 shows the behaviour of a GLR parser on the string `a a b a`. In this example a number of operations are combined into one. This helps to illustrate the lockstep nature of the algorithm. Both the state numbers and language elements are shown in each stack. The stack is initially split at the first shift/reduce conflict on the first character `a`. It is merged once when both stacks arrive in state 8. However, the merge must be separated in the following step when `AB` is reduced.

Tomita parsers can fail to terminate on grammars that contain epsilon productions. Hidden left-recursion and hidden right-recursion both cause Tomita's algorithm problems, which is why Tomita specified his parser as only for epsilon-free grammars. This situation was improved by Farshi [74] who provided an algorithm that works for all context free grammars. Nederhof [73] also solves the problem of hidden recursion by removing epsilon productions as the parser progresses. Two alternate and faster solutions are provided

by Scott et al., called Right Nulled GLR (RNGLR) [89] and Binary Right Nulled GLR (BRNGLR) [90] parsing.

GLR parsers have special requirements when parse trees need to be acquired. Since GLR parsers acquire all possible parses it is useful to be able to build a parse tree forest (consisting of all the possible parses) in a manner that uses as little memory as possible. Billot and Lang [13] show that there is a shared forest structure with at most cubic size for any parse of a context-free grammar. This result was applied by Rekers [82] to Farshi's algorithm to yield what is known as the shared packed parse forest (SPPF). This is the method used by the parsing and rewriting languages ASF+SDF and Stratego.

Recently there has been interest in improving the performance of GLR parsing by attempting to reduce the amount of stack activity, since this is where much of the parsing time goes. This was first explored by Aycock and Horspool [4], where the approach is to 'flatten' a context-free language into a regular language, to the extent permitted by the amount of true recursion inherent in the grammar. True recursion refers to recursion that cannot be rewritten as iteration. Finding the minimal number of locations in a grammar where use of the stack is unavoidable is an NP-hard problem and thus heuristics are used. The disadvantage of this parsing method is the impractical state table sizes produced by interpreting large portions of a context free grammar as a regular language. Despite this impracticality, a number of research papers have explored the concept [8, 87, 48, 49], with the most recent by Scott [88] resulting in a completely general algorithm that applies to all grammars. Speedups are achieved (a factor of 2-10 depending on the source); however table sizes are measured in the millions of rows as opposed to the hundreds of rows for standard LALR(1)-based GLR parsing. Practical applications of this method are not found.

In the GLR field, there has been recent work on making empirical comparisons of GLR algorithms. The Grammar Toolbox [51, 53, 52] has been applied to the problem of pitting the many variants of GLR parsing against one another.

This concludes our description of the generalized parsing methods. In the next section

we begin our critical analysis of the methods. Our ultimate goal is the design of a new transformation system; therefore our analysis is from the viewpoint of a system designer in search of an appropriate general method.

2.3.8 Limitations on Grammars Imposed by the Algorithms

Some of the methods discussed in this chapter are not completely general, but are instead near-general. In this section we discuss the limitations imposed by the parsing method on the grammars.

A naive implementation of Unger's method does not deal well with epsilon productions. Some grammars induce infinite recursion. The problem can be solved by adding a data structure that records which partitions are currently under consideration. An explicit check of this data structure will detect when fruitless infinite exploration of the grammar is about to happen and the search can be cut off. With this enhancement Unger's Method is able to handle all grammars and is therefore completely general.

The CYK algorithm requires that grammars be in Chomsky Normal Form. Any grammar in CNF is supported and there is a transformation from any context-free grammar into CNF. Therefore the CYK algorithm is said to be completely general. The user must be prepared, however, to deal with a parse tree returned in CNF form, or must map it back to the original grammar.

The Earley algorithm is a completely general parsing algorithm that works with any grammar in its natural form. Epsilon productions only cause problems of efficiency, not of termination. Recent work on Earley's algorithm [5] addresses the issue of efficiency caused by allowing epsilon productions.

Generalized top-down methods do not support left-recursive grammars. In practice this is either handled by avoiding such grammars and opting for right-recursive versions, by grammar transformations that eliminate left-recursion, or by *ad hoc* modifications to the parsing algorithm to support left recursion. A more serious problem with generalized

top-down methods is that some grammar forms cause exponential parsing behaviour. In practice these grammars must be avoided, otherwise parse times will be unacceptable.

The backtracking LR approach cannot handle hidden left recursion. In practice this is not normally a problem. Hidden left recursion can usually be factored out of a grammar in a minimally disruptive way. There is nothing explicitly desirable about handling hidden left recursion, other than handling it when it inadvertently arises. As is the case with generalized top-down methods, some grammar forms cause exponential parsing behaviour. Again, it is up to the user to avoid these grammars.

The original GLR algorithm given by Tomita exhibits problems with hidden left and hidden right recursion. The variations of Tomita's algorithm given by Farshi [74] Nederhof [73] and Scott [89] are all completely general. There are no restrictions on the grammars.

While using a generalized parsing algorithm allows us to cope with ambiguous grammars and we consider this to be a good thing, there is a downside to this situation. Sometimes we wish to design a grammar free from ambiguities and the deterministic parsing methods imply a conservative ambiguity detection. By using a generalized method we give up that ability. Schmitz [85] argues that we should explicitly add ambiguity detection back to our generalized parsing methods. Ambiguity detection is an undecidable problem [41]. However, a conservative approximation that returns false positives but no false negatives is possible [86].

2.3.9 Time Complexities

The asymptotically fastest general algorithm is the Valiant parsing technique [96]. It achieves a worst case of $O(n^{2.376})$ by applying boolean matrix multiplication to the parsing problem. Unfortunately, there is a large constant factor that is quite prohibitive. This fact is characteristic of analyzing the speed of parsing algorithms. Though a particular algorithm may be worse than another asymptotically, on real grammars it may perform much better. Though both Earley parsing and GLR parsing have the same asymptotic

Method	**Upper Bound**
Unger	$O(d^n)$
CYK	$O(n^3)$
Earley	$O(n^3)$
Generalized Top-Down	$O(d^n)$
Packrat	$O(n) + O(M)$
Backtracking LR	$O(d^n)$
GLR Tomita	$O(n^3)$
GLR Farshi/RNGLR	$O(n^{p+1})$
BRNGLR	$O(n^3)$

Figure 2.12: The asymptotic time complexities of the generalized parsing algorithms. The variable n is the length of the input, p is the length of the longest right-hand side, d is some value > 1, and M is the cost of managing $n \times |P|$ cached parse trees, with $|P|$ giving the number of productions.

performance, when deciding between the two it is advisable to use GLR to parse grammars with low ambiguity and Earley to parse grammars with high ambiguity [102].

Asymptotically, backtracking parsers can exhibit exponential behaviour. This is the primary reason that the authors supporting GLR recommend avoiding backtracking. Johnstone [50] shows that it is easy to write a grammar that exhibits exponential behaviour when given to a backtracking LR parser generator. Users must themselves guard against producing poorly performing grammars. In practice this is viable both with generalized top-down parsing and backtracking LR. Also, Prolog-style cuts can often be used to improve performance by pruning alternative parses before they are attempted. With small amounts of localized ambiguity and careful grammar construction, backtracking parsers can perform very well. The table in Figure 2.12 gives the asymptotic behaviour of most of the parsing methods described so far.

When making practical speed comparisons among parsers there are many factors to consider, the most significant being the nature of the grammar. Others include engineering

requirements such as whether or not a parse tree must be built. A grammar with productions that have few alternatives and a lot of right recursion can be parsed very quickly using a generalized top-down method. If the number of alternatives increases and more common prefixes are introduced, LR-based parsers behave better. When ambiguity is low and backtracking is kept to a minimum, a backtracking LR parser is a suitable choice. As ambiguity increases a GLR parser will do better due to the lower asymptotic bound. As ambiguity increases further, Earley parsers outperform GLR parsers.

2.3.10 Controlling the Parse of Ambiguous Grammars

Generalized parsers provide the user with an ability to specify ambiguous grammars. When this facility is available, the user may specify any grammar desired without being concerned about the dreaded shift-resolve conflict or implementing some lookahead trick. However, it introduces a new problem, that of deciding which possible parse to accept when a string has several possible parses. Simply accepting them all is one way to approach the problem. However, it is not a terribly useful approach considering that yielding a single parse is the usual requirement of programming language processors.

In generalized top-down parsing methods with full backtracking a single parse is always produced and predicting the outcome of the parse is easy for the user. Given any two mutually ambiguous productions, the first will be returned if a parse exists, otherwise the second will be returned. The user controls the parse by adjusting the order of productions. Though Unger's Method is not applied in practice and the issue of controlling the parse is not discussed in the literature surveyed, it too could be made to implement an ordered choice strategy.

A backtracking LR parser will also always yield a single parse tree; however there is no grammar-based facility for controlling the parse. We address this problem in the following chapter.

The effects of ordered choice can be achieved in a GLR setting. The method depends

on the GLR implementation. In the case of tools like Stratego, which produce a complete parse forest, this can be done by post processing of the returned parse forest. This has performance implications. Acquiring all parses only to discard all but one is less than optimal.

The Elkhound GLR parser generator [68] does not build a parse forest by default. Instead, it executes semantic actions that the user can optionally use to build a parse tree. It also invokes callback functions that can be used to resolve ambiguities. The `merge` function, which is written by the user, is executed in the case of nonterminals that have more than one parse. This can be used to choose one production over another. Unlike previous GLR systems, Elkhound guarantees a bottom-up execution order for semantic actions and merges, allowing the callback technique to work properly.

CYK and Earley parsing both require that a parse tree be constructed following a successful parse. As in GLR parsers, the algorithm that traverses the data structures built by the recognizer can be made to choose the parse tree that an ordered choice strategy would have produced.

2.3.11 Error Recovery

Related to the problem of controlling the parse is that of recovering from errors to resume parsing. Since generalized parsers will accept any grammar, a particular kind of error handling becomes very convenient. If we build productions that consume input into the grammar, with the intent that they will be applied when no other can be applied, then we have an error recovery technique that fits naturally into the parsing algorithm.

This approach works well for generalized parsers based on backtracking with ordered choice. We can define productions that accept any token except the one which defines where error recovery is to stop. With the ordered choice guarantee in place we know that the error production will be accepted only when all others fail. The following error handler consumes input until the input stream and parser go back into a stable state where correct parsing

may resume.

```
statement → for_block
          | while_block
            ⋮
          | not_semi_list ;
```

This technique does not work very well with parsing methods that attempt to acquire all possible parses simultaneously because error productions will be needlessly accepted at every statement (or other unit of parse on which error productions are defined). These parse trees will be constantly acquired only to be later discarded. In the case of non-backtracking methods, error recovery is a technique that must be implemented outside the parsing algorithm when the last potential parse dies off. Many approaches are possible, and some appropriate external method can be implemented. An example is the method by Corchuelo [24] for LR parsers.

2.3.12 Summary

In Figure 2.13 we give a summary of the parsing methods discussed in Section 2.3. The methods are clustered into the major algorithm categories and correspond to the topics in Sections 2.3.2 through 2.3.7.

Each row of the table summarizes a characteristic of the methods. First the top-down/bottom-up classification is given. The next row indicates which methods are directional. Non-directional methods require the entire input to be in memory at once. Directional methods scan through the input, usually left to right. The next row indicates which methods are completely general and the one following that indicates which methods can be applied to a grammar in its natural form. These topics are discussed in Section 2.3.8. The run-time complexities of the algorithms are given on the following row. These are discussed in Section 2.3.9. The next row indicates whether or not the algorithms implicitly construct a parse tree. In the case of GLR, the SPPF fulfills this purpose, though it

	Unger's Method	CYK	Earley	Gen Top-Down	Backtracking LR	Generalized LR
Strategy	top-down	bot-up	both	top-down	bot-up	bot-up
Directional			✓	✓	✓	✓
100% General	✓	✓	✓	no *	no †	✓
Grammar Unchanged	✓		✓	✓	✓	✓
Complexity	$O(d^n)$	$O(n^3)$	$O(n^3)$	$O(d^n)$	$O(d^n)$	$O(n^3)$
Tree Built by Alg	✓	post	post	✓	✓	optional ‡
Run-Time Space	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$	$O(n^3)$‡
Controlling the Parse	✓	extern	extern	✓	✓	extern
Error Recovery	✓	extern	extern	✓	✓	extern
Modern Applications			✓	✓	✓	✓

∗ left-recursion

† hidden left-recursion

‡ building shared packed parse forest (SPPF)

Figure 2.13: Summary of algorithm characteristics.

is optional. In the next row we give the run-time space requirements of the algorithms. In the case of GLR, this refers to the space requirements of the SPPF. Next the facilities for controlling the parse and recovering from errors are given. These are described in Sections 2.3.10 and 2.3.11. Finally, the last row indicates if modern applications of the methods can

be found.

2.4 Context-Dependent Parsing

Source transformation systems employ generalized context-free parsers to allow grammars to be easily composed. If computer languages encountered in practice were within the boundaries of the context-free languages, then the story would end here. We would select a generalized method that appeals to us, possibly GLR, and build our transformation system around it.

Unfortunately, many computer languages used in practice are not context-free. The reality is that languages are implemented with hand-written parsers, or with grammar-based parsers that have been augmented with hand-written code. The language features enabled by these *ad hoc* parsing methods creep into the design of languages. The result is that many languages we use today require the *ad hoc* techniques in order to be parsed properly, leaving the pure context-free parser unable to handle real languages in practice.

2.4.1 Global State

To parse common programming languages such as C, Java, C#, and C++ requires that our parsing method support context dependencies. For example, we often need to look up the meaning of an identifier in a symbol table before we can proceed to parse it. This is handled in deterministic parsing settings by using lexical feedback. While the input is parsed, semantic actions build and maintain a symbol table that is read by the scanner and used to alter the types of tokens.

This feedback model can be generalized to other kinds of context dependencies. The idea is for the semantic actions to feed information back to the earlier lexical stage by way of maintaining global data structures. The earlier stage of the parse uses the information it finds in global data structures to manipulate the tokens sent to the parser.

2.4.2 Nondeterminism and the Global State

Since transformation systems employ generalized parsers, we must consider the issue of access to the global state in the presence of nondeterminism. The central question is: can we have a global state that we use for lexical feedback or other purposes, and at the same time gain all the benefits of a generalized parser?

Reading from a single global state in the presence of nondeterminism is not a problem. This generalizes to the problem of supporting concurrent readers. If we can periodically (perhaps on statement boundaries) reduce to a single possible parse and restrict modifications to the global state to these times, then we don't have a problem. GLR parsing systems can be configured to do this. However, as is the case in concurrent and distributed algorithms, the real difficulty arises when we need to write to the global state in the presence of nondeterminism. This is a problem that we find arises in practice and for which there is no solution, short of resorting to hand-written parsers.

C++ is an example of a commonly used programming language that is very difficult to parse with grammar-based tools. This is because C++ has an ambiguous grammar definition and relies on a context-dependent interpretation of identifiers. This situation creates a need to manipulate global information during the parse of ambiguous constructs. The following example illustrates one area of the language where this is true.

```
namespace ns
{
    struct A { };
};

template <class T> struct C
{
    struct D {};
    static const int i;
};

template <class T> const int C<T>::i = 1;
```

In this program fragment two classes are defined. Class `C` is a template type that provides

some generic code. Class `A` will be used as an argument type. Now consider the following statement.

```
int j = sizeof( int( C<ns::A>::i ) );
```

This C++ code sets the variable `j` to the size of an expression. The expression is an anonymous integer initialized using the static integer variable `i`. Assigning the integer the value in `i` is not necessary. However, according to the C++ standard this is legal code and it must be accepted regardless. Now consider this next statement.

```
int i = sizeof( int( C<ns::A>::D ) );
```

This is the size of an anonymous function declaration, not the size of an expression. Anonymous functions are indeed types in the C++ language. The standard states that, in the case of this ambiguity, any program text that can be a type (determined by looking up identifiers) is a type, otherwise it is an expression. Therefore this program text must be parsed as the size of a type. These two code blocks are covered by the following ambiguous productions in the C++ grammar.

```
unary_expression → 'sizeof' ( type_id )
                 | 'sizeof' unary_expression
```

Before successfully parsing either of these statements the parser must determine the type of language object that the identifiers `D` and `i` represent. This is where the difficulty arises. In order to do this we must parse `C<ns::A>::` and store the class it resolves to in some global data structure that the scanner can reference when it looks up `D` and `i`. Now we have to accommodate ambiguities and the manipulation of global data structures.

The problem only gets worse the deeper one goes into the language. Since template arguments may be arbitrary value expressions, and templates may be overloaded with the appropriate template chosen by the compiler using the list of type arguments (called template specializations), we need a reasonable implementation of the type system before a correct parse tree can be acquired.

2.4.3 Requirements

There are three requirements that a generalized parsing method must meet in order to effectively utilize the feedback mechanism for parsing context-dependent languages.

- Semantic actions must be executed immediately following a reduction. When a token is passed to the parser, any reductions caused by the token should also induce semantic actions. This allows updates to the global state to take effect immediately.
- Distinct parses must be able to manipulate distinct instances of the global data. Different interpretations of an input string must not be forced to utilize the same global data instance for posting feedback data.
- The tokenization routines must be aware of the nondeterministic nature of the global data structure and manipulate tokens accordingly. For example, if two parses are pursued simultaneously and each has led to a distinct global data structure value, then the subsequent token manipulations must be performed twice, once using the first value as a source and again using the second.

In a backtracking setting, meeting the above three requirements is natural, given the assumption that it is possible to revert changes made to the global state when backtracking. Since backtracking parsers are always pursuing a single parse at any given time, all that is needed is a single state that can be updated and reverted as the parser goes forward and backward.

In parsing methods that simultaneously pursue multiple parses, meeting these three criteria is considerably more difficult. Not only are there high costs associated with keeping multiple copies of the global state, the idea conflicts with the natural sharing of data that gives the non-backtracking generalized parsers their advantage. For example, suppose a GLR parser diverges and because of different semantic actions it yields two different global states. Later the stacks may converge if they end up in the same parse state. If this

happens then we have a converged parsing thread but two distinct global states that are used to translate tokens. This situation is problematic. We must instead prevent stacks from converging when they are associated with distinct global states, thereby losing the polynomial-time advantage that GLR has over backtracking methods.

The difficulty of this situation has given rise to an alternate solution in a GLR setting. The common approach to parsing ambiguous context-dependent languages with a GLR parser is to attempt all possible parses irrespective of context dependencies. The tokenizer yields tokens that simultaneously carry multiple types [7] and the GLR parser will acquire all possible interpretations. Following the parse, or at strategic points, disambiguation rules [57, 101, 100] eliminate the parse trees that are illegal according to context dependency rules. This approach has been shown to work when applied to C++ [68].

An argument against this approach is that it requires one to acquire parse trees that will later be thrown away. This is true not only of valid parse trees that are lower in priority than others, it is also true of invalid parse trees that are created due to the fact that tokens cannot be immediately disambiguated. Tokens are sent to the parser as impossible types and are used to construct impossible trees despite the fact that information that can prevent this is available. It is encoded in the previously parsed text.

Similar problems exist with Packrat parsers. These parsers do not acquire all possible parses so there is no option to defer to a post-parse disambiguation. Instead, a global state must be maintained. If a semantic action needs to manipulate global state then any memoized results that have been stored for positions at or ahead of the current parsing position must be thrown out since they were created using a global data structure that is different from the current global state. Forward memoized results must also be thrown out when the parser backtracks over a semantic action that has modified the global state.

In the Rats! packrat parsing system the approach is to use lightweight, nested transactions [39] that are implemented using a manually programmed stack of global data structures. This technique is not general. The stack-based approach to storing global data

requires that alternative parses at the same input position modify the global state in the same way.

2.5 Source Transformation Systems

In this section we give an overview of source transformation systems. Many programming languages and systems have been used for transforming source code. We restrict ourselves to languages with a type system derived from context-free grammars, an integrated parsing method for acquiring parse trees, and a programming language designed primarily for rewriting these trees.

Common among all systems is that a transformation program is divided into a grammar specification and a set of transformation rules. The grammar specifies syntactic forms of the language and the rules specify how the input is to be manipulated. The processing of input is divided into three phases: parsing of the input text into a structured tree, transformation of the parse tree, and unparsing of the new tree to the output text.

2.5.1 ASF+SDF

ASF+SDF [97, 99] is a source transformation system derived from term rewriting systems. ASF is the algebraic specification formalism, which provides the term rewriting facilities. SDF is the syntax definition component of the system. It includes a scannerless generalized LR parsing engine. SDF allows the user to resolve ambiguities using priority and associativity attributes in grammar definitions. Any remaining ambiguity must be resolved by disambiguation filters following the parse.

The rewrite rules specify how to make reductions, also known as simplifications. In term rewriting this refers to matching a pattern and making a replacement. Each rule is composed of a simple equation. On the left-hand side is the pattern to match and on the right-hand side is the replacement. The left-hand side may bind variables that can be used

on either side of the equation. A variable used on the left-hand side will test for equality between the variable and the other input tree fragment. A variable used on the right-hand side will copy the bound variable to the replacement tree. Once a set of term rewriting rules have been declared, program transformation becomes a problem of simplifying terms until no more rules can succeed and a normal form is achieved.

While common in other rewriting applications, the repeated application of all rules to a parse tree until a normal form is achieved is not always the appropriate strategy for program transformation. An ability to specify to which sections of a tree certain rules should be applied is needed. This can be achieved in ASF+SDF by first defining a new type that encapsulates the types we wish to modify. We then target our rewrite rules to this new type. When we wish to cause the rules to be applied at a specific location, we construct the new type in that location with the value to be modified as a child. By placing new types in the tree we can invoke rules by name.

To further control the rewrite process, ASF+SDF allows conditions to be attached to rules. Conditions can be used to refine a pattern or to construct values that are used in the replacement. These rule labels and conditions are useful for controlling the rewrite process, however they are not completely general. Traversals other than the inbuilt top-down and bottom-up traversals must be explicitly programmed by naming the types they operate on. Systems such as ELAN and Stratego fix this deficiency.

2.5.2 ELAN

ELAN [16] is a rule-based programming language designed for developing theorem provers, logic-based programming languages, constraint solvers and decision procedures. ELAN was not specifically designed for writing program transformations. However, it is useful in the problem domain. One of the first applications of ELAN was to prototype a functional programming language.

ELAN is notable to us because it introduced the concept of a rewrite strategy. This

language construct was motivated by a need to manage rule applications in large rewrite programs with many stages and subtasks. Strategies allow programmers to specify order and hierarchy dependencies between rules that would otherwise be applied nondeterministically to the entire input tree. They are used to declare explicitly how trees are walked and which rules are applied during a walk.

Strategies have since made their way into rewrite systems that are specifically designed for program transformation. We discuss strategies further in the next section on the Stratego Language.

2.5.3 Stratego Language

Recognizing the need for better control of the application of rewrite rules, the Stratego Language [107] introduced the concept of strategies to rule-based program transformation. Stratego is a direct descendant of ASF+SDF and ELAN, combining the program definition and transformation facilities of ASF+SDF with the separation of traversal from rewriting that was introduced in ELAN.

A strategy [106] in the Stratego Language is a declarative construct that allows careful control over the dependency relationships between rules. A strategy takes transformation tasks as parameters and specifies how they are to be applied. Theses tasks may be rules or other strategies. Strategies look very much like rules. However, the distinguishing characteristic is that they do not explicitly mention the terms that they modify. Instead, they reference other strategies and rules and implicitly operate on terms.

Complex strategies are constructed using operators that define relationships between the transformation tasks. For example, the sequence operator `s1; s2` causes the application of one strategy, then the next. The `s1 + s2` operator attempts to apply one of two strategies. A set of inbuilt strategies define the basic traversal behaviour. The `rec x(s)` strategy defines a symbol, `x`, whose reference in `s` represents a recursive application of the strategy. The `all(s)` strategy applies the strategy `s` to all subterms of the current term being processed.

```
module traversals
import lists
strategies
    try(s)              = s <+ id
    repeat(s)           = rec x(try(s; x))
    topdown(s)          = rec x(s; all(x))
    bottomup(s)         = rec x(all(x); s)
    downup(s)           = rec x(s; all(x); s)
    downup2(s1, s2)     = rec x(s1; all(x); s2)
```

Figure 2.14: Some common rule application strategies. Strategies do not explicitly mention the terms that they operate on. They take other strategies (a rule is a simple form of a strategy) as parameters and implicitly traverse terms. For example, the `all(s)` strategy applies `s` to all subterms of the current term.

These operators and basic strategies allow the programmer to specify a traversal without giving the specific forms involved.

Figure 2.14 shows some common strategies that may be programmed in Stratego. The `try(s)` strategy first applies strategy `s`; if that fails it applies the identity strategy. The `repeat(s)` strategy applies `s` until it fails. The `topdown(s)` strategy applies `s` to the current term, then recurses. The `bottomup(s)` strategy recurses first, then applies `s`. The `downup` strategies combine top-down and bottom-up applications.

Stratego has many other interesting features; for example dynamic rules allow a program to define rules at run-time. Overlays permit patterns to be abstracted so language syntax need not be directly specified in a rule. Research into how to better express transformations through control of the rewrite process is ongoing in the Stratego community.

2.5.4 TXL

TXL [25] is a hybrid functional and rule-based program transformation system. It uses a generalized recursive-descent parsing engine with extensions for accommodating left-recursive grammars. A TXL grammar is normally divided into a base grammar and grammar overrides. The base grammar defines the lexical and syntactic forms of the language to

be processed. The lexical forms are given in a regular expression notation. The syntactic forms are defined using an extended BNF-like notation. Grammar overrides can be used to extend or modify the base grammar, allowing new language forms to be defined in a modular fashion. In this way, TXL promotes reuse of grammars.

Program transformation behaviour is defined by a set of transformation rules, which are able to search parse trees for by-example style patterns and provide replacements. Rule application may be controlled by further pattern matching on pattern components, as well as by conditions programmed in a functional style.

TXL provides an inbuilt tree traversal mechanism. If the default tree searching is inadequate, arbitrary tree traversals may be programmed by explicitly coding the traversal in a functional style. Similarly, the order of rule application is controlled by the explicit invocation of rules using TXL's function application mechanism.

To fully take advantage of TXL's abilities, understanding the hybrid nature of rules is essential. A rule exhibits characteristics of term rewriting rules in the sense that a rule repeatedly searches its scope for instances of its pattern and makes a replacement. On the other hand, a rule is a functional style program that looks and behaves like a statically typed lisp function with implicit iteration. On some occasions it is advantageous to program using the rewriting paradigm. At other times it is necessary to be programming in a functional style with the iteration disabled.

A good meta-language system must make considerations towards program readability by clearly separating the syntax of the meta-language from the syntax of the input language. The boundary between the two must be unambiguous and visually distinct. This is something that TXL does very well.

2.5.5 Integration with General-Purpose Languages

Transformation systems in the style of rule-based systems provide very nice language constructs for manipulating trees. The systems are heavily tuned for this use. During the

```
// Find definition. Parameterized by comparison
// strategy. Should be applied to a (key, tree) pair.
find(cmp) =
    // First deconstruct into key and tree, then
    // replace with tree.
    ?(k, t) ; !t ; (
        // Check if tree is empty, in which case
        // we return the empty.
        ( ?Empty ) <+ (
            // Tree is not empty. Must be a node. Take
            // the node apart.
            ?Node( n, v, t1, t2 ); (
                // Check for "less than". Recurse on success.
                ( test(<cmp>(k, n)) ; <find(cmp)>(k,t1) ) <+

                // Check for "greater than". Recurse on success.
                ( test(<cmp>(n, k)) ; <find(cmp)>(k,t2) ) <+

                // Else, must have a match. Return the value.
                !v
            )
        )
    )
```

Figure 2.15: Binary search in Stratego.

course of constructing large transformation applications, the need for arbitrary computer algorithms can creep into many applications. If the transformation language is not amenable to general-purpose programming then the benefits gained by exploiting transformation features become lost to the difficulties of implementing custom algorithms that go alongside the transformations. This is a problem that rule-based systems suffer from.

Consider the need for binary search, a task so common it is one of the first concepts taught in computer science curricula. The code in Figure 2.15 implements binary search in the Stratego Language. Asking a programmer to recognize binary search in this comment-aided form is reasonable. However, expecting a time-strapped, results-driven programmer to implement binary search in Stratego as a first exercise in learning the language may not be so successful. It requires an understanding of the semantics of strategies, the following

operators `; ! ? <+`, and the program fragments `<cmp>` and `<find(cmp)>`.

On the theme of combining transformation languages with general-purpose languages, there has been work on adapting high-level transformation infrastructure to language-specific compiler front-ends [54]. This work makes it possible to write domain-specific optimizations as transformations on the host-language AST. The JastAdd [30] compiler system is an example of doing this. It adds transformation and attribute grammar features to Java, allowing users to program extensions to the language. The CodeBoost [9] system brings Stratego rewrites to the C++ AST, with high-performance computing applications in mind.

Tom [72] is a term-based pattern matching library that has been used to extend Java and other languages [10]. Tom brings term definition, pattern matching and term construction to general-purpose languages. Huang presents a programming technique that uses first-class patterns to support the analysis of tree-structured data in general-purpose programming languages. [42].

We find that there has been work on integrating rewriting systems into specific general-purpose languages for the purpose of extending that language, or for term rewriting in general. However, there is no full transformation system, complete with parsing, transformation and unparsing facilities, that uses language constructs common in general-purpose programming languages.

2.6 Summary

Transformation systems are built up from a number of essential features. In Section 2.2 we look at these features to get an idea of what goes into the construction of a transformation system. We then take a closer look at parsing, since there are problems in this area that we wish to address. In Section 2.3 we survey the research into generalized parsing and analyze the methods with respect to criteria that are of interest to us. In Section 2.4 we look at

the issues and research surrounding context-dependent parsing. It is in this section that we expose the difficulty caused by languages with ambiguities and context dependencies. From these problems we are motivated to design a new transformation system that does not stumble on such practical issues. In Section 2.5 we survey the existing source transformation systems, giving brief overviews of the transformation languages they employ. In this section we also motivate our desire to design a transformation language that resembles general-purpose languages. This motivation comes from the difficulty of implementing support algorithms in transformation languages, as well as recent research into extending general-purpose languages with transformation language features.

In the next chapter we take the first step in designing a new transformation system by choosing a generalized parsing method from the methods described in Section 2.3. In choosing a method we must face trade-offs. Most of the issues we must live with, however we find that we are able to address the issue of controlling the parse in our method of choice, and we can do so without compromising any benefit that the algorithm brings.

Chapter 3

Generalized Parsing for Tree Transformation

3.1 Introduction

Generic source transformation systems require a generalized parser. Having explored generalized parsing algorithms in the previous chapter, we now turn to the problem of choosing a generalized parsing algorithm on which to base this research. After discussing the tradeoffs we settle on backtracking LR. This algorithm is not perfect, however, and there are downsides to choosing it. One problem we can do something about; existing backtracking LR systems do not support grammar-based control of the parse. We find that if we carefully order conflicting shift and reduce actions we can approximate ordered choice. This allows us to control the parse as we can in generalized recursive-descent systems such as TXL [25], but we still retain all the benefits of a bottom-up parsing system. After describing our ordering algorithm and its limitations, we discuss how it impacts left-recursive and right-recursive lists. We then give a solution to the problem of pruning the search space, and show how error handling can be implemented in our system. In the final section, we describe how we have integrated grammar-dependent scanning into our system, bringing it under the control

of our ordered choice algorithm.

3.2 Backtracking LR

Choosing a generalized parsing algorithm for source transformation requires considering many factors. For example, in what ways does the method limit the grammars that can be specified? How can context-dependencies be handled? What error recovery mechanisms are supported? What kind of run-time performance can be expected? How much memory will be required at run-time? How much grammar ambiguity can the method handle while remaining practical? How can ambiguities be resolved?

Much of the current parsing research is focused on GLR [94] techniques. Many parsing papers suggest ways to apply the GLR algorithm to real programming languages and there are several examples of real systems that employ GLR. However, we cannot ignore the fact that GLR parsing is incompatible with the feedback technique. Parsing modern programming languages requires the maintenance of complex context-dependency information during parsing. Pursuing multiple parses concurrently, as is done in GLR, requires that we also maintain multiple copies of the global state. This unfortunately prevents us from merging GLR parse stacks and therefore inhibits the GLR algorithm.

Other non-backtracking algorithms, such as Earley [29], CYK [22, 112, 55] and Packrat [35] also suffer from the problem of managing concurrent writes to global information. Maintaining per-parse global information is at odds with the sharing of parse tree results, degrading the parsing algorithms to naive nondeterministic parsing with exponential running time.

The backtracking algorithms are easier to apply to context-dependent parsing problems because at any given time there is only a single parse being pursued. They allow us to maintain a global state and use it for lexical feedback or semantic conditions, provided that during backtracking we are able to revert modifications to the global state.

Conceding the importance of context-dependent parsing we base this work on backtracking. Backtracking has other advantages too: it is easy to understand, it can be controlled for the purpose of resolving ambiguities, and it subsumes an error recovery technique. The caveat is to focus one's effort on limiting the amount of backtracking to make parsing fast in practical cases.

The next choice we must make is between top-down and bottom-up parsing algorithms. The main advantage of choosing a top-down parsing algorithm is that top-down algorithms give us the very convenient ordered choice parsing strategy. This allows us to specify which ambiguous grammar forms are to be preferred over others simply by ordering grammar productions.

On the other hand, if we choose a bottom-up approach then the system will inherit the speed of deterministic bottom-up parsers for deterministic portions of the grammar. This will result in much faster parsing than generalized top-down can give us. Another reason to choose bottom-up is that we can utilize the well-known bottom-up semantic action model of Yacc [46].

Rather than choose between ordered choice and speed we aim for both by looking for a way to control the parsing of backtracking LR. In the next section we focus on backtracking LR and present an enhancement to it that gives us an ability to control the parse of ambiguous constructs.

3.3 User-Controlled Parsing Strategy

The first contribution of this thesis addresses the desire to use ordered choice to control the parsing of nondeterministic language definitions in the context of backtracking LR parsing. Ordered choice gives users localized, grammar-based control of the order in which the parser attempts to match grammar productions to the input. In the case of ambiguous grammar definitions, this allows the user to resolve the ambiguities. In the case of nondeterminism

```
orderState( tabState, prodState, time ):
    if not tabState.dotSet.find( prodState.dotID )
        tabState.dotSet.insert( prodState.dotID )
        tabTrans = tabState.findMatchingTransition( prodState.getTransition() )

        if tabTrans is NonTerminal:
            for production in tabTrans.nonTerm.prodList:
                orderState( tabState, production.startState, time )

                expandToState = traverseProdInTable( tabState, production )
                for all followTrans in expandToState.transList
                    reduceAction = findAction( production.reduction )
                    if reduceAction.time is unset:
                        reduceAction.time = time++
                    end
                end
            end
        end

        shiftAction = tabTrans.findAction( shift )
        if shiftAction.time is unset:
            shiftAction.time = time++
        end

        orderState( tabTrans.toState, prodTrans.toState, time )
    end
end

orderState( parseTable.startState, startProduction.startState, 1 )
```

Figure 3.1: Algorithm for ordering shifts and reduces to emulate a generalized top-down strategy in backtracking LR

---

without ambiguity, this lets the user optimize the parser by reducing backtracking.

Implementing a backtracking LR parser requires that some changes be made to the standard deterministic LR run-time driver. When the parser encounters a parse table conflict it must choose an action to take, then store the alternatives for later consideration. We are interested in statically ordering these alternatives in such a way that an ordered choice parsing strategy is approximated. To accomplish this we traverse the parse tables and assign an order to the shift/reduce actions using the algorithm shown in Figure 3.1.

The order of the traversal is guided by a top-down interpretation of the grammar. We start with a nonterminal as our goal and consider each production of the nonterminal in succession. As we move along a production's right-hand side we recurse on nonterminals before we proceed past them. When we visit a shift transition we assign a time to it if one has not already been given. Following the traversal down the right-hand side of a production, we find the transitions that contain the reduction actions of the production instance and assign a time to each reduction action if one has not already been given.

To limit the traversal and guarantee termination we visit a parse table and production state pair only once. This is accomplished by inserting the dot item of the production state into a set in the parse table state and proceeding with the pair only when the dot item did not previously exist in the set. Note that this yields dot sets identical to those computed during the LR parse table construction.

3.3.1 Ordering Example

In this section we give an example of our ordering algorithm as applied to the parse tables of Grammar 2.3. These tables were first shown in Figure 2.9 without any ordering information. A trace of the backtracking LR algorithm as it parses the input string `a a b a` was shown in Figure 2.10.

```
S → AB b              AB → AB a
  | a A a AB              | AB b        (Grammar 2.3 repeated)
A → A a                   | ϵ
  | ϵ
```

In Figure 3.2, we give the parse tables of Grammar 2.3 with the ordering information displayed. The timestamps assigned by our action ordering algorithm are shown in each transition action. There are two conflict points. The first is in the transition leaving the start state on the input character `a`. This transition will first induce a reduction of `AB`, then a shift. This represents the pursuit of the first production of `S`, then the second. The second conflict represents the choice between extending the `A` sequence and beginning the

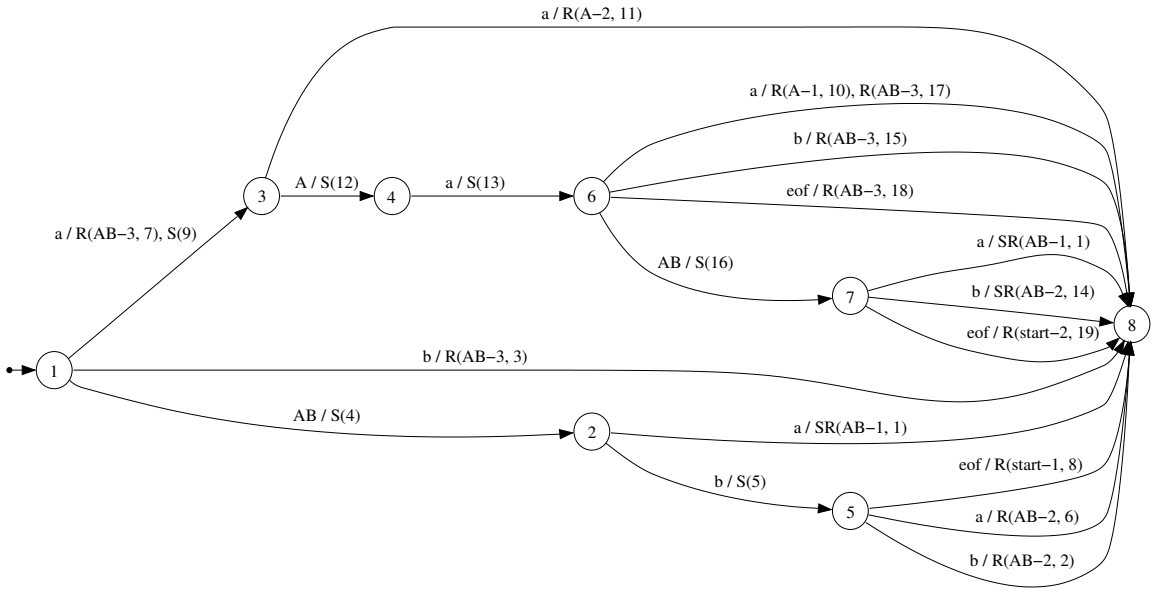

Figure 3.2: LALR(1) parse tables of Grammar 2.3. The unique timestamp assigned to each action is shown. The resulting action ordering emulates a top-down parsing strategy.

`AB` sequence when we are matching the second production of `S`. In this case the parser first shifts and reduces `A` to pursue a longest match of `A`.

3.3.2 Out-of-Order Parse Correction

Unfortunately, it is possible to find grammars whose mutually ambiguous productions will not be parsed in order. As it turns out, the parse strategy we aim for is reserved only for true top-down parsers. LR parsers attempt to parse common production prefixes in parallel. This allows parsers to run very fast, but it can inhibit us from achieving a top-down strategy because it shuffles the order of backtracking decisions by delaying the branching of productions. For example, consider the following grammar.

```
S → F b x          F → a
  | F                | a b c      (Grammar 3.1)
  | F b c
```

When given the input string `a b c`, a generalized top-down parser will attempt the following derivations. Note that it branches on the possible derivations of `S` first, then branches on the possible derivations of `F`.

```
S → F[ a ] b x        fail
S → F[ a b c ] b x    fail
S → F[ a ]            fail
S → F[ a b c ]        accept
```

Our backtracking LR algorithm with the above ordering applied does not yield the same parse. Since all three `S` productions have a common prefix `F`, the prefix will be parsed once for all productions. The parser will branch on the possible derivations of `F` first, then later branch on the derivations of `S`. This out-of-order branching causes an out-of-order parse. When we trace the parser's behaviour, we find that it first reduces `F → a`, then succeeds in matching `S → F b c`. The offending LR state tables are shown in the first part of Figure 3.3.

```
S → F[ a ] b x        fail
S → F[ a ]            fail
S → F[ a ] b c        accept
```

Fortunately we are able to solve this problem easily. When we find our input to be parsed out of order with respect to our grammar, we can force a correct order by introducing unique empty productions at the beginning of the productions that are parsed out of order. The unique empty productions will cause an immediate reduce conflict before any inner productions are reduced, effectively allowing us to force the slower top-down parsing approach in a localized manner. We can change Grammar 3.1 to the following and achieve the same parsing strategy as a top-down parser.

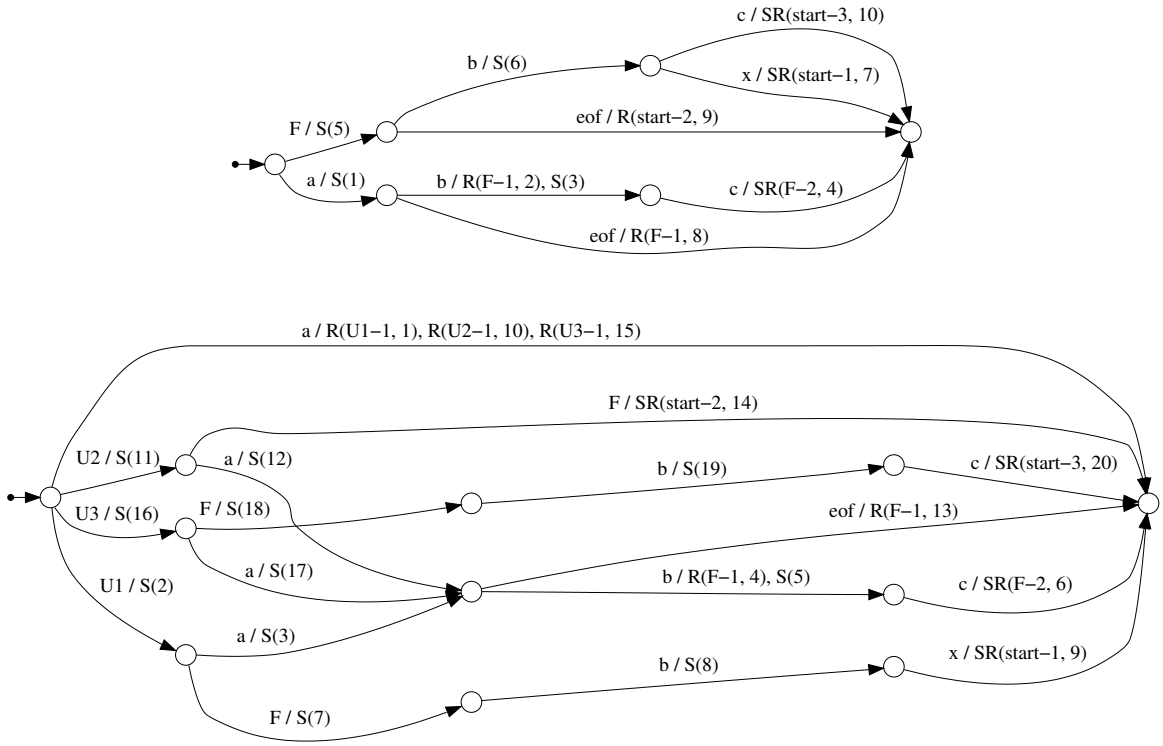

Figure 3.3: LALR(1) parse tables before and after adding unique empty productions that force the parser to select on the possible derivations of S before selecting on the possible derivations of F.

```
S → U1 F b x        F → a          U1 → ϵ
  | U2 F               | a b c     U2 → ϵ    (Grammar 3.2)
  | U3 F b c                      U3 → ϵ
```

The second part of Figure 3.3 shows the LR tables after forcing an initial selection on `S`. An ability to force a branch point is very useful when unioning grammars because it frees us from analyzing how the LR state tables interact. The cost of forcing a branch point lies in increasing the number of states and lengthening parse times. However, we do so only locally, and only when required.

3.3.3 Ordered Choice and Left Recursion

Using our action ordering algorithm we attempt to emulate a top-down parsing approach. This suggests that we may have problems with controlling nondeterminism in left-recursive grammars. This intuition is indeed correct; our emulation of ordered choice cannot be applied to left-recursive grammars with predictable results. Consider the following left-recursive grammar fragment.

```
E → E + F
  | E - F
  | F
```

The ordering algorithm processes the first production and recurses inside each nonterminal that it contains. This results in the recursive exploration of the first production. The guard protecting against the recursive processing of productions prevents the endless recursion that would result from this. The algorithm then moves on to the second production and succeeds at ordering it. The contents of the first production is not ordered until the recursive exploration of `E` returns. The result is that the second production gets a higher priority than the first, and ordered choice is not achieved.

Evidence of this problem can be seen in a trace of our algorithm as it orders the state tables of Grammar 2.3. The trace is shown in Figure 3.4. The ordering of `AB-2` occurs ahead of `AB-1`.

It would be convenient if we could apply the above strategy of inserting unique empty productions to order these productions. However, a unique empty production at the head of a left-recursive production creates hidden left recursion, a grammar structure that backtracking LR cannot process without going into infinite recursion. We must settle on the fact that ordered choice cannot be achieved for left-recursive grammars. Therefore it is recommended that right recursion be used when recursive productions need to be ordered.

Traversal	State	Stopped by Guard
S-1 → . AB b	1	
AB-1 → . AB a	1	
AB-1 → . AB a	1	X
AB-2 → . AB b	1	
AB-1 → . AB a	1	X
AB-2 → . AB b	1	X
AB-3 → .	1	
AB-2 → AB . b	2	
AB-2 → AB b .	5	
AB-3 → .	1	X
AB-1 → AB . a	2	
AB-1 → AB a .	6	
AB-2 → . AB b	1	X
AB-3 → .	1	X
S-1 → AB . b	2	
S-1 → AB b .	5	
S-2 → . a A a AB	1	
S-2 → a . A a AB	3	
A-1 → . A a	3	
A-1 → . A a	3	X
A-2 → .	3	
A-1 → A . a	4	
A-1 → A a .	7	
A-2 → .	3	X
S-2 → a A . a AB	4	
S-2 → a A a . AB	7	
AB-1 → . AB a	7	
AB-1 → . AB a	7	X
AB-2 → . AB b	7	
AB-1 → . AB a	7	X
AB-2 → . AB b	7	X
AB-3 → .	7	
AB-2 → AB . b	8	
AB-2 → AB b .	9	
AB-3 → .	7	X
AB-1 → AB . a	8	
AB-1 → AB a .	6	X
AB-2 → . AB b	7	X
AB-3 → .	7	X
S-2 → a A a AB .	8	

Figure 3.4: Trace of our action ordering algorithm as it computes the orderings in Figure 3.2 from Grammar 2.3. Note that state 9 exists here but does not exist in the state tables because it has been eliminated by the creation of shift-reduce actions.

3.3.4 Sequences

In the previous section we show that our approximation of ordered choice does not work for ambiguous left-recursive grammars, and we recommend using right recursion when ambiguities need to be resolved. Our parsing system is bottom-up and recommending right recursion for use with a bottom-up parsing algorithm is indeed a bit odd. Bottom up parsers normally use left-recursion to implement sequences of items. In a left-recursive grammar reductions happen after each item in the list. The constant alternation between shifting and reducing saves memory since reduced tree nodes can be discarded after reduction.

Right recursion, on the other hand, causes a long sequence of shifts for each item, then a long sequence of reductions. If we use right recursion, all items in the list must remain in memory until the end of the list is reached. However, since we must keep all trees around for the transformation phase, we feel no pressure to use left-recursive lists to save memory.

Aside from controlling the parse there is another reason to recommend right recursion for our transformation system and it is unrelated to parsing. A top-down tree traversal of a right-recursive sequence yields a left-to-right walk of the sequence. This makes it easy to process lists in a functional style. For example, a function can easily deconstruct a list into a head item and the tail of the list, process the head, then recurse on the tail and the result is a left-to-right traversal. If we were to use left-recursion to implement lists this would not be the case. A less natural bottom-up traversal is required to traverse the left-recursive list items in order.

Since we cannot save memory by using left recursion for sequences we have made the in-built sequence operator (Kleene star) syntactic sugar for a right-recursive list. Such lists are more convenient to processes following the parse and the right-recursion is compatible with our action ordering algorithm. The expression `item*` expands to the following grammar fragment.

```
list → item list
     | ϵ
```

3.3.5 Longest/Shortest Match

In addition to controlling ambiguities that involve the relative ordering of productions, we also need an ability to control ambiguities that involve sequence lengths.

Since right recursion affords us an ability to control parsing using ordered choice, it becomes easy to decide between longest and shortest match by choosing the ordering of the productions of a right-recursive list. This is a property of generalized top-down parsers that we have inherited by approximating ordered choice. If we would like to attempt to find the longest possible sequence of items we specify the recursive production first and the empty production second. In this form we declare that given the opportunity to match an additional item or to terminate the list, match an additional item.

```
list → item list
     | ϵ
```

If on the other hand we would prefer to find shorter lists and extend them only when necessary we can specify the empty production first and recursive production second. In this form we declare that given the opportunity to match another item or to terminate the list, choose to terminate. If that strategy turns out to fail then the backtracker revisits the decision and attempts to match another item.

```
list → ϵ
     | item list
```

It should be remembered that, on list termination, there is a cost proportional to the length of the list. In the previous section we noted that right recursion involves a long sequence of shifts followed by a long sequence of reductions when the list is terminated. If the parser is repeatedly closing off a list only to fail and backtrack in order to try a list of a different length then this cost will be incurred each time. Left-recursive lists do not suffer from this problem since each item is reduced after it is shifted.

In the light of this cost it is worthwhile to explore opportunities for specifying shortest or longest match of left-recursive lists. We don't have ordered choice at our disposal but

other techniques are possible. For example, it is possible to get a longest match by wrapping a left-recursive production in a fresh nonterminal. This means that there is one exit point from the list and it is always attempted after any shifts or reduces inside. The result is a forced longest match. This is guaranteed to work for direct left-recursion only.

```
list_wrapper → list

list         → list item
             | ϵ
```

If we want a shortest match with a left-recursive grammar we can provide a hint to the compiler that alters the ordering algorithm on a per-nonterminal basis. Normally, our ordering algorithm causes the shift/reduce actions inside a nonterminal to take precedence over the action that reduces the nonterminal. This gives us longest match by default. If we reverse this order for a particular nonterminal, causing the reduction to happen before any shifts/reduces inside we get a shortest match for that nonterminal. Again, this is guaranteed to work for direct left-recursion only.

3.4 Discarding Alternatives

The performance of a backtracking algorithm is often improved by pruning alternatives from the search space. In our backtracking LR algorithm, discarding alternatives drastically improves parser performance by eliminating fruitless reparsing, reducing the parser's memory usage and enabling it to report erroneous input in a prompt and accurate manner.

We take advantage of the fact that programming language ambiguities are often localized [111]. For example, once the parsing of a C++ statement completes, any alternative parses of the statement's tokens can be discarded.

We allow the user to associate with a nonterminal a commit flag that indicates that following the reduction of the nonterminal, all alternative parses should be discarded. This is a global commit point that causes the entire parse stack to be pruned of alternatives.

This operation guarantees the parser will never backtrack over the previously parsed input in search of another possible parse. Using commit points, the user can declare parsing milestones at important constructs such as statements or declarations. This type of commit point can be found in a number of backtracking LR systems.

There are cases where global commits are too strong. It may be desirable to eliminate alternative parses of a particular nonterminal, but allow alternatives ahead of the nonterminal to exist and be considered if the parser encounters an error. For example, if parsing a sublanguage inside a host language at the statement boundary, it may be desirable to commit a fragment of the sublanguage, but still consider that parsing of the sublanguage may fail, requiring that the parser back up and attempt to parse another statement of the host language.

```
host_stmt → U1 sub_language
          | U2 real_host_stmt
```

To accommodate this we add a second type of commit flag. A reduction-based commit point is associated with a grammar production. When the production is reduced, all alternative parses of the text within the nonterminal are deleted. Alternatives ahead of the nonterminal are preserved.

Note that it can be difficult to cover the entire grammar using these localized commit points. It is therefore recommended that they be used in combination with global commit points so that at periodic intervals it can be guaranteed that all unnecessary alternatives are removed.

3.5 Error Recovery

Ordered choice gives us a convenient error handling mechanism. Unique empty productions can be used to require that error handling productions that consume input be attempted only after all normal productions fail. The following error handler consumes input until the

input stream and parser go back into a stable state where correct parsing may resume.

```
statement → U1 for_block
          | U1 while_block
            ⋮
          | U2 not_semi ';'
```

3.6 Grammar-Dependent Lexical Rules

To complete our parsing system we must devise a scanning approach. Treating the input as a sequence of tokens that come from a single pool of token patterns is too restrictive. The user must be able to specify multiple lexical regions in which different lexical rules apply. Allowing the user to manually choose these regions would be too informal and would present a challenge when composing grammars. The choice of active lexical region should instead depend on the grammar.

The user assigns each token to a single lexical region. There can be many tokens in a region, or just one. Each region is then made into a scanner, in the usual sense, using the token definitions inside the region. When the parser needs to retrieve a token it consults the parser state tables to determine which tokens can be accepted. From this list it determines the list of regions that the accepted tokens can come from. The identity of the first region in this list is used for requesting a token. The identity of the next region is stored for possible use if the parser backtracks.

The list of regions is determined from the grammar and the user is able to control scanning using the grammar, yet at the same time scanning can be controlled from the lexical rules when needed. The parser requests a token from a particular region, but the scanner still has the final say as to which token is sent. The usual prioritization of token patterns can take effect, or hand-written code can be used to generate tokens programmatically.

This minor decoupling makes it possible to drive scanning from the grammar, yet still support whitespace and comment tokens. We refer to these tokens as ignore tokens because

```
statement → . SQL_statement
          | . C_statement
```

Accepted Tokens	From Region	Attempted Regions
SQL_word	SQL	SQL
SQL_number	SQL	
SQL_string	SQL	
⋮		
C_word	C	C
C_number	C	
C_string	C	
⋮		

Figure 3.5: The ordering of tokens at the first state of a union grammar containing C and SQL statements. The token ordering results in the parser first attempting the SQL region, then the C region.

---

they are never mentioned in the grammar. Like real tokens, they must be assigned to a region. It becomes possible to match an ignore token when the parser can accept a real token from the same region. Ignore tokens are always associated with the next real token, so the scanner can simply accumulate ignore tokens until it matches a real token. The real token is sent to the parser with the preceding ignore tokens attached.

As alluded to above, when a token is sent back by the backtracker the next possible region is attempted. Since we enable the scanner to backtrack through the regions of all possible tokens accepted in the current parser state, we must order the list of regions. For this we rely on the ordering algorithm described in Section 3.3. Following the execution of the ordering algorithm, we take the ordered list of tokens that can be accepted in a state, map these to regions, then remove duplicates from the ordered list.

Consider the union grammar in Figure 3.5. It accepts either SQL or C statements at the statement level. A parser state at the head of a statement would accept a list of SQL tokens followed by a list of C tokens, resulting in the parser first attempting the lexical region of the SQL tokens, followed by the lexical region of the C tokens.

3.7 Summary

In this chapter we choose a generalized parsing method and provide an enhancement that addresses one shortcoming of the method. Our method of choice is backtracking LR and in Section 3.2 we give our reasons for this choice. We recognize that existing implementations of this technique do not allow the user to control the parse of ambiguous grammars by ordering productions. This is a facility provided by generalized recursive-descent systems such as TXL, and is one that we deem important in transformation systems. In Section 3.3 we solve this problem by providing an algorithm that orders conflicting shift and reduce actions to approximate ordered choice. True ordered choice is reserved for generalized recursive-descent systems and therefore our technique is limited in the sense that it is not completely automatic. In some cases the user must intervene and assist the algorithm. However, the required intervention is usually minimal, and if the user is unsure if intervention is required the only cost of a conservative approach is a minimal degradation in performance.

All backtracking systems have the potential to blow up to exponential running time. Other generalized context-free parsing algorithms such as GLR and Earley do not face this problem, and this is by the strength of their design. Exponential running times can be avoided in practice, but regardless it is important to provide a mechanism for reducing the search space. Pruning is useful for optimizing a backtracking system. We address this issue in Section 3.4 with global and local commit points. Also related to backtracking is the issue of error recovery. In Section 3.5 we point out how our backtracking system subsumes an error recovery technique. In Section 3.6 we introduce grammar-dependent scanning into our system. We recognize that an ability to backtrack over lexical choices increases the generality of our system. Therefore we put the grammar-dependent lexical analysis under the control of our action ordering algorithm.

This concludes our discussion of generalized context-free parsing. We will come back to the topic of parsing and make further enhancements to the parsing engine. However,

we must first present the design of our transformation language because we will need it to make these further enhancements.

Chapter 4

COLM: A Tree-Based Programming Language

We now divert from the topic of parsing and move to the topic of tree transformation languages. We give an informal definition of COLM, the tree transformation language that we have designed. While it true that we wish to advance the state of tree transformation languages, this experiment in transformation language design is only secondary in this work. We wish to exploit this language and its implementation to advance the parsing capabilities of our system. With this language defined we will move back to the topic of parsing in the next chapter to show how we have enabled context-dependent generalized parsing.

We begin this chapter by talking more about our reasons for developing a new language and implementation. Such a task is a large undertaking and we wish to be clear about our motivations. We then work through the language design. We first describe the features borrowed from general-purpose languages, then cover the transformation features. Topics in these sections include input language definition, matching patterns, constructing trees, traversing trees, and generic tree transformation idioms. Our language definition technique is informal. We give the grammar fragment pertaining to the features we want to describe,

a brief description of the semantics and then a small example to illustrate the idea.

We then give the reader an idea of what a real transformation looks like in our language by giving a goto-elimination example. Finally, we discuss some of our implementation choices. We have several implementation options and some of our choices will have an impact in the following chapter when we continue to improve upon our parsing engine.

4.1 Designing a New Transformation System

Before we present the design of our transformation language we must ask the question, why design a new transformation system at all? Why not adapt an existing transformation system to demonstrate our technique? There are a number of reasons why designing a new language and implementing it from scratch is the better approach for this research. Some reasons are technical, others have to do with our need to extend the transformation language with features that allow us to demonstrate context-dependent parsing. We also believe that designing a new transformation language can lead to improved adoption of transformation systems if we focus on designing one that is closer to general-purpose languages.

4.1.1 Implementing a New System

In this work we have chosen to implement a new transformation system from scratch instead of extending an existing system. Extending an existing system presents several challenges for us.

Since we have a new parsing method we must implement a new parsing engine. Our parsing engine must generate trees that the transformation system can manipulate. If we were to base this work on an existing system such as TXL [25], we would first have to remove the parser from the system, then ensure that the trees generated by our parsing engine can be used with the tree manipulation system that we have separated out. Transformation systems are often tightly integrated with the parsing engine. This technical challenge could

easily inhibit the progress of this work.

The second technical challenge in adapting an existing system lies in adapting the code generated by the transformation language's compiler. As described in the following chapter, we need to extend the implementation of the transformation language with automatic reverse execution. This introduces some unusual requirements into the language's compiler and virtual machine implementation. If we were to utilize an existing system we would need to understand and adapt it to meet our requirements for reverse execution, possibly working around conflicting requirements. Existing implementations are usually optimized and thus not designed to be easily understood and extended in such a way.

Rather than attempt to adapt an existing implementation, which might have resulted in time-consuming implementation challenges that were unforeseeable from the start, we choose to start from scratch and build an entirely new system from the ground up, while keeping our language implementation requirements in mind. This may be the longer path in the end, given that all of the transformation language's features must be written from scratch, but it is a straightforward approach and it is more likely to result in success.

4.1.2 Designing a New Language

Implementing a new transformation system from scratch gives us a good reason to also design a new transformation language. Starting with a new language frees us from having to meet existing implementation requirements and allows us to evolve the language and implementation together, free of external constraints.

To support our parsing technique our transformation language must support the building of common data structures. The goal is to enable parse-time modification of global state, which will support the recognition of context-dependent language constructs. To show our parsing method is capable we need to add to the language any features that are needed to represent the global state of real target languages. This includes explicitly allocated objects, pointers, lists and maps. Designing a new language allows us to include the needed

features without being concerned about the usual language extension problems, such as limited room for syntax expansion and the inadvertent creation of ambiguities.

Designing a new language from scratch also allows us to experiment with transformation language design. As we have seen in Section 2.5.5, researchers are interested in merging transformation language features with general-purpose languages. However, there is no complete transformation system with a language that resembles any general-purpose language. It is our hope that by designing a language with imperative features that are oriented towards the manipulation of state, we can widen the adoption of transformation systems by appealing to the majority of programmers who are already familiar with this programming paradigm.

4.2 Language Overview

Our focus is on designing a language that enables language definition, supports the parsing method described in Chapter 3, supports common transformation techniques, and allows programming in an imperative style. For the purpose of this research we need a small language with a simple semantics. In the following chapter we will use the language to explore context-dependent parsing and we will need to modify the implementation with automatic reverse execution. Due to this extra complexity, a simple language with a simple implementation will be a great asset to us.

Aside from meeting the goals stated above, we leave more focused language design for future work. Examples of issues that we have not concerned ourselves with include standard library design and program modularity. We have skipped many implementation issues as well. We have only a rudimentary reference-counting garbage collector that doesn't break cycles and we have done no exploration of relevant optimization techniques. We expect further evolution of the language and development of the implementation to take place.

Also in future work, it would be worthwhile to explore the formal semantics of this

```
start              → root_item*

root_item          → statement                    (§4.3.1)
                   | function_def                 (§4.3.3)
                   | global_def                   (§4.3.4)
                   | token_def                    (§4.4.1)
                   | literal_def                  (§4.4.1)
                   | region_def                   (§4.4.1)
                   | cfl_def                      (§4.4.2)
                   | generic_def                  (§4.4.3)
                   | iter_def                     (§4.7.3)
```

Figure 4.1: List of root language elements.

language. A formal semantics for the TXL language has been provided by Malton [64]. A formal semantics for our language would be useful not only for precisely communicating the meaning of the language, but also for exposing inconsistencies and other issues with the language.

4.2.1 Value Semantics

When working in a transformation language, many tree variables are used. Tree values are frequently passed around and used to build other trees. In such an environment, a value semantics is essential. With many variables present it becomes difficult for a programmer to maintain a mental model of the dependencies between variables. Using a value semantics eliminates the need to track dependencies. For this reason, each variable in our language represents a distinct value that, when manipulated, does not affect the value in any other variable. Variables may be set to some tree value, or may be set to `nil`.

4.2.2 Root Language Elements

The grammar listing in Figure 4.1 gives the root grammar items of our language. At this level of the language it is possible to define types, functions, global variables and to give code. Code at the root level is collected and treated as the program's entry procedure, as

```
identifier          /[a-zA-Z_][a-zA-Z0-9_]*/
number              /[0-9]+/
single_literal      /'([^'\\]|\\.)*'/
pattern_char        /[^\["]|\\./
comment             /#[^\n]*\n/
```

Figure 4.2: Syntax of the meta-language tokens.

is commonly done in scripting languages. Variables that are declared at the root level are treated as local to the root-level code. Global variables must be explicitly marked as such (Section 4.3.4).

4.2.3 Tokens

In Figure 4.2 we list the tokens used in the grammar that defines our language. We make use of identifiers, numbers, literal strings delimited with single quotes (`'`), and comments that begin with the hash symbol (`#`). Double quotes (`"`) are used to delimit literal text in patterns and constructors. These are defined in Sections 4.5 and 4.6.

4.3 General-Purpose Constructs

In this section we describe the features of the language that are borrowed from general-purpose languages. The concepts in this section should be familiar to most programmers since they are often found in imperative languages. It can be assumed that the general-purpose constructs take on their usual popular meaning.

4.3.1 Statements

The statements of the programming language can be divided into two groups: statements commonly found in imperative programming languages (such as expressions, conditions and loops), and parsing and transformation-related statements. The general-purpose statements

```
stmt_list          → statement*

statement          → var_def ( '=' expr )?
                   | expr
                   | 'if' expr block_or_single elsif_list
                   | 'while' expr block_or_single
                   | 'break'
                   | 'print' '(' expr_list ')'
                   | 'print_xml' '(' expr_list ')'
                   | return_stmt                          (§4.3.3)
                   | reject_stmt                          (§4.4.2)
                   | for_iter_stmt                        (§4.7.2)
                   | if_iter_stmt                         (§4.7.2)
                   | yield_stmt                           (§4.7.3)

var_def            → type_ref identifier

type_ref           → identifier repeat_op?
                   | pointer_type_ref                     (§4.3.5)
                   | extended_type_ref                    (§4.4.3)

repeat_op          → '?'
                   | '*'
                   | '+'

block_or_single    → '{' stmt_list '}'
                   | statement

elsif_list         →  elsif_clause* else_clause?

elsif_clause       → 'elsif' expr block_or_single

else_clause        → 'else' block_or_single
```

Figure 4.3: Syntax of statements.

are defined here in Figure 4.3 and the statements concerning parsing and transformation are defined in the sections that follow.

The if and while statements have a semantics similar to the if and while statements found in most general-purpose languages. The test in an if statement is evaluated and if it evaluates to a boolean true, integer nonzero, or a non-nil tree then the body of the if statement is executed. If statements may have else-if clauses and/or an else clause. The

```
int i = 0
while ( i < 10 ) {
    if ( i * (10-i) > 20 )
        print( i, ' ', j, '\n' )
    i = i + 1
}
```

Figure 4.4: An example of declaration, expression, while, and if statements.

---

while statement continues to execute the body as long as the test evaluates to true, as determined by the same test as in the if statement. An example of these general-purpose constructs is given in Figure 4.4.

4.3.2 Expressions

In this section we give an elided definition of expressions. The syntax is shown in Figure 4.5. The productions in between `expr_list` and `primary` are not shown. The missing productions make up the usual set of boolean, relational and arithmetic operators commonly found in imperative languages. The `primary` definition contains the usual entities found at the root of expressions: literal types, variable references, function calls and a parenthesized expression. It also contains several tree construction forms that are defined later on in this chapter.

In Section 4.4 we give the constructs for defining types. A type may contain named attributes so we must provide a method of accessing attributes. We follow object-oriented languages and allow variable references to be preceded by a sequence of qualifications. In the example in Figure 4.6 we demonstrate the use of expressions and qualified variable references.

4.3.3 Functions

We provide functions as they are commonly known in most imperative languages. Functions may be passed arguments and they may return a value. Functions may also have side

```
expr_list          → expr ( ',' expr )*
⋮
primary            → 'nil'
                   | 'true'
                   | 'false'
                   | number
                   | single_literal
                   | var_ref
                   | '(' expr ')'
                   | 'typeid' type_ref
                   | function_call                    (§4.3.3
                   | dynamic_primary                  (§4.3.5)
                   | match_primary                    (§4.5)
                   | construct_primary                (§4.6)

var_ref            → qual* identifier

qual               → identifier '.'
                   | pointer_qual                     (§4.3.5)
```

Figure 4.5: Elided syntax of expressions.

```
def some_type
    another_type AT
    []

def another_type
    int i
    []

int a = 4
some_type ST = construct some_type []
ST.AT = construct another_type []
ST.AT.i = a * 10 + 2
```

Figure 4.6: An example of qualified names and expressions.

effects. Function calls are part of expressions and they are invoked by naming the function and giving a list of arguments in parentheses.

The function definition is composed of a return type, the function's name, a list of parameters and the body of the function. Parameter passing defaults to a call-by-value

```
function_def      → type_ref identifier '(' param_list? ')' '{' stmt_list '}'

param_list        → param_var_def ( ',' param_var_def )*

param_var_def     → var_def
                  | ref_param_var_def                    (§4.7.1)

return_stmt       → 'return' expr

function_call     → var_ref '(' opt_expr_list ')'
```

Figure 4.7: Syntax of function declarations and function calls.

semantics. However, as is shown in the syntax definition in Figure 4.7, a parameter can be explicitly declared as a reference parameter. These are discussed in Section 4.7.1.

A function can have any type as its return type. If a function has a return statement it must specify a tree of that type. If a function fails to explicitly return a value it returns `nil`. Function calls use the `var_ref` definition from the previous section and therefore may be qualified. Function call qualification is used for inbuilt methods that are contained within the scope of a particular type. These functions take an object of that type as an implicit parameter. The `append` method on the inbuilt generic list (Section 4.4.3) is an example of this. In the future we may wish to pursue the object-oriented style further and allow user-defined methods to be added to types.

In the example in Figure 4.8 we show a simple function that adds two integers and returns the result. Following that, we demonstrate a qualified function call to an inbuilt method contained within the scope of the generic list.

4.3.4 Global Variables

Variables declared in functions, user-defined iterators (Section 4.7.3), reduction actions and token-generation blocks are declared as local variables. Within the code blocks of these constructs there is only a single variable declaration scope (in the future this may change;

```
# User-defined function.
int add( int i, int j )
{
    return i + j
}

int i = add( 1, 2 )

# Qualified call to inbuilt function.
list int_list [int]
int_list IntList = construct int_list []
IntList.append( 42 )
```

Figure 4.8: An example of a user-defined function with an associated call, and a call to an inbuilt method contained within the scope of the generic list.

```
global_def          → 'global' var_def opt_def_init
```

Figure 4.9: Syntax of global variable declarations.

modern languages often confine variables to their enclosing blocks).

Code and variable declarations at the root program level are collected and treated as if they have been defined in the program's entry procedure. Therefore, root-level variables are not global variables by default. A variable must be explicitly made globally accessible by prefixing it with the `global` keyword. Global variables become accessible in all code blocks throughout the program. The syntax of global variable declarations is given in Figure 4.9.

4.3.5 Dynamic Variables

Semantic analysis of computer languages often requires more than just tree data structures. Graph data structures are indeed necessary if we are to enable the processing of languages beyond rudimentary syntactic analysis. For this reason we add dynamic variables to the language. In Figure 4.10 we show the syntactic forms related to dynamic variables. Our dynamic variables can be considered global variables whose names are first class values. A

```
pointer_type_ref   → 'ptr' identifier repeat_op?

dynamic_primary    → 'new' primary
                   | 'deref' expr

pointer_qual       → identifier '->'
```

Figure 4.10: Syntax of dynamic variable declarations and references.

---

dynamic variable is created from a value using the `new` expression and can then be assigned to a pointer variable. Pointer variables may be dereferenced to yield the dynamic variable's value. No arithmetic is allowed on pointers.

An additional qualifying operator, `->`, has been added to allow access to a dynamic variable's attributes directly from a pointer. When a dynamic variable is no longer referenced by any pointer variables it is subject to garbage collection. A pointer type has little meaning outside of a program's run-time context; therefore we do not assign it a textual pattern. It cannot appear in a production of a type definition, but it can appear anywhere else.

4.4 Type Definitions

The type system of our language is based on context-free grammars with named attributes. Tree types are divided into terminal and nonterminal types. Terminals are defined by a regular expression pattern, whereas nonterminals are defined by an ordered list of productions. Both terminal and nonterminal types may have named attributes. Attributes can be accessed in the standard object-oriented fashion using the dot or arrow operators. In the case of nonterminals the attributes are associated with the type, rather than with a specific production of the type, therefore there is only one collection of attributes per type.

4.4.1 Token Definitions

In this section we describe token definitions and the associated lexical region definition. Tokens define the terminal types of the input language. Regions are used to group tokens that may appear together in a single token stream.

There are two types of token definitions, those that result in significant tokens that are sent to the parser, and those that are ignored by the parser when matched. Ignored tokens are accumulated by the scanner and attached to the next significant token that is sent to the parser. Ignored tokens always define the allowable whitespace and comments that can precede the significant tokens in the same region.

Among the significant tokens there are two forms: those with names and the literals. Literals are merely tokens that are defined and referenced in the grammar using the unique literal text that they match. Other than the way they are defined and accessed, they are treated in a manner identical to named token definitions.

A lexical region is used to group tokens. Tokens that appear in the same region are made into a single DFA for scanning, which then becomes the region entity that the parser calls upon when a token is required. If a token is not explicitly defined inside a lexical region, then it is assumed to be in a unique region of its own. The grammar-dependent scanning approach is motivated in Section 2.2.1 and described in Section 3.6.

When a token is requested from the active lexical region, the region's scanner finds the longest prefix of the remaining input that matches a pattern. If more than one pattern matches the longest prefix, then the first of the matching patterns is used. The choice of token to return is a final decision as far as the backtracking parser is concerned. Other possible matches from the same region are not considered if the token does not yield a successful parse. Instead, the text of the token is pushed back to the input stream and another region is called upon to yield a token.

Token definitions may have attributes. These are named values that are associated with

```
token_def          → token_kw name? attr_list
                     '/' re_expr? '/' generate_block?

token_kw           → 'rl'
                   | 'token'
                   | 'ignore'

name               → identifier

attr_list          → var_def*

generate_block     → '{' stmt_list '}'

literal_def        → 'literal' single_literal ( ',' single_literal )*

region_def         → 'lex' identifier '{' root_item_list '}'
```

Figure 4.11: Syntax of token and lexical region definitions.

the token value. They may be set during traversal or when the token is created and sent to the parser by a token-generation block. These blocks of code may be attached to a token definition and when they exist they become responsible for creating tokens and sending them to the parser. They may be used to programmatically generate tokens. Token-generation blocks are described in more detail in Section 5.1.

In addition to token definitions the user may also give regular language definitions. This is done using the `rl` keyword. A regular language definition does not become a member of its containing lexical region. It is a named regular expression that can be used in the definition of other tokens, allowing regular expression patterns to be reused.

The listing in Figure 4.11 shows the section of the grammar concerning token definitions. Note that we have omitted the definition of regular expressions. For our purpose common regular expressions suffice.

The example in Figure 4.12 contains the terminal nodes of an expression grammar. In it we make use of two lexical regions. The primary lexical region contains the expression tokens and a secondary region is used just for handling strings. The regions are automatically

```
lex start
{
    token ident /[a-zA-Z_]+/
    token number /[0-9]+/

    literal '+', '*', '"', '(', ')'

    ignore /[ \t\n]+/
}

lex string
{
    token escape /'\\' any/
    token chr /[^\\"]+/
}
```

```
def str_item
    [escape]
|   [chr]

def string
    ['"' str_item* '"']

def factor
    [number]
|   [ident]
|   [string]
|   ['(' expr ')']
```

Figure 4.12: The terminal nodes of a simple expression grammar. The tokens in the `factor` definition come from two lexical regions. The `start` lexical region contains identifiers, numbers, operators and the string delimiter character. The `string` lexical region contains the tokens that are permissible inside strings.

---

selected according to the parser state.

In the primary lexical region there are two named tokens `ident` and `number`, literal definitions for the operators, and a single ignore token for consuming whitespace. This ignore definition matches whitespace ahead of the `ident`, `number` and operator tokens. Until a double quotation character is seen, in an attempt to parse the only production of the `string` type definition, this region is the only active lexical region.

The second lexical region is used only for the contents of strings. The string grammar captures only escape characters and plain text. However, more elaborate parsing is possible. When the parser shifts a double quotation character in an attempt to parse a string, the `string` region becomes active, along with the main region. The primary region is active for the purpose of matching the closing double quotation delimiter.

4.4.2 Context-Free Language Definitions

In this section we give the syntax of the context-free language definitions. A context-free definition is composed of an optional list of attributes and a sequence of one or more productions.

Our context-free notation is equivalent to BNF with a few common regular-expression enhancements. We have added the repetition operators `*`, `+` and `?` for convenience. The presence of a star operator following a reference to a type causes the automatic definition of a right-recursive list that accepts zero or more items of that type. The reference to the type and the star operator are then replaced with a reference to the list. The plus operator causes the automatic definition and reference of a right-recursive list that accepts one or more items of that type. The question operator produces a new definition that contains the type unioned with an epsilon production. Reasons for using right-recursive lists as opposed to left-recursive lists are given in Section 3.3.4.

Like other source transformation systems, our system supports attributes. In the TXL language the attributes are embedded into productions and are therefore dependent on the form of the value. In the Stratego [107] language, attribute definitions (called annotations) are a list of terms that follow the definition of a production. In our system we instead make attributes a property of the nonterminal type and give them names. Doing so allows access to the attributes without deconstructing the type. The attributes can be accessed using the dot operator, following the convention of object-oriented languages.

In Section 3.4 we describe commit points. They show up, here in the language definition, as keyword attributes on productions. The `commit` keyword signifies that all alternative parses should be deleted following a reduction of the production. The `lcommit` keyword signifies that any alternative parses of the text of the production should be deleted.

```
cfl_def            → 'def' name attr_list cfl_prod_list

cfl_prod_list      → def_prod ( '|' def_prod )*

def_prod           → '[' prod_el* ']' commit? reduction_block?

prod_el            → identifier repeat_op?
                    | single_literal repeat_op?

commit             → 'commit'
                    | 'lcommit'

reduction_block    → '{' stmt_list '}'

reject_stmt        → 'reject'
```

Figure 4.13: Syntax of context-free definitions.

The syntax of context-free language definitions is given in Figure 4.13. Reduction actions and the reject statement are covered in the next chapter, which deals with context-dependent parsing. These topics are in Sections 5.2 and 5.3, respectively.

The example in Figure 4.14 shows a lean expression grammar. Attributes are used to store the value that an expression evaluates to and reduction actions are used to propagate the expression value up the tree during parsing. This technique is commonly used with deterministic parser generators such as Yacc.

4.4.3 Extended Types

If we hope to perform semantic analysis of input, we need types that hold more than just strings. We need types that can store numerical data, and trees that can be used as dictionaries and doubly-linked lists. As shown in Figure 4.15, we have added some extended terminal types: `int`, `bool` and `str`. The `int` and `bool` types hold integer and boolean values instead of strings and have no regular expression pattern. The `str` type holds a string and has no pattern.

```
lex start
{
    token number /[0-9]+/
    literal '+', '*', '"', '(', ')'
    ignore /[ \t\n]+/
}

def factor
    int value
    [number]
    {
        lhs.value = r1.data.atoi()
    }
|   ['(' expr ')']
    {
        lhs.value = r2.value
    }
```

```
def term
    int value
    [term '*' factor]
    {
        lhs.value = r1.value * r3.value
    }
|   [factor]
    {
        lhs.value = r1.value
    }

def expr
    int value
    [expr '+' term]
    {
        lhs.value = r1.value + r3.value
    }
|   [term]
    {
        lhs.value = r1.value
    }
```

Figure 4.14: An example of context-free grammar definitions with attributes and reduction actions. The expression value is computed during parsing by passing results up the tree.

```
extended_type_ref  → 'int' | 'str' | 'bool' | 'any'

generic_def        → 'map' identifier '[' type_ref type_ref ']'
                   | 'list' identifier '[' type_ref ']'
```

Figure 4.15: Syntax of extended types.

We have also added two additional type definition statements, `list` and `map`. The `list` definition statement can be used to define a variable-length list of any single type. The `map` definition statement can be used to define a dictionary that maps keys of any single type to values of any single type. The syntax of these type definition statements is a subset of the context-free type definition statement, only using different keywords in place of `def`.

Finally, we add one more extended type to our language to assist with the handling of trees. The `any` keyword can be used in place of real types to declare variables that hold a

tree of any type. This is useful for writing generic transformations (Section 4.8). Note that none of the extended types have a textual pattern and therefore they cannot appear in a production of a context-free type definition.

We have extended the language with extra types. However, we do not deviate from the tree-based value semantics. We have made every type of data a tree-based value. For example, an integer is a non-parsable terminal with an integral value instead of a string value. The primary advantage is that it allows dynamic typing over all values. Inbuilt algorithms such as search and print can be applied to any value, including integers, strings, list elements and map elements. Very few exceptions need be made in the semantics and the implementation.

The downside to using a tree type to represent all data is that it incurs a base storage cost much higher than systems based on machine-level words. On most computer hardware integers can be represented using only four or eight bytes. By encapsulating integers and other small ordinal types inside trees we drive the cost up. In the future it may be worthwhile to add primitive non-tree types to the language, for the purpose of optimization.

In Figure 4.16 we show how these generic types can be used to create a symbol table. This symbol table is for a fictitious language that allows a single name to represent different language objects. For example, the identifier `C` may represent both a class and a function that constructs a class of that type. The symbol table maps a name to a list of symbol table entries. This data structure is known as a multimap. In the example we create the symbol table and initialize it with two inbuilt types.

4.5 Matching Patterns

Patterns are an essential part of every transformation system. They are used for testing forms of trees and capturing variables. In Figure 4.17 we give the syntax of the pattern-related portion of our language. The match statement is a primary expression that tests a

```
# An entry describing a symbol.
def symbol_entry
    str type_class
    []

# A symbol may resolve to more than
# one symbol entry.
list symbol_list
    [symbol_entry]

# Map names to a list of entries.
map symbol_table
    [str symbol_list]

# Initialize the symbol table.
symbol_table SymbolTable =
    construct symbol_table []

# A list of entries containing only inbuilt.
symbol_list InbuiltList = construct symbol_list []
InbuiltList.append( construct symbol_entry( type_class: 'inbuilt' ) []  )

# Add inbuilt types int and bool.
SymbolTable.insert( 'int', InbuiltList )
SymbolTable.insert( 'bool', InbuiltList )
```

Figure 4.16: A symbol table for a language that allows names to have more than one meaning. Names map to the list of entities that have been defined using the name.

---

variable against a pattern.

Patterns must have a type associated with them. In the case of the match statement the type is taken from the variable that is being tested. The pattern content consists of a sequence of literal text strings, named types and variable references. The named types may have a label, which indicates that following a match a capture is to be made. Variable references in the pattern allow the user to require that a fragment of the input tree exactly matches a known value.

At compile-time the system's context-free parsing algorithm is applied to the entire list of items, with the literal text scanned on demand and the variables resolved to types. The

```
match_primary      → 'match' var_ref pattern_list

pattern_list       → pattern*

pattern            → '"' litpat_el* '"'
                     | '[' pattern_el* ']'

litpat_el          → '[' pattern_el* ']'
                     | pattern_char                          (§4.2.3)

pattern_el         → '"' litpat_el* '"'
                     | label? type_or_lit
                     | '?=' identifier

type_or_lit        → identifier repeat_op?
                     | single_literal repeat_op?

label              → identifier ':'
```

Figure 4.17: Syntax of patterns.

named types and the types that have been resolved from the variable references are treated as terminals, but they retain their nonterminal identity. For example, an occurrence of `expr` is sent to the parser as a token that represents the `expr` nonterminal. It is accepted by the parser as an instance of the `expr` nonterminal with the child list unspecified.

At run-time the system tests the parsed form of the pattern against the input tree. The forms of the two trees must be identical, excluding the content underneath any subtrees of the input tree that match named types in the pattern. If a match occurs the next step is to test the form of any variables in the pattern against the subtree that the variable's type matched. In these comparisons the subtree of the input tree must exactly match the value given in the pattern. If this succeeds then the pattern match is considered a success. The labelled types in the pattern are bound to new variables and the match expression returns the input tree. If the match fails it returns `nil`.

In Figure 4.18 we show two semantically identical patterns that are expressed in different styles. The pattern matches an XML-style tag with id `person` and an attribute list that

```
lex start
{
    token id /[a-zA-Z_][a-zA-Z0-9_]*/
    literal '=', '<', '>', '/'
    ignore /[ \t\n\r\v]+/
}

def attr
    [id '=' id]

def open_tag
    ['<' id attr* '>']

def close_tag
    ['<' '/' id '>']

def tag
    [open_tag item* close_tag]

def item
    [tag]
|   [id]
```

```
tag Tag = parse tag( stdin )

# Style: list of literal text and types.
match Tag ["<person name=" Val1:id attr* ">" item* "</person>"]

# Style: literal text with embedded lists of types.
match Tag "<person name=[Val2:id attr*]>[item*]</person>"
```

Figure 4.18: Two semantically identical patterns expressed in different syntactic styles. The first pattern is a sequence of literal text strings and named items. The second is a literal text string with embedded sequences of named types.

---

begins with the `name` attribute. The value of the attribute is bound to a variable. In the first match statement, a list of literal strings and types is used. This style is appropriate if the pattern consists of mostly types with a few literal components. In the second match statement, a literal pattern with embedded lists of types is used. This style is appropriate if the pattern consists of mostly literal text and few named types.

4.6 Synthesizing Trees

While patterns are used for testing forms of trees and binding subtrees to variables, constructors are used for the opposite purpose, the synthesis of trees. Like patterns, constructors are

```
construct_primary  → 'construct' type_ref attr_init? syn_list
                   | 'make_tree' '(' expr_list? ')'
                   | 'make_token' '(' expr_list? ')'
                   | 'parse' type_ref '(' expr_list? ')'
                   | 'parse_stop' type_ref '(' expr_list? ')'
                   | 'reparse' type_ref '(' expr_list? ')'

attr_init          → '(' attr_assign_list? ')'

attr_assign_list   → attr_assign ( ',' attr_assign )*

attr_assign        → identifier ':' expr

syn_list           → synthesis*

synthesis          → '"' litsyn_el* '"'
                   | '[' syn_el* ']'

litsyn_el          → '[' syn_el* ']'
                   | pattern_char                          (§4.2.3)

syn_el             → '"' litsyn_el* '"'
                   | single_literal
                   | var_ref
```

Figure 4.19: Syntax of tree synthesis patterns.

scanned and parsed and therefore must have a type associated with them. Unlike patterns, constructors must specify complete trees. They cannot have any named types without an associated value. They are comprised of a list of literal text strings and variable references. The syntax is given in Figure 4.19.

The `construct` expression requires a type and a list of literal text strings and variables to make up the constructor pattern. A list of attribute assignments is optional. At compile-time the list of literal text and variables is parsed. The variables are resolved to type names and they are sent to the parser as tokens that represent nonterminals. The literal text is scanned on demand. At run-time a tree is allocated in the form that was produced by parsing the constructor. The nodes that represent the type names that were derived from variable references are initially empty. The values of the variables that were given in the

constructor are then copied into the tree.

The optional list of attribute assignments can be used to give a set of name-value pairs to initialize attributes of the tree. There is no requirement on the order of assignments or on the presence of any particular attribute initialization. Uninitialized attributes are given the value `nil`.

The `make_token` and `make_tree` statements can be used to construct terminals and non-terminals using an integer value to represent the type, rather than requiring that it be specified as a literal type. The integer value corresponding to a type can be extracted from the literal type name using the `typeid` expression (Section 4.3.2). The remainder of the arguments are used for the children/text of the tree, then any attributes.

The `parse`, `parse_stop` and `reparse` expressions are used to invoke the parsing engine. These statements require a type and an argument list. The `parse` expression consumes all remaining input from the stream. It requires one argument, the stream to pull input from. The `parse_stop` expression parses until a reduction of the parse type can be achieved. At this point it stops parsing, leaving the remaining input on the input stream. The `reparse` expression parses the literal text of some tree as another type. It requires one argument: the tree from which the literal text should be taken.

In Figure 4.20 we give an example of tree construction, using the XML-like grammar from Figure 4.18. It parses the input, takes the resulting tree apart, and constructs a new tree using a component of the input. A more robust example might check the result of the parse expression and the pattern match for errors. Like patterns, tree construction patterns can be expressed in different styles.

4.7 Traversing and Manipulating Trees

Constructing tree values in a bottom-up fashion using the `construct` expression is straightforward. What we need now is an ability to traverse and manipulate trees that have already

```
tag PersonTag = parse_stop tag( stdin )

match PersonTag
    ["<person name=" Val:id attr* ">" item* "</person>"]

tag NameTag1 = construct tag
    ["<name type=person>" Val "</name>"]

tag NameTag2 = construct tag
    "<name type=person>[Val]</name>"
```

Figure 4.20: An example of tree synthesis using construct. The input is parsed and deconstructed, then two new trees are constructed using a constituent of the input tree. The two construct expressions demonstrate the different synthesis pattern styles that are possible.

---

been constructed, or otherwise produced by the parser. This requires language constructs that can give us the same effect as a cursor: we need a read/write head that we can move about a tree and use to analyze or manipulate trees.

4.7.1 References

The first construct that we add in support of traversing and manipulating trees is the reference parameter. Function arguments are normally passed by value and this is consistent with variable assignment. This is the right choice for our tree manipulation language, however it does not support the creation of procedures whose purpose is to manipulate trees. We therefore allow for arguments to be explicitly passed by reference. This is done by placing the `ref` keyword in front of the type of a function parameter. Manipulating a parameter that has been passed by reference results in changes to the value of the variable given at the call point. Figure 4.21 shows the section of the grammar relevant to reference parameters.

Reference parameters allow us to define a procedure that encapsulates the manipulation of a tree. Next we need a tool for moving about a tree.

```
ref_param_var_def  → ref_type_ref identifier

ref_type_ref       → 'ref' identifier
```

Figure 4.21: Syntax of reference parameters.

```
for_iter_stmt      → 'for' identifier ':' type_ref 'in' iter_call block_or_single

if_iter_stmt       → 'if' identifier ':' type_ref 'in' iter_call block_or_single

iter_call          → var_ref '(' expr_list? ')'
                   | identifier
```

Figure 4.22: Syntax of iterator control structures.

4.7.2 Iterators

For the purpose of traversing and manipulating trees we add iterators. An iterator is a coroutine that yields a sequence of references to trees. Each time the iterator is suspended it yields the next reference in the sequence or it yields `nil`. Manipulating a reference returned from an iterator results in a change to the variable that was yielded by the iterator.

Iterators may be inbuilt or defined by the user (Section 4.7.3). The inbuilt iterators provide the basic tree traversal algorithms such as `topdown`, `child` and `rev_child`. These iterators yield references to the subtrees of some root tree. The user-defined iterators provide more freedom, leaving the specification of what is yielded up to the user. They enable the writing of reusable search algorithms.

Iterators can be used only by the designated loop and if-statement control structures. These control structures, shown in Figure 4.22, instantiate the iterator context and repeatedly invoke it until it yields `nil`, at which point they break out of the iteration loop. Using iterators in this way we can statically guarantee that the value semantics is not violated. For example, by analyzing the control structure we can prevent a single variable from simultaneously being used as the root of more than one iterator. This is discussed more in

```
# The default iterator, internally defined.
iter topdown( ref any Tree )

for I: type in topdown(Tree) {
    if match I [some pattern] {
        I = construct type
            [Some Replacement]
    }
}
```

Figure 4.23: Using an iterator to traverse and manipulate a tree. The `topdown` iterator is the default iterator, which is used when no iterator name is given.

---

the implementation section (4.10).

The for iterator and the if iterator statements are comprised of a new variable, a type, and an iterator call. The new variable provides access to the yielded references. The given type is enforced on each iteration. The iterator may yield any type, but the body of the iterator is executed only when the type of the tree yielded matches the type given.

The iterator call specifies what to iterate and how. If only a single name is given then it is treated as the first argument to the default iterator. If a name and expression list is given then the explicitly named iterator is invoked with the expressions passed as arguments. Presumably, the root of the iteration is one of these arguments. If the current element of the iterator is modified in the control structure body, then the root parameter should be a reference parameter.

In the example in Figure 4.23 the `topdown` iterator takes a reference argument and yields references to trees contained in that argument. The body of the `topdown` iterator is not shown (see Figure 4.25). A top-down, left-right traversal is the default iteration strategy, and the effect of giving only a tree expression instead of a full iterator call is a call to this iterator.

```
iter_def        → 'iter' identifier '(' param_list? ')' '{' stmt_list '}'

yield_stmt      → 'yield' var_ref
```

Figure 4.24: Syntax of user-defined iterators.

4.7.3 User-Defined Iterators

User-defined iterators allow custom traversals to be written. A user-defined iterator looks like a function, only it does not have any return type, nor does it support the return statement. Instead, it allows the yield statement. This statement suspends the iterator and provides a reference to a variable to the calling context. The loop executing the iterator is responsible for enforcing the type of the tree yielded. When the iterator yields `nil` it is terminated. If control reaches the end of the iterator, `nil` is automatically yielded. The syntax of user-defined iterators is shown in Figure 4.24.

4.8 Generic Transformation Idioms

By putting together reference parameters, iterators, iterator loops and the `any` type, we can program arbitrary generic tree traversals. For example, a reference argument that was passed to a user-defined iterator can then be passed to another iterator as its root. A reference yielded from that iterator can then be passed to a function as a reference argument, used as the root of another iterator, or yielded from the original user-defined iterator. This allows us freedom to compose traversals in stages and abstract them. Furthermore, the use of references allows these abstract traversals to be used to write to trees.

4.8.1 Top-Down, Left-Right, One-Pass

The default iteration strategy is to make a single top-down, left-right pass over a tree. If the current node of the tree is replaced during the course of iteration, then the replacement's

```
int do_topdown_leftright( ref any T )
{
    for C: any in child(T) {
        yield C
        do_topdown_leftright( C )
    }
}

iter topdown_leftright( ref any T )
{
    do_topdown_leftright( T )
}

for T: tree in topdown_leftright(S) {
    # Do something.
    T = construct tree
        [Some Replacement]
    # Iterator proceeds inside SomeReplacement.
}
```

Figure 4.25: A user-defined iterator that implements the top-down, left-right, one-pass search strategy of the default iterator. This is the search strategy of TXL functions.

---

children are immediately traversed. The implication of this is that it is possible to construct an infinitely deep tree. Such an iteration will not terminate on its own.

The top-down, left-right, one-pass iterator is inbuilt. In Figure 4.25 we show its implementation using the `child` iterator. The `child` iterator yields the immediate children of a tree from left to right. When combined with a recursive call, a top-down, left-right traversal is achieved.

As shown in Figure 4.25, the `yield` statement does not need to be used directly in the body of a user-defined iterator. It can be used in a subroutine that is called by an iterator. The `yield` then suspends the calling iterator. Judicious language designers might note that we have a function that has meaning in the context of an iterator. However, there is no syntactic link between the function and the iterator. This is an issue that we acknowledge and hope to address in future work.

```
iter fixed_point( ref any T )
{
    bool modified = true
    while ( modified ) {
        modified = false
        for C: any in T {
            yield C

            if ( this_iter_modified() ) {
                modified = true
                break
            }
        }
    }
}

for C: tree in fixed_point( T ) {
    if match C [pattern]
        C = construct tree [Some Replacement]
}
```

Figure 4.26: A user-defined iterator that implements fixed-point iteration, the default rule application strategy in many transformation systems.

4.8.2 Fixed-Point Iteration

Many transformation systems employ fixed-point iteration as the default search strategy. We can construct a fixed-point iterator using the default one-pass iterator and an inbuilt function that provides some insight into how the iterator has been utilized during iteration. The `this_iter_modified()` inbuilt function tells us if the user replaced the current node of the iterator in between the yield and the iterator resume. In Figure 4.26 we show how we can use this function to implement a fixed-point iterator. The fixed-point iterator traverses the tree using a one-pass iterator and when a replacement occurs the one-pass iterator is restarted. If a full pass of the tree happens without any modification then the iterator terminates.

```
int do_bottomup_leftright( ref any T )
{
    for C: any in child(T) {
        do_bottomup_leftright( C )
        yield C
    }
}

iter bottomup_leftright( ref any T )
{
    do_bottomup_leftright( T )
}
```

Figure 4.27: A user-defined iterator that implements a bottom-up, left-right traversal.

4.8.3 Bottom-Up Traversals

The default iterator is a top-down iterator. Sometimes bottom-up traversals are required and these can be programmed in the usual recursive way: first recursing on children of a tree, then visiting the tree. In Figure 4.27 we show the implementation of a generic bottom-up traversal.

4.8.4 Right-Left Traversals

By using an iterator with a recursive call we are also able to implement right-left traversals. We just need access to the immediate children of a tree in the reverse direction. We use the `rev_child` inbuilt iterator for this. Right-left traversals can be written going top-down or bottom-up. In Figure 4.28 we show a top-down, right-left traversal.

4.9 Transformation Example: Goto Elimination

In this section we give an example of a real transformation on the Tiny Imperative Language (TIL) [108]. This small language has been designed for benchmarking transformation languages. A section of the TIL grammar relevant to the transformation example is shown

```
int do_topdown_rightleft( ref any T )
{
    for C: any in rev_child(T) {
        yield C
        do_topdown_rightleft( C )
    }
}

iter topdown_rightleft( ref any T )
{
    do_topdown_rightleft( T )
}
```

Figure 4.28: A user-defined iterator that implements a top-down, right-left traversal.

in Figure 4.29.

Our transformation example shows just one case of goto elimination. The transformation looks for a label and then looks through the statements that follow for a jump back up to the label that is guarded by a condition. The statements in between are put inside a do/while block. The TIL language does not contain a do/while block. However, we have added it for the purpose of this example. The transformation is shown in Figure 4.30.

4.10 Implementation Notes

In this section we give some notes on our implementation choices. We first discuss our approach to internal tree sharing for the purpose of reducing memory consumption. We then discuss the data records that we use to implement shared trees. We explain how shared trees are copied when they are written to during iteration, as required by the value semantics. We then describe a language restriction that simplifies our implementation. Finally, we give a brief overview of the virtual machine.

```
def statement
    [declaration] | [assignment_statement] | [if_statement] | [while_statement] |
    [do_statement] | [for_statement] | [read_statement] | [write_statement] |
    [labelled_statement] | [goto_statement]

def if_statement
    ['if' expression 'then' statement* else_statement? 'end']

def else_statement
    ['else' statement*]

def do_statement
    ['do' statement* 'while' expression ';']

def labelled_statement
    [id ':' statement]

def goto_statement
    ['goto' id ';']
```

Figure 4.29: A section of the Tiny Imperative Language (TIL) grammar relevant to the goto-elimination example in Figure 4.30.

4.10.1 Tree Sharing

The naive approach to implementing value semantics for trees is to fully copy a tree whenever a value must be copied. This would result in excessive use of memory. Instead, some technique for reducing memory usage should be employed. The ATerm library, which is used by Stratego [107] and ASF+SDF [97, 99], employs maximal subtree sharing. It guarantees maximal sharing by hashing unique trees; before constructing a new tree it determines if an identical tree already exists and uses it instead [98].

The TXL [25] engine uses partial tree sharing. It statically determines when it is safe to share trees and does so without using any run-time reference counts or hash consing. The functional nature of the TXL programming language allows such optimizations to be used.

Our approach is to use run-time reference counting to implement multiple-readers with a copy-on-write policy. Trees are always shared when a copy of a value is taken and the

```
# Lists are right recursive.
for S: statement* in Program
{
    if match S [Label: id ':'
            First: statement
            Rest: statement*]
    {
        expression Expr = nil
        statement* Following = nil

        # Look though the remaining statements for a goto back to the label.
        # The repeat iterator yields only top-level statement lists. It
        # restricts our search to the same nesting depth as the label.
        for Check: statement* in repeat(Rest)
        {
            if match Check
                ['if' E: expression 'then'
                    'goto' ?=Label ';'
                'end'
                SL: statement*]
            {
                Expr = E
                Following = SL

                # Check iterates over tails of Rest. Assigning an empty list
                # to Check truncates the Rest list. What we cut off is saved in
                # Following (excluding the if statement).
                Check = construct statement* []
            }
        }

        # If a goto was found, then perform the rewrite.
        if ( Expr != nil )
        {
            # Replace the labelled statement through to the goto
            # with a do ... while.
            S = construct statement*
                ['do'
                    First
                    Rest
                'while' Expr ';'
                Following]
        }
    }
}
```

Figure 4.30: A transformation example that performs one case of goto elimination. The transformation looks for labels, scans the following statements for a conditional jump back to the label, then replaces the enclosed block with a `do ... while`.

actual copying is delayed until absolutely necessary.

4.10.2 The Tree-Kid Pair

We use a combination of two data records to build a shared tree structure. The `Tree` node represents the data of a particular tree node. It is reference counted and multiple pointers may point to it, provided that the reference count is maintained. The `Kid` node is used to build a linked list of children underneath a tree. The `Kid` node may not be shared. Its size is kept to a minimum: just a pointer to a `Tree` and a pointer to the next kid. Using the tree-kid pair we can easily share trees by turning the copy operation into an upref operation and enforcing copy-on-write rules.

Figure 4.31 shows an example of tree sharing using trees and kids. There are two tree values. Invisible to the user is the fact that three of the `Tree` records involved in the representation of these two trees are shared. A tree such as this could come into existence by a number of different bottom-up tree constructions.

The parsing algorithm produces trees in this format at every step. When parsing is complete the tree need not be transformed to be made accessible to the transformation language. This also allows us to do transformation during parsing and incorporate the result of a transformation into a tree that is produced by the parser.

4.10.3 Reference Chains

The language features that we have for manipulating trees require that we take references to trees. These references are used as a location for read and write operations. Since trees may be shared we must first copy a tree fragment that is in use by other values before we modify it.

We cannot use only reference counting to determine if any given tree is shared. A tree record may have only one parent, but some tree records higher up in the ancestry may be shared. For this reason, a reference to a tree record must be a chain of references, recording

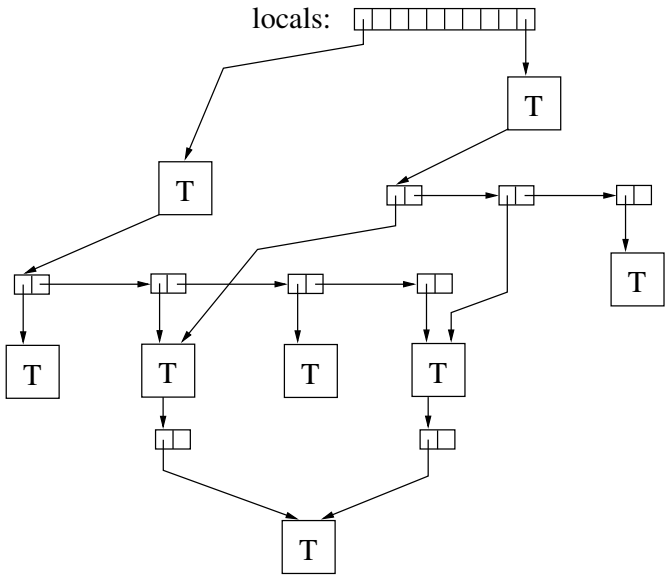

Figure 4.31: An example of tree sharing using the Tree and Kid data records. There are two values and three of the subtrees are shared.

---

the history of the traversal all the way up to the root variable where it was started.

Figure 4.32 gives an example of a reference chain. The chain was originally rooted at the local variable T_1. The traversal that yielded this chain went down to the second child T_2 and then down to the first child T_3 from there.

When the end of the chain is written to, the chain is traversed upwards to its root. From the root, the chain is traversed back down and the reference count of each tree record along the path is examined. When a tree record that has a reference count greater than one is found it is copied. The copy-on-write procedure then keeps moving down the chain copying each record until it arrives back at the end of the chain where we need to make a modification. When finished, the path from the root of the reference chain to the leaf will contain only trees that are referenced once. The example in Figure 4.33 shows the result of the copy-on-write procedure after it has made it safe to update $T3$.

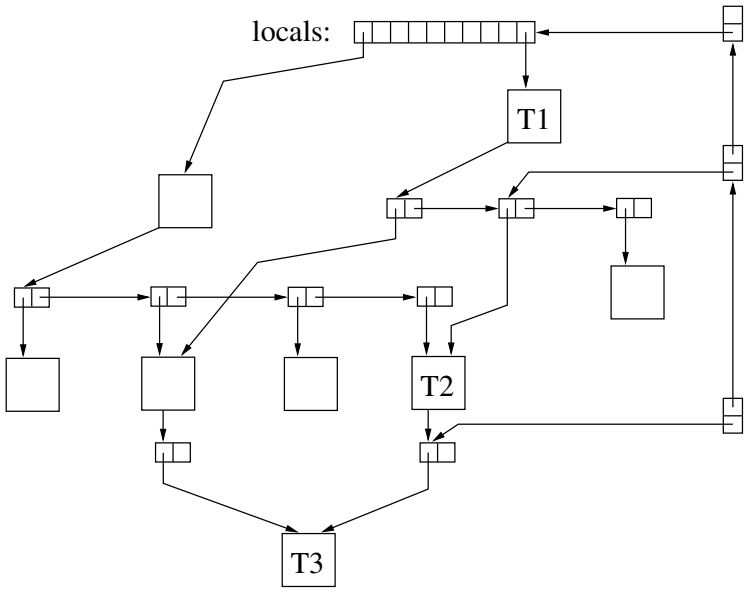

Figure 4.32: An example of a reference chain resulting from a traversal of a shared tree. Reference chains record the traversal history of iterators and the source of reference parameters.

Note that reference chains are restricted for use by the implementation; they are not exposed as first-class values. The compiler is then able to ensure that they are used safely and it can efficiently implement them on the stack.

The idea of making a distinction between structures that represent data and pointers that are equipped with an ability to move about the data is not new. It has been used as an implementation technique for functional languages. This idea is described in more depth by O'Keefe [75].

4.10.4 Implementation-Caused Language Restrictions

Maintaining value semantics requires that we place a restriction on how reference chains are used. Once a reference to a variable is taken, and for as long as that reference remains active, the variable must not be written to. The reference must activate a write lock and

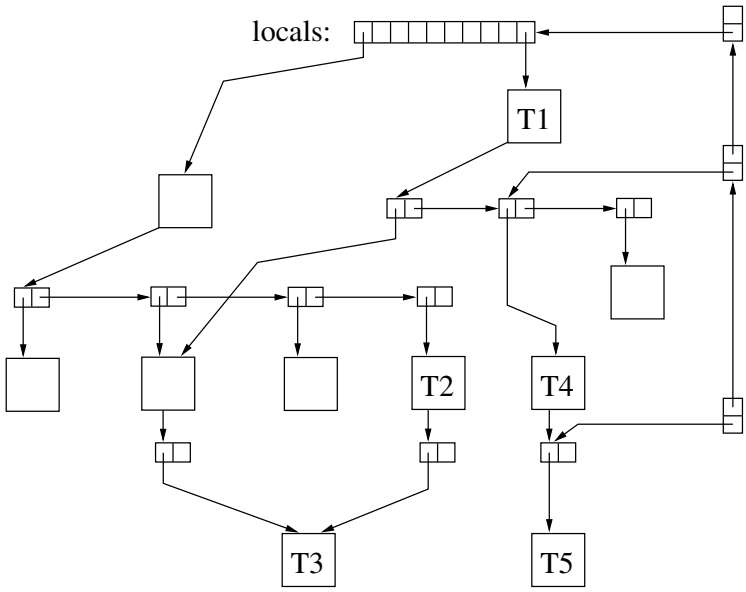

Figure 4.33: The result of the copy-on-write procedure after it has made it safe to write to the end of the reference chain in Figure 4.32.

---

become the primary method by which the variable is written to. If we allow trees to be modified when a reference is active, then the copy-on-write procedure could easily cause two references that have a parent-child relationship to point to tree records that do not have a parent-child relationship, thus rendering the reference chain invalid.

Write-locking local variables that have active reference chains is straightforward. The compiler can easily use static analysis to prevent referenced local variables from being accessed for writing. This is not so easy for globally accessible variables, however. Because of the added complexity of this kind of static analysis, we do not allow references of globals or dynamic variables to be taken. In Section 5.5.5 we describe an additional need for this restriction.

```
case IN_TST_EQL: {
    Tree *o2 = pop();
    Tree *o1 = pop();
    long r = cmp_tree( o1, o2 );
    Tree *val = r ? program->falseVal : program->trueVal;
    tree_upref( val );
    push( val );
    tree_downref( program, o1 );
    tree_downref( program, o2 );
    break;
}
case IN_JMP_FALSE: {
    short dist;
    read_half( dist );

    Tree *tree = pop();
    if ( test_false( program, tree ) )
        instr += dist;
    tree_downref( program, tree );
    break;
}
```

Figure 4.34: Implementation of two virtual machine instructions. Trees stored on the stack are considered referenced by the stack. Reference counts must be incremented before a push operation and decremented after a pop operation and subsequent use by the instruction.

4.10.5 The Virtual Machine

We employ a stack-based virtual machine, allowing for a simple compiler and VM implementation. The instruction set is variable width, with the size of each instruction dependent on the number and size of the required arguments. Instruction arguments are encoded into the bytecode following the instruction identifier code. Instructions generally operate on trees and therefore they pop and then push pointers to the `Tree` data structure. When a pointer to a `Tree` is loaded onto the stack, the reference count must first be increased. After it is popped and used by an instruction, the reference count can be decreased.

In Figure 4.34 we give an example of two VM bytecodes. The `IN_TST_EQL` instruction pops two trees from the top of the stack and compares them. If they are equal it leaves the value `true` on the top of the stack, otherwise it leaves `false` on the top of the stack. The

instruction does not have any arguments. The `IN_JMP_FALSE` instruction takes the jump distance as a two-byte argument. It pops the value to test from the stack and adjusts the instruction pointer if it is false.

4.11 Summary

In this chapter we present the design of a new transformation language. In the next chapter we will see that we need a new transformation language implementation. We choose to also design a new language, with the goal of making a language that contains constructs common in general-purpose languages and transformation languages. We hope the new language will make transformation languages more accessible, as well as ease the writing of program components that are not well suited to the rewriting paradigm.

The language definition begins in Section 4.2, where we give a general overview. In Section 4.3 we define the language features that we have taken from general-purpose languages. To this base we add the essential transformation language features: tree pattern matching in Section 4.5, tree synthesis in Section 4.6, and tree searching in Section 4.7. In Section 4.8 we give examples of generic tree traversals that we can express using our tree traversal constructs. In Section 4.9 we give an example of a transformation.

In the last section of this chapter, Section 4.10, we briefly discuss some aspects of our implementation. This is for the reader's interest, but it also provides some background for what is to come. At this point in the work our transformation system is not complete. We have a generalized parsing engine and we have a transformation language, but we do not yet have a generalized context-dependent parsing engine. Completing the system will require making some modifications to the implementation, and we do this in the following chapter.

Chapter 5

Context-Dependent Parsing

In Chapter 2 we described a generalized context-free parsing algorithm known as backtracking LR and in Chapter 3 we enhanced it to support ordered choice and grammar-dependent lexical analysis. In Chapter 4 we described a new transformation language that merges transformation constructs with a base language inspired by general-purpose programming languages. While interesting and different, so far our system is not any more capable than what exists in the current suite of transformation systems. In this chapter we push ahead of existing systems by extending our generalized parsing engine with an ability to handle languages that are both ambiguous and context-dependent.

Our approach is to start by adding token-generation and semantic actions to the parsing engine. These actions are familiar to the users of the deterministic parsing systems Lex [61] and Yacc [46]. While it is possible to use these actions to make local modifications to parse trees while the trees are acquired, that is not the purpose. We wish to encourage users to modify global data structures, then use these structures to influence the parse either through feedback to the lexical phase or semantic conditions in the reduction actions.

Encouraging the modification of global data in a generalized parser creates a data consistency problem that we must solve. We add undo actions that are executed by the parse engine as it backtracks over failed parse attempts. In these undo actions the modifications to

global data can be reverted in order to ensure that the global state remains consistent. We then automate the process with reverse execution. During forward execution, changes that are visible globally are logged. During backtracking they can be automatically reverted.

5.1 Token-Generation Actions

Many programming languages require a scanner that can look up the meaning of identifiers to determine their type before they are sent to the parser. The established practice for dealing with this problem is to use lexical feedback. During forward parsing, semantic actions are responsible for maintaining lookup tables. The lexical analyzer is then responsible for querying the type of an identifier before sending it to the parser. We have many tools that support this method of parsing.

The Haskell programming language allows users to define operators, their precedence, and their associativity. Parsing user-defined operators can be accomplished by applying the lexical feedback technique to sequences of symbols and generating operator tokens with the appropriate precedence level and associativity.

Other modern programming languages rely on an ability to programmatically generate tokens. The off-side rules of Python are an example of this. The whitespace at the beginning of a line must be analyzed, and based on how it compares to the previous line, some number of indent or dedent tokens must be sent to the parser. Off-side rules require an ability to use language-specific program logic to decide how many tokens to send. We need to be able to send none at all, or some arbitrarily large finite number.

Other parsing problems require that we generate tokens with a size previously specified in the input. For example, many network protocols have fragments that carry data payloads. The length of the payload appears somewhere ahead of the payload. This requires that we be able to programmatically decide how much of the input is used to construct a token. It should be possible to consume no characters, or some arbitrarily large number of characters.

Some programming languages allow for nested comments. Normally, nested comments are parsed by augmenting the scanner with hand-written code that keeps track of the nesting depth. Another possibility is to manually invoke a context-free parser from the scanner. This relies on an ability to recursively invoke the parsing engine.

In other cases, we may want to treat specific characters as significant only in certain semantic contexts. We should be able to match a pattern, then decide, using language-specific program logic, whether or not the matched token is to be ignored or sent as a token.

To address all of these demands we allow blocks of code to be associated with token patterns. When a token pattern with a generation block matches the input, the block of code is executed and it becomes responsible for sending tokens to the parser. It may send any number of tokens. Tokens may be constructed by pulling text from the input stream, parsing the input stream, or using immediate strings. Token types may be given literally or they may be specified as an integer value. Attributes may be used to pass additional information with the token.

In Section 4.4.1 we give the syntax of token-generation actions and in Figure 5.1 we give an example. A symbol is looked up in the symbol table and a token of the corresponding type is sent to the parser. The text of the match is manually removed from the token stream and used as an argument to the token construction function `make_token`.

5.2 Reduction Actions

Reduction actions are blocks of code that are associated with a production of a type definition. Following the recognition of the right-hand side of the production, the elements are popped from the stack and replaced with the nonterminal. In the process the reduction action is executed. Reduction actions are common in bottom-up parser generators such as Yacc. They are used for building abstract syntax trees, performing semantic analysis and

```
# Types to translate to.
token declaration_ref //
token type_ref //
token function_ref //

# The symbol table maps identifiers to numerical values representing
# the above terminal types.
map symbol_table [str int]
symbol_table SymbolTable

token id
    /( [a-zA-Z_] [a-zA-Z0-9_]* )/
    {
        # The match_text inbuilt variable contains the text that was matched by
        # the pattern. We assume that the symbol table has been populated with
        # entries that each link an identifier to a numerical value that
        # represents the type of the identifier. These numerical values are
        # implementation-defined and can be derived from the literal types
        # using the 'typeid' primary expression.
        int TypeId = SymbolTable.find( match_text )

        # The make_token function constructs a token from a
        # type number and a string.
        any Token = make_token( TypeId, pull(stdin, match_length) )
        send( Token )
    }
```

Figure 5.1: An example of a token-generation block. This action looks up the type of an identifier in a symbol table, then creates and sends a new token of that type.

---

maintaining global state.

In this thesis we are interested in the use of semantic actions to maintain global state. When just using reduction actions for this purpose they tend to be brief and infrequent, but critical for acquiring a correct parse. Examples of the tasks that these reduction actions perform include inserting or deleting dictionary items, attaching or detaching list items, and pushing or popping stack items.

Since we may need to unparse a reduced node, we always preserve the children of a reduction. This results in a stack of parse trees. Since we are parsing for transformation and probably going to be manipulating the resulting tree, we do not worry about the waste

that keeping all subtrees implies. We treat it as a necessary cost.

Reduction actions have access to both the left-hand side of the reduction (`lhs`) and the right-hand-side elements (`r1`, `r2`, `r3` ... `rN`). The `lhs` variable may be replaced or modified, which affects the result of the parse. Right-hand-side elements are distinct values; modifying them will not result in the modification of the `lhs` value or the returned parse tree. To avoid confusion the right-hand-side elements are made constant.

In Section 4.4.2 the syntax of reduction actions is shown. Below we give an example of a reduction action. It is used to insert a symbol into a symbol table, which supports the lookup example in the previous section.

```
def declaration
    ['var' identifier ';']
    {
        SymbolTable.insert( r2, typeid declaration_ref )
    }
```

5.3 Semantic Predicates

In Section 4.4.2 the reject statement is defined. This statement causes the parser to act as if a parse error has occurred and begin backtracking. This statement can be used to implement semantic predicates [38], which are tests that query the state of the parse and use the result of the test to disambiguate the parse. In our system this is accomplished by rejecting the current reduction and invoking the backtracker.

```
def something
    ['a' production '!']
    {
        if ( !someCondition() )
            reject
    }
```

5.4 Undoing Generation and Reduction Actions

So far we have described a system that supports generalized parsing and the building of global data structures during scanning and parsing. But in the capacity described thus far, the features enabling the manipulation of global data can work only with deterministic grammars. Our system needs to be augmented with support for nondeterministic grammars if we are to encourage the use of generation and reduction actions for the manipulation of global data.

When the parser backtracks over some action that has modified the global state we need the changes to the global state to be undone. To enable this we add undo actions to the parsing algorithm. When the parser backtracks it replaces a nonterminal with the child nodes it previously reduced from. At this time an undo action can be executed. When the parser sends a token back to the input stream, an undo action for the token-generation action that was used to construct the token can be executed. If undo actions exactly revert any changes to the global state that the forward reduction actions made, we are free to have a generalized parser with token-generation and reduction actions that manipulate global data structures.

The backtracking portion of our algorithm now behaves as follows. On each iteration of the unparsing loop, one item is popped from the top of the parse stack. If the node is a token it is pushed back to the input stream. If the token was generated by a token-generation action, the undo action for the generation action is executed. If the popped node is a nonterminal, the production's undo action is executed, the node is discarded, and the children of the node are pushed onto the parse stack. In both cases, if the popped node contained an alternate action then unparsing terminates and forward parsing resumes, with the initial action to take determined by the next alternate parsing decision that was stored in the popped node.

If a forward action sets a global variable to a new value, the reverse action must restore

```
def declaration
    str toRemove
    ['var' identifier ';']
    {
        bool wasInserted = SymbolTable.insert( r2, typeid declaration_ref )
        if ( wasInserted )
            lhs.toRemove = r2
    }
    __undo {
        # If a symbol was inserted, remove it. Note that attributes are
        # initialized to nil.
        if ( lhs.toRemove != nil )
            SymbolTable.remove( lhs.toRemove )
    }
```

Figure 5.2: A reduction action that inserts a symbol into a symbol table. The corresponding undo action removes the symbol if it was successfully inserted.

---

the old value. If a forward parsing action adds some value to a dictionary, the reverse action must remove it. If a forward action pops some value from a stack, the reverse action must push it back. Provided that proper housekeeping is done, the addition of undo actions can permit the parser to backtrack over productions that have modified the global state in arbitrary ways.

In Figure 5.2 we show how an undo action can be used to revert changes to the global state during backtracking. Note that the `__undo` block is not part of our language. We show it here to illustrate what must take place in order for global-state modifications to work in the presence of backtracking. In the next section we describe our final solution, which is to automate the process.

User-written reduction actions are normally very short. The amount of code that modifies global variables during parsing for the purpose of feedback usually amounts to just a few instructions. If a global variable is set or a data structure is modified, it is often easy to implement the reverse action. But even though reverse actions are normally short, it is easy for mistakes to creep in. It is critical that the reverse code be correct, otherwise the global

state becomes incorrect with respect to the context-free state of the parse. This results in spurious parse errors later on.

Most undo actions are trivial. However, in some cases undo actions are more complicated, especially when the forward actions contain conditional branches. The task gets more complicated still, when these become nested within each other. In other cases, there are dependencies between variables that must be respected by reverting the forward changes in the opposite order. Consider the reduction action in Figure 5.3. This reduction action is a part of a C++ grammar and it is the largest reduction action in the entire grammar. The `declarator_id` defines the new name in most declarations, including variables, functions, typedefs, and template variables and functions.

In Figure 5.4 the undo action for `declarator_id` is given. The action to take depends on what type of object was declared in the forward reduction action. The logic of the forward action must therefore be reproduced in enough detail to ensure that for every case of the action, the correct reverse modifications are made during backtracking. In this example the reverse modifications are performed in the opposite order, though it is not required here because there are no dependencies between the modifications of global state.

5.5 Automating Undo Actions with Reverse Execution

Instead of asking the user to write undo actions, we wish to automatically revert changes to the global state during backtracking. The advantages of this are that it guarantees undo actions to be free of errors and it saves the user methodical work that is amenable to automation.

5.5.1 Reverse Execution

Forward program execution takes a program through a sequence of states. A program that is able to execute in reverse is one that can stop and move backwards through the sequence

```
# Namespaces: declaration and lookup.
global object_list declNs
global object_list lookupNs

# Declaration and declarator data accumulated during parsing.
global declaration_data_list declarationData
global declarator_data_list declaratorData

def declarator_id
    lang_object obj
    [declarator_id_forms]
    {
        str name = r1.lookupId.data
        ptr lang_object qualObj = r1.lookupId.qualObj
        ptr lang_object parentObj = declNs.top
        if ( qualObj )
            parentObj = qualObj

        # Decide if we are declaring a constructor or a destructor.
        bool isConstructor
        if ( parentObj == r1.lookupId.obj )
            isConstructor = true

        if ( parentObj->specializationOf && parentObj->specializationOf == r1.lookupId.obj )
            isConstructor = true

        ptr lang_object obj = nil
        if ( name && !isConstructor && declarationData.top.isFriend == 0 ) {
            if ( declarationData.top.isTypedef ) {
                obj = createLangObject( TypedefType, name, lookupNs.top )
                obj->typedefOf = declarationData.top.typeObj
                insertObject( parentObj, name, obj )
            }
            else if ( !qualObj ) {
                if ( declarationData.top.isTemplate )
                    obj = createLangObject( TemplateIdType, name, lookupNs.top )
                else
                    obj = createLangObject( IdType, name, lookupNs.top )
                insertObject( declNs.top, name, obj )
            }
        }

        # Store the new object for use by the undo action.
        lhs.obj = obj

        declaratorData.push( construct declarator_data (
                qualObj: qualObj, lookupObj: lookupNs.top ) [] )

        # If the declarator is qualified, push the qualification to the lookup
        # stack. Also save it in the declarator data so it can be passed to a
        # function body if needed.
        if ( qualObj ) {
            lookupNs.push( qualObj )
            declaratorData.top.lookupObj = qualObj
        }
    }
```

Figure 5.3: The `declarator_id` reduction action from a C++ grammar. This reduction action contains a number of global data manipulations that are dependent on local and global data. This makes writing a correct undo action difficult.

```
__undo {
    ptr lang_object qualObj = r1->lookupId.qualObj
    if ( qualObj )
        lookupNs.pop()

    declaratorData.pop()

    # If we made a new object, we must remove it.
    ptr lang_object namedObj = lhs.obj
    if ( namedObj ) {
        if ( namedObj->type == TypedefType ) {
            ptr lang_object parentObj = declNs.top
            if ( qualObj )
                parentObj = qualObj

            parentObj->remove( namedObj )
        }
        elsif ( namedObj->type == IdType || namedObj->type == TemplateIdType )
            declNs.top->remove( namedObj )
    }
}
```

Figure 5.4: Undo action for `declarator_id`. Though the number of fixes to the global state are few, the logic required to arrive at the correct changes is not simple.

---

of states that it has previously passed through, perhaps even backing up all the way to the initial state. Reverse execution is a convenient (but not always practical) method for implementing nondeterministic algorithms [34].

Research into reverse execution has been carried out primarily for the purpose of enhancing debuggers [1, 91, 15, 23, 2]. The motivation is to develop a debugger that can step backwards as well as forwards, allowing the user more freedom in locating the source of an error. Other uses include fault-tolerant programming [21], speculative programming [45] and program animation [14]. The Dynalab program animation environment allows students to visualize program execution in the forward and reverse directions. Reverse execution has also been suggested as a convenient solution to the problem of programming undo/redo features in interactive applications [20].

Ideally, a reverse execution system would perform the inverse function of each statement

to regenerate old states during backtracking. But due to destructive updates of memory, inverse functions do not always exist. A reverse execution system must be prepared to save old states during forward execution and restore them when needed.

The computation of inverse functions and the logging of states can be approached at different levels of program abstraction. Inverse functions can be computed on source-level statements and logging can be carried out on source-level variables [1, 15]. Inverse function computation and state saving and restoration can also happen at a lower levels. Destructive updates to the memory locations visible to bytecode can be logged and later restored [23, 2]. There has also been research into adding hardware support for reverse execution [91].

5.5.2 An Instruction-Logging Reverse Execution System

We have modified the compiler and virtual machine for our tree transformation language such that it automatically generates undo actions. We have taken a simple logging approach. During forward execution any virtual machine instructions that must be reverted log the changes that they make. The name and old value of variables are encoded by the virtual machine in reverse bytecodes, which can later be executed when the action must be undone. This approach works well because there are normally only a few instructions in token-generation and reduction actions that must be undone.

As the virtual machine moves forward through bytecodes it records the reverse instructions that must be executed. The list of statements that were executed in the forward direction must be executed in the opposite order during the undo action. The virtual machine uses a list of reverse instructions and always appends the latest reverse instruction to the end of the list. The reverse code is executed by traversing the list backwards.

Most source-level statements are implemented using a group of virtual machine instructions. For example, one instruction will load the tree representing the global state, the next instruction will load an attribute from this tree, and the final instruction will modify that tree. To revert the effect of a single statement, the reverse instructions for a statement

Forward Statements	Instructions Executed	Recorded Instructions and Length
`statement1`	`s1a, s1b, s1c`	`r1a, r1b, r1c, 3`
`statement2`	`s2a`	`r2a, 1`
`statement3`	`s3a, s3b`	`r3a, r3b, 2`

Reverse Instructions Executed
`r3a, r3b, r2a, r1a, r1b, r1c`

Figure 5.5: These tables show the order in which reverse instructions are written, grouped and executed for a sequence of statements.

must follow the same sequence of loads, and then at the end undo the operation. A reverse traversal of the reverse instruction list will not suffice for loading then reverting. The instructions that load a variable and revert it must be executed in the same order in which they were written in the reverse execution buffer.

We accomplish the grouping of instructions by writing group lengths at the end of each instruction group. This length can then be used by the reverse virtual machine to execute the instructions in one group in the forward direction, while overall execution moves backwards. The reverse virtual machine starts at the end of the list, reads a length l, jumps backwards by l bytes, executes in the forward direction until the l bytes are consumed, and then jumps back again by $l + 1$ bytes and repeats.

The tables in Figure 5.5 demonstrate the order that reverse instructions are logged and executed in. In the forward direction, shown in the first table, the reverse instructions are written in the same order as the corresponding forward instructions are executed. The group length is written at the end of each group. In the reverse direction, shown in the next table, the grouped instructions are executed in the forward direction. However, the groups are executed in reverse.

The position of an instruction within a group is predetermined. Instructions can either be at the beginning of a group, the middle, or the end. Each instruction that sets a reverse instruction knows whether to initialize the group length, increment it, or increment it and

```
case IN_LOAD_GLOBAL_WV: {
    tree_upref( program->global );
    push( program->global );

    /* Write the reverse instruction. */
    reverseCode.append(
            IN_LOAD_GLOBAL_BKT );
    rcodeUnitLen = 1;
    break;
}
case IN_GET_FIELD_WV: {
    short field;
    read_half( field );

    Tree *obj = pop();
    tree_downref( program, obj );

    Tree *notShared = get_field_copy(
            program, obj, field );
    tree_upref( notShared );
    push( notShared );

    /* Write the reverse instruction. */
    reverseCode.append( IN_GET_FIELD_BKT );
    reverseCode.appendHalf( field );
    rcodeUnitLen += 3;
    break;
}
```

```
case IN_SET_TOKEN_DATA_WV: {
    Tree *tree = pop();
    Tree *val = pop();

    Head *oldval = tree->tokdata;
    Head *head = ((Str*)val)->value;
    head_upref( head );
    tree->tokdata = head;

    /* Write the reverse instruction. */
    reverseCode.append(
            IN_SET_TOKEN_DATA_BKT );
    reverseCode.appendWord( (Word)oldval );
    rcodeUnitLen += 5;
    reverseCode.append( rcodeUnitLen );

    tree_downref( program, tree );
    tree_downref( program, val );
    break;
}
```

Figure 5.6: Three instructions that modify global state and therefore must log changes by generating reverse instructions. These instructions are always at the beginning, middle and the end of a group, respectively.

---

then write it out. For example, the load of the global object and the load of a pointer are both always at the beginning of a group. The load of a field for writing is always in the middle. The setting of a field is always at the end.

In Figure 5.6 the implementation of three virtual machine instructions is shown. The `IN_LOAD_GLOBAL_WV` instruction is always at the beginning of a group. After loading the global tree it writes the reverse instruction and then initializes the current group length to one byte. The `IN_GET_FIELD_WV` instruction is always preceded by some tree load. There are several variants of this instruction and this particular one is always used when the value that

```
def two_numbers
    int i       # Field number 0
    int j       # Field number 1
    []

global two_numbers TN =
    construct two_numbers []

def start
    [ident*]
    {
        TN.i = 0
        TN.j = 1
    }
```

Forward:

```
IN_LOAD_INT 0
IN_LOAD_GLOBAL_WV
IN_GET_FIELD_WV 0       # Load TN
IN_SET_FIELD_WV 0       # Set i
IN_LOAD_INT 1
IN_LOAD_GLOBAL_WV
IN_GET_FIELD_WV 0       # Load TN
IN_SET_FIELD_WV 1       # Set j
```

Reverse:

```
IN_LOAD_GLOBAL_BKT
IN_GET_FIELD_BKT 0      # Load TN
IN_SET_FIELD_BKT 1      # Restore j
IN_LOAD_GLOBAL_BKT
IN_GET_FIELD_BKT 0      # Load TN
IN_SET_FIELD_BKT 0      # Restore i
```

Figure 5.7: A small example of automatically generated reverse instructions. Reverse instructions that come from the same statement are executed in the order that they were logged, but at the statement level there are groups of instructions that are executed in the reverse order.

---

is loaded is going to be written to. Therefore this instruction is never at the end of a group. It writes the reverse instruction then increments the group length. The last instruction shown, `IN_SET_TOKEN_DATA_WV`, modifies the value on the top of the stack and then discards it. Therefore it is always at the end of a group. It appends the reverse instruction, increments the group length, then writes the group length to the reverse instruction buffer.

In the example in Figure 5.7 we give a small reduction action and a corresponding trace of bytecodes in the forward and reverse directions. The reduction action modifies two attributes, `i` and `j`, of a global variable `TN`. Following the example code, we show the virtual machine instructions that are executed in the forward direction and then the instructions that are executed in the reverse direction. Each group contains two loads, first the global tree, which contains all global variables, then the `TN` variable from the global tree. The third instruction modifies the appropriate attributes of `TN`.

5.5.3 Specializing Virtual Machine Instructions

Reverse execution of arbitrary programs is a costly activity. To reduce work we are selective about which state-change operations we generate reverse instructions for. Since we always revert entire token-generation and reduction actions, we only need to revert changes to global and dynamic variables. We never need to revert changes to local variables because we do not backtrack into the middle of a token-generation or reduction action.

Being selective about which state change operations are undone requires that we specialize bytecode instructions. The compiler analyzes statements and determines which modified variables are globally visible and therefore must be reverted during backtracking. From the static analysis the compiler decides which load and store bytecodes to use when generating the code.

The load bytecodes are categorized by the purpose of the load. If the purpose is to eventually read from the loaded value, then the suffix is `R`, for *read*. If the purpose is to eventually write, but the write will not need to be reverted, then the suffix is `WC`, for *write commit*. If the value is loaded for writing, and the write will need to be reverted, then the suffix is `WV`, for *write reversible*. The reverse load instructions have the suffix `BKT`.

The store bytecodes have three forms. If the store does not need to be undone during backtracking, as is the case with storing values into trees that are local to the current scope, then the suffix is `WC`. If the store will need to be undone during backtracking, then the suffix is `WV`. The reverse store instructions have the suffix `BKT`.

Below we show the classification of load and store bytecode instructions with information about the additional tasks they must perform. These tasks for load and store bytecodes apply to the instructions for most concepts in the language, including dynamic variables, tree attributes, list functions and map functions. Not all instruction variants are necessary. For example, there is a `IN_GET_LOCAL_WC` instruction but no `IN_GET_LOCAL_WV` instruction because changes to local variables do not need to be reverted.

Load Instructions

- `load_R` – Load a value for reading.

  Do not copy the tree. Do not generate a reverse instruction.

- `load_WC` – Load a value for writing (with commit).

  Copy the tree if shared. Do not generate a reverse instruction.

- `load_WV` – Load a value for writing (with reverting).

  Copy the tree if shared. Generate a reverse instruction.

- `load_BKT` – Load a value for reverting.

  Copy the tree if shared.

Store Instructions

- `store_WC` – Store (with commit).

  Copy the tree if shared. Do not generate a reverse instruction

- `store_WV` – Store (with reverting).

  Copy the tree if shared. Generate a reverse instruction.

- `store_BKT` – Revert a store.

  Copy the tree if shared.

In Figure 5.8 we show all variants of the attribute load and store bytecodes. The bytecodes with the `R` suffix or the `WC` suffix do not need to be reverted. The bytecodes with the `WV` suffix do, so they write to the reverse instruction log. The instructions with the `BKT` suffix are the instructions that are executed in the reverse direction.

```
case IN_GET_FIELD_R: {
    short field;
    read_half( field );

    Tree *obj = pop();
    tree_downref( program, obj );

    Tree *val = get_field( obj, field );
    tree_upref( val );
    push( val );
    break;
}
case IN_GET_FIELD_WC: {
    short field;
    read_half( field );

    Tree *obj = pop();
    tree_downref( program, obj );

    Tree *notShared = get_field_copy( program,
            obj, field );
    tree_upref( notShared );
    push( notShared );
    break;
}
case IN_GET_FIELD_WV: {
    short field;
    read_half( field );

    Tree *obj = pop();
    tree_downref( program, obj );

    Tree *notShared = get_field_copy( program,
            obj, field );
    tree_upref( notShared );
    push( notShared );

    /* Write the reverse instruction. */
    reverseCode.append( IN_GET_FIELD_BKT );
    reverseCode.appendHalf( field );
    rcodeUnitLen += 3;
    break;
}
case IN_GET_FIELD_BKT: {
    short field;
    read_half( field );

    Tree *obj = pop();
    tree_downref( program, obj );

    Tree *notShared = get_field_copy( program,
            obj, field );
    tree_upref( notShared );
    push( notShared );
    break;
}
```

```
case IN_SET_FIELD_WC: {
    short field;
    read_half( field );

    Tree *obj = pop();
    Tree *val = pop();
    tree_downref( program, obj );

    /* Downref the old value. */
    Tree *prev = get_field( obj, field );
    tree_downref( program, prev );

    set_field( program, obj, field, val );
    break;
}
case IN_SET_FIELD_WV: {
    short field;
    read_half( field );

    Tree *obj = pop();
    Tree *val = pop();
    tree_downref( program, obj );

    /* Save the old value, then set
     * the field. */
    Tree *prev = get_field( obj, field );
    set_field( program, obj, field, val );

    /* Write the reverse instruction. */
    reverseCode.append( IN_SET_FIELD_BKT );
    reverseCode.appendHalf( field );
    reverseCode.appendWord( (Word)prev );
    rcodeUnitLen += 7;
    reverseCode.append( rcodeUnitLen );
    break;
}
case IN_SET_FIELD_BKT: {
    short field;
    Tree *val;
    read_half( field );
    read_tree( val );

    Tree *obj = pop();
    tree_downref( program, obj );

    /* Downref the old value. */
    Tree *prev = get_field( obj, field );
    tree_downref( program, prev );

    set_field( program, obj, field, val );
    break;
}
```

Figure 5.8: Load and store instructions for tree attributes. Only the instructions with the `WV` suffix need to be reverted. The instructions with the `WC` suffix must copy shared trees, as do the instructions with the `WV` and `BKT` suffixes.

5.5.4 Reverse Execution Example

In Section 5.4 we noted that reduction actions are often short and simple, with similar undo actions to complement them. Sometimes they are longer and more complex, making the manual writing of undo actions difficult. This is illustrated by the `declarator_id` reduction and undo actions for C++, shown in Section 5.4. Nevertheless, the amount of reverse code that needs to be executed is still small because the global state normally contains only a few data structures. Modifications to these data structures are minor. Most of the work is actually handled by the context-free parsing system; we just need to provide it with a little assistance by supplying some additional state for handling the context-dependent portions.

In Figure 5.9 we give an example trace of the forward execution of the `declarator_id` reduction action that is shown in Figure 5.3. The instructions that log reverse bytecodes are marked with asterisks. A trace of the reverse instructions that were automatically generated is also shown. This trace was generated using the declarator id `C::f`. This type of declarator is common in the definition of functions that belong to a class.

5.5.5 A Necessary Language Restriction

Reverting changes to values that are directly named by a global variable is a matter of recording the name and the old value. Reverting changes to values that are accessed via an iterator is not as simple. Reverting such a change requires first loading the root of the iterator, then reproducing the traversal that led to the tree that was modified. We cannot simply store a reference to the tree since automatic tree copying can cause references to become invalid.

Consider the example in Figure 5.10. Suppose we need to revert the action. In the forward direction we log the modifications made by the iterators by taking references to the trees that are visited by the iterator. In the first iterator loop over `Start`, a number of references would be recorded into reverse instructions. The next statement takes a copy

Forward:

```
  IN_INIT_LOCALS 6
  IN_INIT_RHS_EL 0 -1
  IN_GET_LOCAL_R -1
  IN_GET_FIELD_R 0
  IN_GET_TOKEN_DATA_R
  IN_SET_LOCAL_WC -2
  IN_GET_LOCAL_R -1
  IN_GET_FIELD_R 0
  IN_GET_FIELD_R 1
  IN_SET_LOCAL_WC -3
  IN_LOAD_GLOBAL_R
  IN_GET_FIELD_R 1
  IN_GET_LIST_MEM_R 1
  IN_SET_LOCAL_WC -4
  IN_GET_LOCAL_R -3
  IN_JMP_FALSE 6
  IN_GET_LOCAL_R -4
  IN_GET_LOCAL_R -1
  IN_GET_FIELD_R 0
  IN_GET_FIELD_R 0
  IN_TST_EQL
  IN_JMP_FALSE 4
  IN_GET_LOCAL_R -4
  IN_PTR_DEREF_R
  IN_GET_FIELD_R 6
  IN_DUP_TOP
  IN_JMP_FALSE 18
  IN_JMP_FALSE 4
  IN_LOAD_NIL
  IN_SET_LOCAL_WC -6
  IN_GET_LOCAL_R -2
  IN_DUP_TOP
  IN_JMP_FALSE 5
  IN_GET_LOCAL_R -5
  IN_NOT
  IN_TST_LOGICAL_AND
  IN_DUP_TOP
  IN_JMP_FALSE 17
  IN_LOAD_GLOBAL_R
  IN_GET_FIELD_R 13
  IN_GET_LIST_MEM_R 1
  IN_GET_FIELD_R 1
  IN_LOAD_INT 0
  IN_TST_EQL
  IN_TST_LOGICAL_AND
  IN_JMP_FALSE 163
  IN_LOAD_GLOBAL_R
  IN_GET_FIELD_R 13
  IN_GET_LIST_MEM_R 1
  IN_GET_FIELD_R 2
  IN_JMP_FALSE 53
  IN_GET_LOCAL_R -3
  IN_NOT
  IN_JMP_FALSE 90
  IN_LOAD_GLOBAL_R
  IN_GET_FIELD_R 13
  IN_GET_LIST_MEM_R 1
  IN_GET_FIELD_R 3
  IN_JMP_FALSE 40
  IN_LOAD_GLOBAL_R
  IN_GET_FIELD_R 0
  IN_GET_LOCAL_R -2
  IN_LOAD_GLOBAL_R
  IN_GET_FIELD_R 4
  IN_GET_LIST_MEM_R 1
  IN_CALL createLangObject
  IN_INIT_LOCALS 1
  IN_GET_LOCAL_R 5
  IN_GET_LOCAL_R 4
  IN_CONSTRUCT 9
  IN_CONSTRUCT 10
  IN_GET_LOCAL_R 3
  IN_CONSTRUCT 11
  IN_SET_FIELD_LEAVE_WC 5
  IN_SET_FIELD_LEAVE_WC 2
  IN_SET_FIELD_LEAVE_WC 3
  IN_SET_FIELD_LEAVE_WC 0
  IN_SET_FIELD_LEAVE_WC 1
  IN_TREE_NEW
  IN_SET_LOCAL_WC -1
  IN_GET_LOCAL_R -1
  IN_SAVE_RET
  IN_JMP 0
  IN_RET createLangObject
  IN_SET_LOCAL_WC -6
  IN_LOAD_GLOBAL_R
  IN_GET_FIELD_R 1
  IN_GET_LIST_MEM_R 1
  IN_GET_LOCAL_R -2
  IN_GET_LOCAL_R -6
  IN_CALL insertObject
  IN_INIT_LOCALS 1
  IN_GET_LOCAL_R 4
  IN_GET_LOCAL_R 5
    IN_PTR_DEREF_R
    IN_GET_FIELD_R 3
    IN_MAP_FIND
    IN_SET_LOCAL_WC -1
    IN_GET_LOCAL_R -1
    IN_NOT
    IN_JMP_FALSE 6
    IN_CONSTRUCT 12
    IN_SET_LOCAL_WC -1
    IN_GET_LOCAL_R 3
    IN_GET_LOCAL_WC -1
    IN_LIST_APPEND_WC
    IN_POP
    IN_GET_LOCAL_R 4
    IN_GET_LOCAL_R -1
    IN_GET_LOCAL_R 5
  * IN_PTR_DEREF_WV
  * IN_GET_FIELD_WV 3
  * IN_MAP_STORE_WV
    IN_POP
    IN_LOAD_NIL
    IN_SAVE_RET
    IN_RET insertObject
    IN_POP
    IN_GET_LOCAL_R -3
    IN_LOAD_GLOBAL_R
    IN_GET_FIELD_R 4
    IN_GET_LIST_MEM_R 1
    IN_CONSTRUCT 23
    IN_SET_FIELD_LEAVE_WC 2
    IN_SET_FIELD_LEAVE_WC 0
  * IN_LOAD_GLOBAL_WV
  * IN_GET_FIELD_WV 12
  * IN_LIST_APPEND_WV
    IN_POP
    IN_GET_LOCAL_R -3
    IN_JMP_FALSE 22
    IN_POP_LOCALS 59 6
```

Reverse:

```
  IN_LOAD_GLOBAL_BKT
  IN_GET_FIELD_BKT 12
  IN_LIST_APPEND_BKT

  IN_PTR_DEREF_BKT
  IN_GET_FIELD_BKT 3
  IN_MAP_STORE_BKT
```

Figure 5.9: A forward and reverse trace of the `declarator_id` reduction action used in our C++ grammar, as run on the input `C::f` in the context of a class member function definition. The reverse instructions are few compared to the forward instructions.

```
def name
    int i
    [ident]

global Start

def start
    [name*]
    {
        Start = lhs
        for N1: name in Start   # The Start tree is modified.
            N1.i = 1

        start Copy = Start      # The value in Start becomes shared
                                # by two variables.

        for N2: name in Start   # Modification causes the Start
            N2.i = 1            # tree to be copied.
    }
```

Figure 5.10: Code that demonstrates why we must prevent iteration over globally accessible values. Doing so would violate the value semantics.

---

of the global variable and causes the value to become shared. The second iterator loop must now copy the tree because it is shared. This time the references stored in the reverse instructions will be different from those stored during the first iterator loop. During reverse execution, the undoing of the second iterator loop will work fine. However, the undoing of the first iterator loop will not affect `Start`, but instead the trees that make up the `Copy` variable.

Since we cannot store references to values during iteration, and the alternate strategy of storing tree traversal directions for reproduction during backtracking is impractical, we disallow iteration over global and dynamic variables. Iteration of globally visible data causes other problems for our tree transformation language. These problems are described in Section 4.10.4.

5.6 Context-Free Pattern and Constructor Parsing

The primary purpose of our parsing engine is to parse user input, though we also use it to parse transformation rule patterns and replacements (Sections 4.5 and 4.6). Token-generation and reduction actions are executed when user input is parsed, but should we also be executing them when we parse patterns and constructors? If we do not execute them, then we are not using the same algorithm to parse input as we are to parse patterns and constructors. This discrepancy could lead to confusion. However, if we do execute them, then there are a number of other problems to consider.

Context-dependent parsing requires an instance of global data. If we execute actions for pattern and constructor parsing then we have to decide if global data is shared among all patterns, or if a new version is instantiated for each pattern. We also need to decide if initialization routines should be executed. But what happens when the language becomes more sophisticated and it supports external libraries such as database connection libraries? Should it be required that all external dependencies function when programs are compiled? More seriously, what happens when a reduction action contains a pattern that must be parsed using the reduction action?

Rather than pursue solutions to all of these problems, we choose to support only context-free parsing of patterns and constructors. We note that context-dependent parsing is most often used for resolving grammar ambiguities, and therefore we feel it is sufficient to perform only context-free parsing on patterns and constructors. The user can resolve ambiguities in these situations by refining patterns with more information. For example, in the code that follows, only literal C++ program text is given in the pattern. Context-dependent parsing features would be required at compile-time to look up the type of `C` to determine that it is a class name.

```
match statement
    "C v(1);"
```

Instead of relying on context-dependent parsing features to resolve the identifier `C` to a class name, the `class_name` type can be used directly in the pattern and afterwards the text that it matched can be tested against a literal string to enforce the correct name.

```
match statement
    [ClassName: class_name " v(1)"]
&& ClassName == 'C'
```

5.7 Summary

In this chapter we complete the parsing method of our source transformation system by making parse-time modifications to global data structures compatible with generalized parsing. This enables generalized context-dependent parsing.

In Sections 5.1 and 5.2 we describe semantic actions and token-generation actions that can be used to maintain global information and programmatically generate tokens, possibly with the assistance of global data. Global data can also be used to influence the parse with semantic predicates, as we describe in Section 5.3.

In Section 5.4 we add undo actions. These actions are a class of semantic actions that are executed by the parse engine as it undoes parsing. A forward semantic action is executed when a sequence of nonterminals and terminals on the top of the stack is replaced with a nonterminal that derives the sequence (a reduction). An undo action is executed when the engine replaces a nonterminal with the sequence of items it previously reduced from. If the undo action reverts any changes to the global state that were made by the forward action, then context-dependent parsing can be made generalized.

Asking the user to write undo actions is possible. However, writing these actions is somewhat tedious and it requires that a correct undo action be derived from the forward action. This is a situation that is amenable to automation. In Section 5.5 we make use of reverse execution to free the user from having to write undo actions and provide assurance that the global state cannot become corrupt due to mismanagement of global data.

In the next chapter we demonstrate the parsing capabilities of our system. We provide example solutions to a selection of parsing problems taken from data languages, network protocols and programming languages.

Chapter 6

Examples

Our transformation system can be used to parse and transform computer languages that normally cause grammar-based parsing systems trouble. Since our parsing algorithm is generalized we don't have to contend with parsing algorithm restrictions, as is the case with LALR(1) [28] and LL(1) [62] systems. We can use grammars found in language definitions as they are given and perform stepwise refinement on them while continually generating and testing the parser. Since our system supports the modification of global state during parsing we can use established techniques for supplementing the grammar with any data structures that are necessary to parse context-dependent language features. Since our lexical analysis system is grammar-dependent we can parse languages with lexical rules that vary by grammar context. Since both the lexical and context-dependent features work in combination with the generalized parser we are not restricted in our ability to evolve or combine grammars.

We demonstrate the capabilities of our system by solving parsing problems that exist in common computer languages. We do not present entire grammars, but instead focus on challenging characteristics of these languages and show how to parse the language fragments using our system. We sample problems from the communication protocols HTTP [31] and DNS [70], the document markup language HTML [81], and the programming languages

Ruby [33], Python [80] and C++ [44]. In the course of this we will demonstrate the lexical analysis capabilities of our system, the use of ordered choice, and the context-dependent capabilities of our system.

The lexical features of our system will be demonstrated by showing how to parse languages that do not have uniform lexical rules, how to parse nested comments, how to handle grammar-dependent newline interpretation and how to parse binary protocols. We will demonstrate ordered choice by using our system to resolve ambiguities in language definition grammars. We will demonstrate the context-dependent parsing abilities of our system by showing how to parse markup languages (including broken HTML markup), how to parse *here* documents, how to implement off-side rules, how to do symbol table maintenance, how to translate token types according to a symbol table, and how to handle unusual parsing requirements such as the partially delayed parsing of C++ class method bodies.

6.1 HTTP

There are two characteristics of HTTP that we give solutions for. At the top level of the language there are no contiguous streams of tokens. There is a request line, a list of headers and the content of the request. We present a small grammar that parses a request into the request line components and the headers. Though it is not very useful on its own, this grammar demonstrates how to define languages without contiguous token streams. Each token in the grammar is defined in a lexical region of its own.

The second parsing problem that we explore is encountered when we drill down further into the language. To extract additional details of a request the header fields must be parsed. Some of these fields allow nested comments. We present a solution to the problem of nested comments. Though this problem is not exclusive to HTTP, we take this opportunity to demonstrate it.

6.1.1 Varying Lexical Rules

At a high level, HTTP is a simple protocol. It consists of a request method (for example `GET`, or `POST`), the URI of the request, a protocol version string, a sequence of headers, then the free-form content of the request. The headers are simply name-value pairs, with the name and value separated by a colon. Most header field content is on a single line, however field content may be extended onto multiple lines by preceding each extension line with whitespace. HTTP is different from programming languages in that it is not composed of a sequence of tokens taken from the same pool of lexical definitions. The request line components, the header names and the header fields each require different lexical rules.

We demonstrate rudimentary HTTP request parsing for the purpose of showing how languages that do not have a uniform token stream can be parsed. We simply break up a request into the request line components and the list of headers. Inside headers there may be more complex syntax and semantics; however, we do not consider header field content in this example.

In the HTTP grammar in Figure 6.1 no token is specified in an explicitly defined lexical region, and therefore each token is assumed to be defined in a lexical region of its own (Section 4.4.1). This grammar also makes use of regular language definitions for defining character classes. These definitions may be used in token definitions and result in a tidier specification.

6.1.2 Nested Comments

In the HTTP grammar of the previous section we do not delve into the syntax of header field content. When we explore the syntax of the headers we find that some HTTP headers permit comments that are delimited by the single parentheses symbols `(` and `)`. These comments can be nested.

The HTTP definition includes comments in the grammar and does not stipulate that

```
#
# Character classes
#
rl CTL /0..31 | 127/
rl CR /13/
rl LF /10/
rl SP /32/
rl HT /9/
rl CHAR /0..127/

rl separators / '(' | ')' | '<' | '>'
        | '@' | ',' | ';' | ':' | '\\'
        | '"' | '/' | '[' | ']' | '?'
        | '=' | '{' | '}' | SP | HT /

rl token_char /CHAR - CTL - separators/

#
# Literal tokens
#
literal 'HTTP/', ' ', ':'
token CRLF /CR LF/

#
# Request Line
#
token method /token_char+/

token request_uri /(^SP)+/

token http_number /digit+ '.' digit+/

def http_version
    [ 'HTTP/' http_number ]

def request_line
    [method ' ' request_uri
     ' ' http_version CRLF]

#
# Header
#
token field_name /token_char+/

token field_value
    /(^(CR|LF) | CR LF (SP|HT))* CR LF/

def header
    [field_name ':' field_value]

#
# Request
#
def request
    [request_line header* CRLF]
```

Figure 6.1: A rudimentary grammar for the high-level components of HTTP. Each token is defined in a lexical region of its own. Because of this, the parser attempts to scan each token using an independent scanner invocation.

---

they be removed during the lexical phase. This can be handled using most parsing systems by defining the comment as a nonterminal and referencing it in the grammar where needed. This is fairly easy to do. However, to demonstrate the abilities of our parsing system we parse a superset of the specification and allow nested comments anywhere by removing comments during the lexical phase. This represents a challenge for many other systems. The syntax of nested comments cannot be captured using regular expressions; therefore most parsing systems require the user to augment the scanner with manually written code

```
# Primary lexical region.
lex start
{
    ignore /[ \t\n\r\v]+/
    token id /[a-zA-Z_][a-zA-Z0-9_]*/

    token open_paren /'('/
    {
        send_ignore( parse_stop nested_comment( stdin ) )
    }
}

# Lexical region for nested comments.
lex nc_scan
{
    literal '(', ')'
    token nc_data /[^()]+/
}

def nc_item
    [nc_data]
|   [nested_comment]

def nested_comment
    ['(' nc_item* ')']
```

Figure 6.2: Parsing nested comments by invoking a context-free parse from a token-generation action. The nested comment grammar uses a lexical region of its own.

---

to parse nested comments.

In the example in Figure 6.2 a token pattern recognizes the opening parenthesis of a comment. A token-generation action associated with this pattern is then used to invoke context-free parsing of the comment. The returned tree is sent to the parser as an ignored tree so that it can be reproduced during unparsing.

6.2 DNS Protocol

The domain name system (DNS) is the name resolution service used on most computer networks, including the Internet. The DNS protocol specifies how messages are passed

between computers that need to look up names in the DNS. Generic transformation systems are not normally used for parsing and rewriting protocols such as DNS. Normally, *ad hoc* techniques are used, or parser generators that are designed for binary network protocols [66, 32, 76].

The DNS protocol makes heavy use of blocks of data with a pre-specified cardinality, or length in bytes. These length restrictions are the reason other transformation systems are not capable of parsing DNS. The length fields do not cause a problem for us because of the context-dependent parsing abilities of our system. We employ a grammar that uses mostly one token type and controls parsing by enforcement of semantic predicates.

6.2.1 Parsing with One Token Type

In our DNS grammar we use mostly just a single token type, `octet`. There are a few additional tokens that we have defined for convenience, but they are not necessary. It is possible to use only one token type and enforce all syntax and semantics using a grammar augmented with context-dependent predicates. In Figure 6.3 we show a section of such a grammar. In the case of the `rr_type`, `rr_class`, and `rdlength` definitions, the two-octet values that are accepted are converted to integer identifiers and lengths. These are referenced elsewhere in the grammar, in the enforcement of semantic predicates.

6.2.2 Length and Cardinality

Many low-level communication protocols make use of length and cardinality fields for specifying the number of bytes or the number of units of something in a message. In this situation the length or cardinality is the only clue about when to stop parsing a list of items. Without counting the items the input is ambiguous. This is a major problem for existing transformation systems because they do not provide the tools to deal with the context-dependent nature of these restrictions.

Requiring a particular list length can be achieved by counting items during parsing and

```
# Used for most of the grammar.
token octet /any/

def resource_record
    [name rr_type rr_class ttl rdlength rdata]

def rr_type
    [octet octet]
    {
        rr_type_value = network_uord16( r1, r2 )
    }

def rr_class
    int value
    [octet octet]
    {
        rr_class_value = network_uord16( r1, r2 )
    }

def ttl
    [octet octet octet octet]

def rdlength
    [octet octet]
    {
        rdata_length = network_uord16( r1, r2 )
    }
```

Figure 6.3: A section of our DNS grammar. We use mostly a single token type and control parsing with semantic predicates. In many cases these are length and cardinality restrictions, but we also need to decode identifiers that indicate payload types.

---

rejecting parses in which the length restriction is violated. There are several approaches to this. We show a left-recursive approach and a right-recursive approach. They both make use of a lower bound and an upper bound to squeeze out the correct list length. The left-recursive version is shown in Figure 6.4. The global variable `target` stores the number of items to accept and it is assumed that this variable is set somewhere ahead of the list. The current number of items is passed up the tree as the list is parsed.

If the target number were immediately ahead of the list the target could be parsed at the root of the list and passed up the tree as well, eliminating the need for a global variable.

```
global int target

def count_items
    int count

    # Left-recursive list production gives the list an upper bound.
    [count_items item]
    {
        lhs.count = r1.count + 1
        if ( lhs.count > target )
            reject
    }

    # List base case.
|   []
    {
        lhs.count = 0
    }

# A wrapper around the list gives it a lower bound.
def counted_list
    [count_items]
    {
        if ( r1.count < target )
            reject
    }
```

Figure 6.4: An example of left-recursive counting, with the target length previously stored in a global variable and the current list length passed up the tree. Two conditions are used to squeeze out the correct length.

This would allow a counted list to appear nested inside a counted list of the same type. If the number of items were not immediately ahead and nesting was required then using a stack of target lengths would suffice.

Items can also be counted using right-recursive lists. In this form the counting tools can be made generic. In Figure 6.5 two empty definitions `count_inc` and `count_end` enforce the semantic predicates. The `count_inc` definition is placed at the head of a recursive list production and prevents a list from getting too long. The `count_end` definition is placed in

```
global int target
global int count

def count_items
    [count_inc item count_items]
|   [count_end]

def count_inc
    []
    {
        if ( count < target )
            count = count + 1
        else
            reject
    }

def count_end
    []
    {
        if ( count < target )
            reject
    }
```

Figure 6.5: Generic right-recursive counting definitions. The `count_inc` definition is placed at the head of the recursive production of the list, and the `count_end` definition is placed in the terminator production.

---

the terminator production of a list and prevents a short list from being accepted. Unfortunately, globals must be used for both the target and the counter variables in this case. If these generic semantic predicates must be used to make a nested list then both global integer values can be replaced with stacks of integers values.

6.3 HTML

As a member of the family of markup languages that includes SGML and XML, HTML requires a parser that is capable of matching opening and closing markup tags. This is a problem for transformation systems that employ purely context-free parsing systems, however we are able to give a solution for tag matching. Further difficulty is caused by

the fact that it is very common for HTML pages to be invalid. Our system is also able to handle broken HTML.

Another interesting property of markup languages is that they do not have uniform token streams. The document data that is marked up can have unique lexical properties and the scanning of the document data must not be dictated by the lexical rules of the tags. This is a problem we can handle by using multiple lexical regions.

6.3.1 Matching Tags

Tags must be matched in order to parse markup languages. HTML presents an additional challenge because over the years the developers of popular web browsers have crafted very forgiving implementations. The result of this is that quality assurance testing of websites that uses only standard web browsers will fail to reveal many errors. Validation must be checked using a parser other than the browser's, a step which appears to not be carried out very often.

Since broken HTML is pervasive, it is very difficult to make a grammar-based HTML parser and apply it to HTML found in a random sampling of the Web. Undoubtedly, a very robust parser is required. Robust parsing can be implemented using ambiguous island-style grammars and existing systems are capable of this. However, when using the existing tools, island parsing conflicts with the context-dependent requirement of matching HTML tags. HTML does not pose a problem for our system since we are able to craft a robust parser that takes advantage of what generalized parsing has to offer, yet we can also implement tag matching.

In Figure 6.6 we demonstrate the first step of tag matching in our HTML grammar. Closing tag identifiers are translated into one of two types. A `close_id` indicates that the closing tag id matches some tag in the stack of currently open tags. A `stray_close_id` indicates that the closing tag id does not match anything in the stack of currently open tags.

```
# This lexical region is only for the id in close tags. The id needs to be
# looked up in the tag stack so we can determine if it is a stray.
lex close_id
{
    # Ignore whitespace.
    ignore /space+/

    token stray_close_id //
    token close_id /def_name/
    {
        # If it is in the tag stack then it is a close_id. If not then it is a
        # stray_close_id.
        int send_id = typeid stray_close_id

        tag_stack LocalTagStack = TagStack
        for Tag: tag_id in LocalTagStack {
            tag_id T = Tag
            if ( match_text == T.data ) {
                send_id = typeid close_id
                break
            }
        }

        send( make_token( send_id, pull(stdin, match_length) ) )
    }
}
```

Figure 6.6: The lexical region for handling close tag identifiers. We translate the id into either a `close_id` that matches a tag in the tag stack, or a `stray_close_id`.

Once closing tag ids are translated into either a legitimate closing tag id or a stray, we are able to write the grammar, which we give in Figure 6.7. We define four types of tags: the regular tag that has a matching close id, the unclosed tag that has the nonterminal `opt_close_tag` parsed as the empty string, the explicitly empty tag, and the stray close tag. The `item` definition alternates between these tags and we can rely on the generalized parser to sort out the details. We just need to make sure that when we accept a regular tag, that the `close_id` matches the tag on the top of the stack and that we properly maintain the tag stack by pushing the open tags and popping the close tags.

```
def tag
    [open_tag item* opt_close_tag]

def open_tag
    ['<' tag_id attr* '>']
    {
        TagStack = construct tag_stack
            [r2 TagStack]
    }

def opt_close_tag
    ['</' close_id '>']
    {
        match TagStack
            [Top:tag_id Rest:tag_stack]
        if ( r2.data == Top.data )
            TagStack = Rest
        else
            reject
    }
|   []
    {
        match TagStack
            [Top:tag_id Rest:tag_stack]
        TagStack = Rest
    }
```

```
def empty_tag
    ['<' tag_id attr* '/>']

def stray_close
    ['</' stray_close_id '>']

def item
    [DOCTYPE]
|   [tag]
|   [empty_tag]
|   [stray_close]
|   [doc_data]
|   [comment]

def start
    [item*]
```

Figure 6.7: A robust HTML grammar capable of handling unclosed open tags and stray close tags. This grammar relies on the translation of close tag identifiers in Figure 6.6.

6.3.2 Transforming Unclosed Tags

By designing our HTML grammar to use an optional close tag we make the assumption that tags are normally closed, and an unclosed tag is the exception. Tags that follow an open tag are contained within the open tag. When it is discovered that a close tag is missing, rather than backtrack, the parser simply skips the close. Therefore tags that follow an unclosed tag become contained in it. After a parse we may wish to add in missing close tags. The transformation in Figure 6.8 does just that, thereby fixing broken HTML.

On the other hand, for the purpose of transformation it may be desirable to treat tags

```
# Finds unclosed tags that should be closed and adds the missing close tag.
int close( ref start Start )
{
    for TL: item in Start {
        require TL
            [OpenTag: open_tag Inside: item*]

        match OpenTag
            ['<' TagId: tag_id attr* '>']

        if ( should_close( TagId ) )
        {
            close_id CloseId = construct close_id
                [TagId.data]

            opt_close_tag CloseTag =
                construct opt_close_tag ['</' CloseId '>']

            # Close the tag.
            TL = construct item
                [OpenTag Inside CloseTag]
        }
    }
}
```

Figure 6.8: A transformation that closes unclosed tags. The `should_close` function determines which tags should be transformed.

---

with missing close tags as standalone items, rather than items that capture the items that follow. We can employ a transformation that removes data from inside an unclosed tag and places it after the tag. This transformation is shown in Figure 6.9.

6.4 Ruby

The primary implementation of the Ruby language uses a handwritten lexical analyzer coupled with a Yacc grammar. Given the context-dependent parsing and scanning trickery that easily creeps into a Yacc-based parser with a handwritten lexer, it is no surprise that

```
# Finds unclosed tags that should be flattened and puts the content after the
# tag. Afterwards all flattened tags will be empty inside.
int flatten( ref start Start )
{
    for TL: item* in Start {
        require TL
            [OT: open_tag Inside: item* Trailing: item*]

        match OT
            ['<' TagId: tag_id attr* '>']

        if ( should_flatten( TagId ) ) {
            require Inside
                [item item*]

            # Put Trailing at the end of Inside.
            for END: item* in Inside {
                if match END [] {
                    END = Trailing
                    break
                }
            }

            opt_close_tag EmptyCloseTag =
                construct opt_close_tag []

            # Close the tag and put Inside after it.
            TL = construct item*
                [OT EmptyCloseTag Inside]
        }
    }
}
```

Figure 6.9: A transformation that moves content from inside an unclosed tag to after the tag. This kind of transformation is useful for fixing the parse of empty tags such as `<br>`.

---

transformation systems stumble on the Ruby language. In fact, all known Ruby implementations use a derivative of the original handwritten lexer and the Yacc grammar, and build the parser with some Yacc-like parser generator.

There are two difficulties associated with parsing Ruby for which we present solutions in this section. The first is the interpretation of newlines. In some contexts newlines should be ignored, in others they should be treated as significant. The second problem we give a

solution for is *here* documents.

6.4.1 Grammar-Dependent Newline Interpretation

The Ruby language makes use of the semi-colon for terminating statements, but makes the semi-colon optional. Instead, a newline can be used to terminate a statement. This requires that newlines be significant and not ignored by the scanner. However, Ruby also allows lines to be continued when it seems like the reasonable thing to do. If the last lexical token on a line is an operator then the line that follows it continues the expression. Therefore newlines need to be ignored some of the time. In the following code the first newline must be treated as significant because it terminates the assignment statement.

```
a = b
c
```

In this next example the newline must be ignored because it follows a binary operator. Other examples of operators that require following newlines to be ignored include comma operators in argument lists and access operators such as `.` and `::`.

```
a = b +
    c
```

Another interesting aspect of the Ruby language is that a large part of the language can appear at the leaf nodes of an expression. The `primary` nonterminal contains the function definition, module definition, if statement, while statement, generic code block, and many others. These constructs are normally considered top-level or statement-level definitions in other languages.

```
a = [
    def f(a)
        return a
    end,
    f(1)
]
```

While these features are interesting, they complicate the matter of lexical analysis, in particular what to do when a newline is seen. The question of ignoring it or making it significant depends on the state of the parser. The Ruby interpretor handles this situation by maintaining state variables in the semantic actions of the parser. These state changes occur throughout the grammar and are difficult to understand. The scanner queries these variables to decide if a newline should be sent to the parser to be accepted as part of the language, or if it should be ignored.

We can solve this problem by using lexical regions instead of explicitly maintaining state. In the example in Figure 6.10 the primary region contains the tokens that may begin an expression. It ignores preceding newlines and therefore the tokens in it can be used to continue an expression onto another line. A secondary region named `expr_cont_ops` contains the binary operators. In this region, preceding newlines are not accepted and therefore tokens in it cannot be used to continue an expression. The `term` region contains the significant newline that terminates an expression. We then rely on our system to automatically select the appropriate lexical region according to the current parser state.

Optional Argument Parenthesis

In the previous section we show the lexical regions that we use for Ruby. In some grammatical contexts the newline is ignored, such as in front of whole expressions. In other contexts it is significant and allowed, such as after whole expressions. These rules allow an expression to continue to the next line after some operator, but before the next whole expression. They also allow newline to terminate a statement when no operator is present at the end of a line.

Unfortunately, there is also a grammatical context in which newline is significant, but *not* allowed. Ruby allows optional parentheses around function arguments and parameters. In the following example the expression `print a` is a call to the `print` function with one argument, `a`. However, when a newline separates the `print` and `a` words then this is two

```
# The items in this scanner may have an insignificant newline in front.
lex start
{
    # Reserved Words.
    literal '__LINE__', '__FILE__', '__ENCODING__', 'BEGIN', 'END', 'alias',
        'and', 'begin', 'break', 'case', 'class', 'def', 'defined?', 'do',
        'else', 'elsif', 'end', 'ensure', 'false', 'for', 'in', 'module',
        'next', 'nil', 'not', 'or', 'redo', 'rescue', 'retry', 'return',
        'self', 'super', 'then', 'true', 'undef', 'when', 'yield', 'if',
        'unless', 'while', 'until'

    token tNTH_REF /'$' [0-9]+/
    token tBACK_REF /'$' ( '&' | '`' | '\'' | '+' ) /
    token tUPLUS /'+'/
    token tUMINUS /'-'/
    literal ')', ',', ']', '{', '}', ':', '.', '::', '->', '!', '~'

    # These text of these items also appear as expression operators. In this lexical
    # region whitespace is insignificant, but as an operator it is significant. We let
    # the backtracker sort it out.
    token tLBRACK /'['/                         token tGVAR /'$' [a-zA-Z_]+/
    token tLPAREN /'('/                         token tIVAR /'@' [a-zA-Z_]+/
    token tSTAR /'*'/                           token tCVAR /'@@' [a-zA-Z_]+/
    token tBAR /'|'/
    token tAMPER /'&'/                          token tINTEGER /[0-9]+/
                                                token tFLOAT /[0-9]+ '.' [0-9]+/
    token tIDENTIFIER /[a-z][a-zA-Z_]*/
    token tFID /[a-z][a-zA-Z_]* ('!'|'?')/      token tDSTRING_BEG /'"'/
    token tCONSTANT /[A-Z][a-zA-Z_]*/           token tSSTRING_BEG /'\''/
                                                token tXSTRING_BEG /'`'/
    ignore /[ \t\n]+/
    ignore comment /'#' [^\n]* '\n'/
}

# This region does not consume preceding newlines as whitespace and
# therefore the items in it cannot appear at the beginning of a line. Newlines
# are not accepted at all in this lexical region.
lex expr_cont_ops
{
    ignore /[\t ]+/

    literal '+', '-', '*', '**', '/', '%', '^', '|', '&', '||', '&&', '[', '(',
        '=', '<<', '>>', '?', '<=>', '=>', '[]', '[]=', '=~', '!~', '<', '>',
        '>=', '<=', '!=', '==', '===', '..', '...'
}

# The terms region contains semi-colon and treats newline as a significant token.
lex terms
{
    ignore /[\t ]+/
    ignore /'#' [^\n]*/
    literal ';'
    literal '\n'
}
```

Figure 6.10: Using lexical regions to selectively ignore newlines. Tokens that begin an expression may have insignificant newlines in front of them. Binary operators and parentheses that open arrays and function calls cannot have newlines in front of them.

```
# To the right of ':' is required trailing lexical context that is
# not included in the pattern match.
token ws_no_nl /[ \t]+ : [^ \t\n]/

def method_call
    [operation '(' call_args ')']
|   [operation ws_no_nl call_args]
```

Figure 6.11: Parsing Ruby requires that we make exceptions to the grammar-dependent newline rules and forbid newlines in particular places where they would otherwise be allowed. Here we prevent newlines from occurring in front of method call arguments that do not have enclosing parentheses.

---

function calls, `print` and `a`.

```
def f a
    print a
    print
    a
end
```

In the lexical context ahead of `a` we are expecting an expression and by the above rules this dictates that newlines are ignored. But if we ignore newlines, then two words that are each on their own line will be parsed as a function call with a single argument that is not wrapped in parentheses, which is wrong. Instead, we need to maintain the rules from the previous section, but we need to introduce a special exception for this case.

This could be solved by duplicating the `start` lexical region, making newline significant in this region, referencing the terminals from this duplicate region in a duplicate of the expression nonterminal, then using this special-purpose expression as the first argument of a function call. Unfortunately, this would entail large amounts of grammar duplication. A simpler solution is much more desirable.

A better solution, shown in Figure 6.11, is to add a whitespace-consuming terminal in between the function call and the argument list. This terminal comes from a lexical region of its own and forcefully consumes the whitespace between the function name and the argument, but in the process disallows newlines. In this way the default lexical rule

of ignoring newlines is overridden for a single grammar form only. If the `ws_no_nl` token fails to match because of a newline the backtracker is invoked and the parser moves on to a different possibility.

6.4.2 Here Documents

The *here* document is a feature of many scripting and high-level languages including Ruby, Bash, Perl, and PHP. A here document allows a programmer to specify a block of text that is to be interpreted as a literal string. At the beginning of the here document the user gives an identifier that marks the end of the here document when it appears at the beginning of a line. The string `EOF` is often used by convention, but it is important to allow an arbitrary word in order to not restrict the text of the here document.

Here documents can be handled easily using handwritten lexical analyzers. However, generalized transformation systems have trouble recognizing a user-defined identifier. Another problem with parsing the here document is that the text of the document normally starts on the line following the opening here document identifier. Any text on the same line as the opening identifier must be parsed normally. It is not possible to begin collecting text immediately following the opening of the here document. Consider the following Ruby code. Two here documents open on the same line.

```
print( <<DATA1, more, <<DATA2, 99 )
"&^#(@ almost
!arbitrary text!
DATA1
hello
world
DATA2
```

The here document text begins at the first newline that follows the appearance of a here document identifier. One strategy for handling this is to check if here document text must be consumed immediately after a newline is parsed, with the consumption dependent on a global variable that contains a queue of here document identifiers. This could get difficult

because newlines can appear in multiple places in the grammar: significant statement terminators, insignificant whitespace, strings, and comments. The here document data could get associated with any of these.

Another strategy is to manipulate the input stream during parsing. When a here document identifier is seen, the text up to the end of the current line is removed from the input stream and stored. The here document data is then parsed. When the parsing of the here document data is complete the rest of the line that contained the identifier is put back to the input stream and parsing can resume. The advantage of this technique is that the here document data can be associated with the here document identifier. The example in Figure 6.12 implements this technique.

The trace in Figure 6.13 shows the sequence of instructions that are executed when the grammar in Figure 6.12 is used to parse a single here document that contains two lines of content, then the closing id. Comments mark the major steps in the process. The reverse of this execution is included.

6.5 Python

There are two Python parsing problems for which we show a solution. A well-publicised characteristic of Python is that it contains off-side rules as an alternative to block delimiters such as `begin` and `end`. The other characteristic that we focus on is an ambiguity in the language definition surrounding access to collections such as dictionaries and arrays.

6.5.1 Off-Side Rules

Off-side rules are used by some programming languages for the purpose of grouping code into blocks without using explicit block delimiters. The technique involves analyzing the indentation levels of code and determining the block structure from relative indentation. Since programmers often represent block structure with indentation to make programs

```
global str HereId
token rest_of_line /[^\n]*'\n'/

lex here_start
{
    ignore /[ \t\n]+/
    token here_id
        here_data HereData
        /ident_pattern/
        {
            # Take the text of the here_id from the input stream.
            HereId = pull( stdin, match_length )

            # Get the data up to the rest of the line.
            rest_of_line ROL = parse_stop rest_of_line( stdin )

            # Parse the here document data.
            here_data HereData = parse_stop here_data( stdin )

            # Push the rest-of-line data back to the input stream.
            push( stdin, ROL )

            # Send here_id and attach the here document data.
            send( make_token( typeid here_id, HereId, HereData ) )
        }
}

lex here_data
{
    token here_close_id
        / ident_pattern '\n' /
        {
            if ( match_text == HereId + '\n' ) {
                send( make_token(
                    typeid here_close_id,
                    pull(stdin, match_length) ) )
            }
            else
                send( make_token( typeid here_line, pull(stdin, match_length) ) )
        }

    token here_line
        / [^\n]* '\n' /
}

def here_data [here_line* here_close_id]
def heredoc ['<<' here_id]
```

Figure 6.12: Here document parsing. Since the here document data starts only on the line following the opening identifier, we pull the remaining data from the line, parse the here document, then push the remainder of the line back to the input stream to allow it to be parsed naturally.

Forward:

```
token: here_id
translating: here_id
IN_INIT_LOCALS 2
IN_LOAD_GLOBAL_R
IN_GET_STDIN
IN_GET_MATCH_LENGTH_R
IN_STREAM_PULL
IN_LOAD_GLOBAL_WV
IN_SET_FIELD_WV 0
IN_LOAD_GLOBAL_R
IN_GET_STDIN
IN_PARSE 0 14    # Parse rest of line
    token: rest_of_line
    shifted: rest_of_line
    stopping the parse
    new token region: ___ROOT_REGION
    scanner has been stopped
IN_SET_LOCAL_WC -1
IN_LOAD_GLOBAL_R
IN_GET_STDIN
IN_PARSE 1 38    # Parse here_data
    token: here_line
    shifted: here_line
    new token region: <here_data>
    token: here_line
    shifted: here_line
    new token region: <here_data>
    token: here_close_id
    translating: here_close_id
    IN_INIT_LOCALS 0
    IN_GET_MATCH_TEXT_R
    IN_LOAD_GLOBAL_R
    IN_GET_FIELD_R 0
    IN_LOAD_STR 0
    IN_CONCAT_STR
    IN_TST_EQL
    IN_JMP_FALSE 16
    IN_LOAD_INT 17
    IN_LOAD_GLOBAL_R
    IN_GET_STDIN
    IN_GET_MATCH_LENGTH_R
    IN_STREAM_PULL
    IN_MAKE_TOKEN 2
    IN_SEND
    IN_POP
    IN_JMP 13
    IN_POP_LOCALS 1 0
    IN_STOP
    sending queue item: here_close_id
    reduced: _repeat_here_line-2
    shifted: _repeat_here_line
    reduced: _repeat_here_line-1
    shifted: _repeat_here_line
    reduced: _repeat_here_line-1
    shifted: _repeat_here_line
    shifted: here_close_id
    reduced: here_data-1 rhsLen: 2
    shifted: here_data
    stopping the parse
    new token region: ___ROOT_REGION
    scanner has been stopped
IN_SET_LOCAL_WC -2
IN_LOAD_GLOBAL_R
IN_GET_STDIN
IN_GET_LOCAL_R -1
IN_STREAM_PUSH   # Push rest-of-line back.
IN_POP
IN_LOAD_INT 16
IN_LOAD_GLOBAL_R
IN_GET_FIELD_R 0
IN_GET_LOCAL_R -2
IN_MAKE_TOKEN 3  # Make and send here_id.
IN_SEND
IN_POP
IN_POP_LOCALS 0 2
IN_STOP
sending queue item: here_id
```

Reverse:

```
sending back: here_id (artificial)
IN_STREAM_PUSH_BKT
IN_PARSE_BKT 1
    hit error, backtracking
    unparsed non-term: here_data
    unparsed effective term: here_close_id
    unparsed non-term: _repeat_here_line
    unparsed non-term: _repeat_here_line
    unparsed non-term: _repeat_here_line
    sending back: here_close_id (artificial)
    IN_STREAM_PULL_BKT
    push back of 6 characters
    pushing back runbuf
    IN_STOP
    unparsed effective term: here_line
    sending back: here_line (artificial)
    IN_STREAM_PULL_BKT
    push back of 6 characters
    IN_STOP
    unparsed effective term: here_line
    sending back: here_line (artificial)
IN_STREAM_PULL_BKT
push back of 6 characters
IN_STOP
IN_PARSE_BKT 0
    hit error, backtracking
    unparsed effective term: rest_of_line
    sending back: rest_of_line
    push back of 7 characters
IN_LOAD_GLOBAL_BKT
IN_SET_FIELD_BKT 0
IN_STREAM_PULL_BKT
push back of 5 characters
IN_STOP
```

Figure 6.13: An execution trace of here document parsing in the forward and reverse directions. The here document contains two lines of content and then the closing identifier.

```
simple_stmt    → expression_stmt

compound_stmt  → for_stmt

for_stmt       → 'if' expression ':' suite
                 ( 'elif' expression ':' suite )*
                 ( 'else' ':' suite )?

stmt_list      → simple_stmt ( ';' simple_stmt )* ';'?

suite          → stmt_list NEWLINE
                | NEWLINE INDENT statement* DEDENT

statement      → stmt_list NEWLINE
                | compound_stmt
```

Figure 6.14: A section of the Python grammar that shows how generated `INDENT` and `DEDENT` tokens are used.

---

readable, the block structure can be taken from the layout. Languages other than Python that support off-side rules include ABC, Miranda and Haskell.

Off-side rules pose a problem for parsing systems that do not support the generation of tokens using handwritten code. To parse languages with off-side rules the tokenizer must look at the amount of whitespace at the beginning of each line and compare it to the amount of whitespace at the beginning of the previous line. If there is less whitespace it generates some number of `DEDENT` tokens, indicating that the current indentation level has dropped down. If there is more it generates an `INDENT` token, indicating an increase in the indentation level. Doing this is straightforward when it is possible to add arbitrary code to a scanner, as is the case with a Lex-based scanner, but it is a problem when working in scanner and parser environments that employ a fixed token stream, or when there is no opportunity to programmatically generate tokens.

In Figure 6.14 a section of the Python grammar concerning statements is given. It shows how the generated `INDENT` and `DEDENT` tokens are used in the grammar in place of explicit delimiters. The `suite` type is the only grammar definition that uses `INDENT`s and `DEDENT`s.

```
token INDENTATION
    /'\n' [ \t]*/
    {
        # First send the newline.
        send( make_token( typeid NEWLINE, '' ) )

        # Compute the indentation level. Note: a more sophisticated
        # implementation would account for TABs.
        int data_length = match_length - 1

        if ( data_length > IndentStack.top ) {
            # The indentation level is more than the level on the top of
            # the stack. Send an INDENT token and record the new indentation
            # level.
            send( make_token( typeid INDENT, '' ) )
            IndentStack.push( data_length )
        }
        else {
            while ( data_length < IndentStack.top ) {
                # The indentation level is less than the level on the top of
                # the stack. Pop the level and send a DEDENT token.
                IndentStack.pop()
                send( make_token( typeid DEDENT, '' ) )
            }

            # Make sure the DEDENT lines up with an existing indentation level.
            if ( data_length != IndentStack.top )
                raise_error( 'invalid DEDENT' )
        }

        # We have now finished resolving the block structure implied by the
        # indentation. Send a whitespace token containing the text of the match.
        send_ignore( make_token( typeid WS, pull(stdin, match_length) ) )
    }
```

Figure 6.15: An implementation of Python off-side rules. The token pattern matches a newline and all following whitespace. The associated token-generation action compares the amount of whitespace to the previous indentation level, sending `DEDENT` tokens if there is less or an `INDENT` if there is more.

In Figure 6.15 we give an implementation of off-side rules. We define a token pattern that matches newlines and all following whitespace. A token-generation action then computes the current indentation level and compares it to the previous indentation level. From this

comparison it decides if it should send some number of `DEDENT` tokens or an `INDENT` token. In the process it updates the global variable that tracks the currently open indentation levels so that the indentation at the beginning of the next line can be properly analyzed.

This token-generation action may send more than one token to the parser. Since we have resigned ourselves to reverting only entire token-generation and reduction actions to avoid reverting changes to local variables, the system must queue up tokens and send them following the execution of the generation action. This allows the reverse code for the reduction action to exist as a single sequence of instructions. The entire block of reverse code is associated with the first token sent by the generation action. For example, consider the following block of Python code.

```
def fun():
    print "fun"
```

The token-generation action for the `INDENTATION` pattern is executed when the newline and space characters following the colon are matched. The indentation stack begins with zero on top, therefore an indentation level of four causes an `INDENT` token to be sent. Following that a whitespace token consumes the newline and four spaces, ensuring that the indentation pattern does not repeatedly match. A trace of the forward and reverse code is shown in Figure 6.16.

6.5.2 Resolving Ambiguities

There is an ambiguity in the Python grammar, surrounding access to ordered collections. This is detailed in the Python Reference Manual version 2.5, Section 5.3.3. The ambiguity is between subscriptions and different forms of slicings. Subscriptions are used to take an object out of a collection of objects, returning a single object. A slicing is used to select a range of objects out of a sequence, returning a sequence of objects. The subscription and slicing definitions both use the square bracket symbols as delimiters on the selection expression. The Python grammar fragment in Figure 6.17 details slicings and subscriptions.

Forward:

```
token: INDENTATION
translating: INDENTATION
IN_INIT_LOCALS 1
IN_LOAD_INT 94
IN_LOAD_STR 7
IN_MAKE_TOKEN 2
IN_SEND
IN_POP
IN_GET_MATCH_LENGTH_R
IN_LOAD_INT 1
IN_SUB_INT
IN_SET_LOCAL_WC -1
IN_GET_LOCAL_R -1
IN_LOAD_GLOBAL_R
IN_GET_FIELD_R 0
IN_GET_LIST_MEM_R 1
IN_TST_GRTR
IN_JMP_FALSE 26
IN_LOAD_INT 92
IN_LOAD_STR 7
IN_MAKE_TOKEN 2
IN_SEND
IN_POP
IN_GET_LOCAL_R -1
IN_LOAD_GLOBAL_WV
IN_GET_FIELD_WV 0
IN_LIST_APPEND_WV
IN_POP
IN_JMP 37
IN_LOAD_INT 89
IN_LOAD_GLOBAL_R
IN_GET_STDIN
IN_GET_MATCH_LENGTH_R
IN_STREAM_PULL
IN_MAKE_TOKEN 2
IN_IGNORE
IN_POP
IN_POP_LOCALS 4 1
local tree downref: -1
local tree downref: 0
IN_STOP
sending queue item: NEWLINE
shifted: NEWLINE
```

Reverse:

```
sending back: DEDENT (artificial)
sending back: NEWLINE (artificial)
IN_STREAM_PULL_BKT
push back of 1 characters
IN_LOAD_GLOBAL_BKT
IN_GET_FIELD_BKT 0
IN_LIST_REMOVE_END_BKT
IN_STOP
backing up over effective terminal:
    identifier
```

Figure 6.16: An execution trace of off-side rule parsing in the forward and reverse directions.

The Python language definition states that, to resolve these ambiguities, a subscription takes precedence over a slicing and a simple slicing takes precedence over an extended slicing. It is possible to unify these language constructs, then resolve the ambiguity following the parse. However, we can also resolve these ambiguities at parse-time, which saves writing additional code. To do this we need the `subscription` form to take precedence over `slicing` and `simple_slicing` to take precedence over `extended_slicing`. We already have these ordered properly. However, our ordering algorithm does not work on this grammar in its natural form. When we attempt to force ordered choice with a unique empty production,

```
primary          → atom
                  | attributeref
                  | subscription
                  | slicing
                  | call

subscription     → primary '[' expression_list ']'

expression_list  → expression ( ',' expression )* ','?

slicing          → simple_slicing
                  | extended_slicing

simple_slicing   → primary '[' short_slice ']'

extended_slicing → primary '[' slice_list ']'

slice_list       → slice_item (',' slice_item)* ','?

slice_item       → expression
                  | proper_slice
                  | ellipsis

proper_slice     → short_slice
                  | long_slice

short_slice      → expression? ':' expression?

long_slice       → short_slice ':' expression?
```

Ambiguity 1: Subscription or Slicing

```
subscription → primary '[' expression_list ']' → primary '[' expression ']'

slicing → extended_slicing → primary '[' slice_list ']' → primary '[' expression ']'
```

Ambiguity 2: Simple Slicing or Extended Slicing

```
simple_slicing → primary '[' short_slice ']'

extended_slicing → primary '[' slice_list ']' → primary '[' slice_item ']'
       → primary '[' proper_slice ']' → primary '[' short_slice ']'
```

Figure 6.17: A section of the Python grammar showing the ambiguity between subscriptions and slicings. The derivations illustrate the ambiguities in the grammar. Subscription and slicing can both derive identical forms, as can simple slicing and extended slicing.

```
def primary
    [atom primary_ext*]

def primary_ext
    [attributeref]
|   [subscription]
|   [slicing]
|   [call]

def subscription
    ['[' expression_list ']']

def slicing
    [simple_slicing]
|   [extended_slicing]

def simple_slicing
    ['[' short_slice ']']

def extended_slicing
    ['[' slice_list ']']

def slice_list
    [slice_item slice_list_ext* ','?]

def slice_list_ext
    [',' slice_item]

def slice_item
    [expression]
|   [proper_slice]
|   [ellipsis]
```

Figure 6.18: A grammar fragment for Python subscriptions and slicings. To resolve ambiguities we must express the grammar as right-recursive.

---

as prescribed in Section 3.3.2, we are foiled by the left recursion. This problem is outlined in Section 3.3.3. The following example derivations show the left recursion.

```
primary → subscription → primary '[' expression_list ']'

primary → slicing → simple_slicing → primary '[' short_slice ']'

primary → slicing → extended_slicing → primary '[' slice_list ']'
```

Solving this problem requires that we transform the left recursion into right recursion. We are then able to take advantage of ordered choice to resolve the ambiguity. We give the corrected Python grammar fragment in Figure 6.18.

6.6 C++

C++ is an example of a language that has an ambiguous grammar definition and context-dependent language features. This property of C++ is shown in Section 2.4.2. We very

rarely find C++ parsers to be generated from a grammar. Many compilers use a hand-written recursive-descent parser, including GCC, OpenC++ and OpenWatcom. Since our system is generalized and supports context-dependent parsing, the ambiguous, context-dependent nature of the language does not cause a problem. We are free to implement symbol table maintenance and lookup using global data structures of our design, and we are able to resolve ambiguities by ordered choice.

In this section we give examples of symbol table maintenance during parsing and symbol table lookup during lexical analysis. We then give four examples of ambiguities in the language and show how each one can be solved with ordered choice.

6.6.1 Maintaining Symbol Information

We use semantic actions to manipulate a representation of the C++ name hierarchy. We also use them to prepare for name lookups by posting name qualifications. In the grammar fragment in Figure 6.19, we show some of the actions used to maintain the name hierarchy. These empty nonterminals open and close a C++ declaration. They are used to initialize and free a data structure that we use to collect information about a declaration. The information will be used when it is time to record the declaration in the C++ name hierarchy.

When a declaration is opened, a fresh instance of the structure is pushed onto the declaration-data stack. The instance is popped when the declaration has been completely parsed. Declarations are often unparsed during backtracking and automatic reverse execution performs the necessary unpop and unpush when these nonterminals are backtracked over. The actual insertion into the name hierarchy is shown in the example in Figure 5.4, which contains the `declarator_id` reduction action.

The grammar fragment in Figure 6.20 shows the definition for `qualifying_name`. The reduction actions for this definition set the top of the `qualNs` stack, which gives the containing object to be used in the next name lookup. Since the qualifying name is often

```
# Since declarations may be nested in function parameters we use stacks for
# accumulating declaration information.
global declaration_data_stack declarationData;
global int_stack templDecl;

# Open a declaration.
def declaration_start
    []
    {
        declarationData.push( construct declaration_data (
                isTypedef: 0, isFriend: 0, isTemplate: 0 ) [] )

        # Transfer the template flag and reset it.
        declarationData.top.isTemplate = templDecl.top
        templDecl.push( 0 )
    }

# Close a declaration.
def declaration_end
    []
    {
        declarationData.pop()
        templDecl.pop()
    }
```

Figure 6.19: Maintaining the C++ name hierarchy. These actions initialize and free data structures into which declaration data is collected. Since declarations are sometimes nested (e.g., function parameters) we must use a stack for this.

```
def qualifying_name
    [templ_class_id '<' template_argument_list_opt '>']
    {
        qualNs.top = lookupSpecialization( r1.obj, r3.argList )
    }
|   [class_id]           { qualNs.top = r1.obj }
|   [templ_class_id]     { qualNs.top = r1.obj }
|   [namespace_id]       { qualNs.top = r1.obj }
|   [typedef_id]         { qualNs.top = r1->obj->typedefOf }
```

Figure 6.20: The `qualifying_name` definition of C++ must post the qualifying object in preparation for name lookup.

---

used in grammar fragments that are ambiguous, this definition is backtracked over very frequently. Automatic reverse execution handles the necessary revert to the original value of `qualNs.top`.

6.6.2 Generating Tokens

Name lookup is a large part of producing a correct C++ parse tree. Names must be looked up in the symbol table so the parser can correctly distinguish different forms of the language. In Figure 6.21 we give the name lookup routine with some supporting functions omitted.

In the token-generation example of Section 5.1 we send a terminal to the parser. We also have the option to send a nonterminal type. Sending a nonterminal allows us to preserve the original type of the token as a child of the nonterminal. Later we can search for the identifier terminal type and find all names regardless of how they have been specialized. It also allows us to extract the common type from the specialized identifier so we can have a uniform interface to identifiers. This is a form of polymorphism.

The result of the name lookup function is an integer that represents the nonterminal we are translating the identifier to. First a token of type `lookup_id` is created. Then a tree with the returned nonterminal type is created, with the `lookup_id` token as the only child. Note that the global variable `qualNs` is modified by the lookup. Again, automatic reverse execution reverts this change when an identifier must be sent back to the input stream during backtracking.

6.6.3 Resolving Ambiguities

C++ has a number of ambiguities documented in the language standard [44]. These ambiguities can be resolved according to the standard by utilizing ordered choice. In the remainder of this section we describe how we have implemented the resolution of each ambiguity.

```
def class_id [lookup_id]
def namespace_id [lookup_id]
⋮
def template_id [lookup_id]

token lookup_id
    ptr lang_object obj
    ptr lang_object qualObj

    /( [a-zA-Z_] [a-zA-Z0-9_]* )/
    {
        str name = match_text
        ptr lang_object found = nil
        ptr lang_object qualObj = nil
        if ( qualNs.top ) {
            # Transfer the qualification to the token and reset it.
            qualObj = qualNs.top
            qualNs.top = nil

            # Lookup using the qualification.
            found = lookupWithInheritance( qualObj, name )
        }
        else {
            # No qualification, full search.
            found = unqualifiedLookup( name )
        }

        # If no match, return an unknown_id.
        int id = typeid unknown_id
        if ( found )
            id = found->typeId

        any LookupId = make_token( typeid lookup_id,
                pull(stdin, match_length), found, qualObj )
        send( make_tree( id, LookupId ) )

    }
```

Figure 6.21: The token-generation action that performs C++ name lookup. We send the translated token as a tree with the original type `lookup_id` sent as a child of that tree.

Ambiguity 1: Declaration or Expression

As shown in Figure 6.22, there is an ambiguity between declaration statements and expressions statements. To resolve this ambiguity, we follow the rule that any statement that

```
struct C {};
void f(int a)
{
    C(a)[5];      // declaration (ambiguous)
    C(a)[a=1];    // expression
}

def statement
    [declaration_statement]
|   [expression_statement]
|   ...
```

Figure 6.22: An example showing the ambiguity between declarations and expressions. It is resolved by placing the declaration statement ahead of the expression statement.

---

can be interpreted as a declaration is a declaration. We program this by specifying the declaration-statement production ahead of the expression-statement production, which is also shown in Figure 6.22.

Ambiguity 2: Function or Object Declaration

There is an ambiguity between a function declaration with a redundant set of parentheses around a parameter-declaration identifier, and an object declaration with an initialization using a function-style cast expression. An example is shown in Figure 6.23. Again, we apply the rule that any program text that can be a declaration is a declaration. Therefore we must prefer the function declaration. The resolution of this ambiguity is handled automatically by the ordered choice parsing strategy, because parameter specifications are innermost relative to object initializations. The relevant grammar fragment is also shown in Figure 6.23.

Ambiguity 3: Type-Id or Expression

In contexts where we can accept either a type-id or an expression, there is an ambiguity between an abstract function declaration with no parameters and a function-style cast. Two examples are shown in Figure 6.24. The resolution is that any program text that can be a

```
struct C {};
int f(int a)
{
    C x(int(a));  // function declaration (ambiguous)
    C y(int(1));  // object declaration
}

def init_declarator
    [declarator initializer_opt]

def declarator
    [ptr_operator_seq_opt declarator_id array_or_param_seq_opt]

def array_or_param_seq_opt
    [array_or_param_seq_opt array_or_param]
|   []

def array_or_param
    ['[' constant_expression_opt ']']
|   ['(' parameter_declaration_clause ')'
        cv_qualifier_seq_opt exception_specification_opt]

def initializer_opt
    ['=' initializer_clause]
|   ['(' expression ')']
|   []
```

Figure 6.23: An example showing the ambiguity between a function declaration with a single parameter that has parentheses around the name, and an object declaration that is initialized with a temporary object. The effect of ordered choice is that innermost constructs take precedence, which correctly resolves the ambiguity.

---

type-id is a type-id. We program this by specifying the productions that contain type-ids ahead of the productions that contain expressions, as shown in the second half of Figure 6.24.

Ambiguity 4: Anonymous Function Pointer or Variable

In contexts that accept both abstract declarators and named declarators, there is an ambiguity between an abstract function declaration with a single abstract parameter, and an

```
template<class T> class D {};
int f()
{
    sizeof(int());  // sizeof type-id (ambiguous)
    sizeof(int(1)); // sizeof expression

    D<int()> 1;   // type-id argument
    D<int(1)> 1;  // expression argument
}

def unary_expression
    ['sizeof' '(' type_id ')']
|   ['sizeof' unary_expression]

def template_argument
    [type_id]
|   [assignment_expression]
```

Figure 6.24: An example showing the ambiguity between type-id and expression. The resolution is to place productions that contain type-id ahead of productions that contain expressions.

---

object declaration with a set of parentheses around its name. This arises in function parameter lists. The resolution is to consider the text as an abstract function declaration with a single abstract parameter. We program this by specifying abstract declarators ahead of named declarators. An example of the ambiguity is shown in Figure 6.25. The resolution of the ambiguity is shown in the second half of the figure.

6.6.4 C++ Class Method Bodies

In most of the C++ language, types, classes and identifiers need to be declared before they are used. This means that most parsing can be done in a single pass. There is one area of the language where this is not true, however. Class member functions are allowed to reference class members that are further down in the class. This requires delaying the parsing of class member functions until after a class has been fully parsed. Note that we need to delay parsing, but we do not want to delay parsing until the end of the input has been reached.

```
struct C {};
void f(int (C)); // anonymous function pointer parameter (ambiguous)
void f(int (x)); // variable parameter

def parameter_declaration
    [decl_specifier_seq param_declarator_opt parameter_init_opt]

def param_declarator_opt
    [abstract_declarator]
|   [declarator]
|   []
```

Figure 6.25: An example showing the ambiguity between an anonymous function pointer and an object declaration with parentheses around the name. The resolution is to place abstract declarators ahead of named declarators.

This would allow globally defined names that appear after a class to be visible to the body of a class member function.

In Figure 6.26 we demonstrate how we can perform this two-stage parsing. We first capture class member functions as text. When the parsing of a class completes we can iterate through the class member functions and reparse the captured text as C++ program code. At this time all class members will have been inserted into the symbol table. We use the reduction action of the class definition to perform this task.

6.7 Summary

In this chapter we give examples of difficult parsing problems that can be handled easily in our system. These parsing problems are taken from a variety of common computer languages, including HTTP (6.1), DNS (6.2), HTML (6.3), Ruby (6.4), Python (6.5) and C++ (6.6). In Chapter 1 we identified three requirements for a transformation system's parsing engine. The parsing system must be generalized, it must be able to handle languages with varying lexical rules, and it must be able to handle languages with context-dependent features. The solutions to the parsing problems in this chapter demonstrate our ability to

```
def class_specifier
    [class_head base_clause_opt '{' class_member* class_body_end '}']
    {
        # At this point the class has been popped from the declaration and
        # lookup stacks. Add them back.
        lookupNs.push( r1.class )
        declNs.push( r1.class )

        # Visit class function bodies, but skip nested classes.
        for CFB: class_function_body in topdown_skipping(lhs, typeid class_specifier)
        {
            # Reparse the text of the class function body as a function body.
            function_body FB = reparse function_body( CFB )

            # Replace the class function body with the parsed function body.
            CFB = construct class_function_body
                [FB]
        }

        lookupNs.pop()
        declNs.pop()
    }
```

```
def class_member
    [member_declaration]
|   [access_specifier ':']

def member_declaration
    [declaration_start
        member_declaration_forms
        declaration_end ';']
|   [class_function_definition]
|   [using_declaration]
|   [template_declaration]

def class_function_definition
    [function_def_declaration
        ctor_initializer_opt
        class_function_body
        function_def_end]

def class_function_body
    ['{' cfb_conts]
|   [function_body]
```

```
def function_body
    [function_body_begin '{'
        statement_rep
        function_body_end '}']

def cfb_conts
    [cfb_item* cfb_close]

def cfb_item
    [cfb_data]
|   [cfb_string]
|   [cfb_comment]
|   [cfb_open cfb_item* cfb_close]

lex cfb_conts
{
    token cfb_open /'{'/
    token cfb_close /'}'/
    token cfb_string /
        '\'' ( [^'\\\n] | '\\' any )* '\'' |
        '"' ( [^"\\\n] | '\\' any )* '"'/
    token cfb_comment /'//' [^\n]* '\n'/
    token cfb_data /[^{}'"/]+ | '/'/
}
```

Figure 6.26: Delayed parsing of C++ class method bodies. Initial parsing collects the bodies as text, interpreting comments and strings, and balancing curly brackets. When the class parsing is complete the method bodies are parsed fully.

meet all three of these requirements.

In the next chapter we conclude with a brief summary, an overview of our contributions, and a discussion of future research directions for improving upon this work.

Chapter 7

Summary and Conclusion

In this work we set out to design a new transformation system with a parsing engine that is more capable than what exists in the current systems. Our secondary goal is to make transformation languages more accessible by designing a language that is closer to general-purpose languages than the existing transformation languages.

In Chapter 2 we surveyed existing source transformation systems, focusing on generalized parsing techniques and the problems associated with context-dependent parsing. In Chapter 3 we chose backtracking LR as the parsing algorithm and addressed the issues of controlling the parse of ambiguous grammars, pruning the search space, handling errors, and handling languages with varying lexical requirements. In Chapter 4 we presented the design of a new source transformation language. We first presented the features borrowed from general-purpose languages, then moved on to describe the transformation-related features. Following the definition of our language we showed how it is useful for analyzing and rewriting trees, then gave some implementation details. In Chapter 5 we addressed the issue of generalized context-dependent parsing. We showed how the parsing engine and virtual machine can be extended to allow the automatic reversal of global-state modifications. In Chapter 6 we showed how our system can solve parsing problems that cause other systems trouble. These problems came from a selection of document formats, network protocols and

programming languages. In the remainder of this chapter we detail the six contributions that we have made, we discuss limitations and future work, then we conclude the thesis.

7.1 Contributions

1. ***Ordered Choice for Backtracking LR.***

   We assign an ordering to conflicting shift and reduce actions that causes the parser to emulate the parsing strategy of a generalized top-down parser with ordered choice. In some cases, common prefixes inhibit the desired top-down strategy. Unique empty productions can be inserted at the beginning of productions to force a localized top-down approach. This guarantees that the parser attempts to parse mutually ambiguous productions in the order in which they are given. Using our method, we can apply a top-down backtracking strategy where needed for resolving ambiguities, while retaining the speed of LR parsing for sections of the grammar that are deterministic and amenable to bottom-up parsing.

2. ***Local Commit Points for Backtracking LR.***

   Declarative commit points can be used to eliminate fruitless backtracking and improve performance in a localized manner. We have added two types of commit points. A strong commit clears all alternative parses, thereby preventing any backtracking over previous decisions. A localized commit, which is unique to this work, removes alternatives underneath a single nonterminal, while preserving earlier alternatives.

3. ***Generalized Grammar-Dependent Tokenization.***

   We employ grammar-dependent tokenization and make it generalized by incorporating it into the backtracking algorithm. When a token is requested by the parser the current parse state is used to determine which lexical regions can generate tokens that can be accepted. The list of regions is ordered using our approximate ordered choice

algorithm and the first one is selected. The next possible region is encoded in the token as an alternative. This approach makes it possible to mix grammar fragments that have different lexical regions, and to allow the backtracker to switch between the lexical regions as necessary.

4. ***A New Tree Transformation Language.***

   We have designed a new transformation language that we hope will appeal to developers. Recognizing that no existing transformation language includes the imperative language paradigm, we have designed a transformation language that borrows from general-purpose languages. We identify the essential transformation language features such as a formal-language type system, pattern matching, tree construction and tree traversal. We merge these features with a rudimentary imperative-style programming language foundation. We hope that this approach will make transformation systems more accessible and ease the writing of any ancillary algorithms that are not suited to the rewrite paradigm.

5. ***Undo Actions for Reverting Side Effects.***

   We introduce a new class of semantic actions for reverting changes made to the global state. We call these undo actions. When the backtracking LR parser performs the reverse of a reduction by replacing a nonterminal with its children, the undo action is executed. These actions permit the parser to backtrack over areas of input text that require global data modification in preparation for handling context-dependent language features.

6. ***Automatic Reverse Execution for Reverting Side Effects.***

   To free the user from having to write undo actions, which in some cases can be difficult to write, we employ reverse execution. As the virtual machine of our transformation language moves over instructions it logs the reverse instructions that are necessary to revert the effects of the forward execution. This is only necessary for instructions that

modify globally accessible data. Changes to local variables do not need to be reverted. These recorded undo actions can then be executed during backtracking. This allows the user to write semantic actions that modify global state without being concerned about correctly reverting the changes during backtracking to avoid corruption.

7.2 Limitations and Future Work

7.2.1 Out-of-Order Parse Correction

In Section 3.3.2 we show that our ordering algorithm does not always order productions in the sequence that they are given. In some cases it is necessary to force the sequential ordering by inserting unique empty productions. This only works when the affected nonterminal is free of left recursion.

The problem of detecting out-of-order parses and eliminating them by inserting unique empty productions is a task that we leave up to the user. It would be desirable to have a static analysis that is able to detect out-of-order parses and automatically correct the problem by inserting unique empty productions where appropriate. In initial investigations we found this to be a difficult problem, closely related to the detection of ambiguities in context-free grammars. Detecting ambiguities has been shown to be an undecidable problem [41].

An alternate strategy for guaranteeing that no out-of-order parses are possible might be to begin by inserting unique empty productions at the beginning of every production, then later eliminate those that are not necessary. This approach of maintaining in-order parses might be easier than the approach of detecting out-of-order parses.

When we insert unique empty productions at the beginning of every production we guarantee that no input is parsed out of order. This causes a backtracking LR parser to behave like a generalized top-down parser and only works when we have a grammar free of left recursion. This idea was tested with a C++ grammar. We automatically inserted

unique empty productions at the beginning of every production that did not contain left recursion, direct, indirect or hidden. The resulting parser produced the same output, but performance slowed by a factor of 20 and there was an increase in the number of states by a factor of 2.

If automatic detection of out-of-order parses proves too difficult or is found to be unnecessary in practice, it may be worthwhile to pursue methods for analyzing an explicitly specified pair of ambiguous productions for potential out-of-order parses. This would allow a user to mark known ambiguities and rely on the parsing engine to decide if unique empty productions are necessary.

7.2.2 Context-Dependent parsing with GLR

In Section 2.4.3 we show that proper GLR [94] parsing is incompatible with the maintenance of global state. The problem is that the diverging global state can prevent the merging of parse stacks. This merging is what sets GLR apart from a naive nondeterministic implementation of generalized parsing.

However, even though the maintenance of global data prevents us from doing proper GLR we may be able to support context-dependent parsing using an approximation of GLR. We could merge according to the GLR algorithm only when doing so did not require that we also merge distinct global contexts. Determining the value of such a system would then become a matter of empirical assessment. We may find that in practice, when used with real grammars, that a near-GLR algorithm that supports context-dependent parsing is worthwhile.

Since we already have a value-based tree semantics and an implementation with copy-on-write tree sharing, we may be not far from an efficient implementation of this idea. Furthermore, if we were able to design and build such a system using the same transformation language that we have designed in this work then we would be in a good position to perform a comparison of the two methods.

7.2.3 Grammar Modularity and Reuse

From the perspective of grammar programming, our system has a number of missing features that we have left for future work. File inclusions allow a program's definitions to be partitioned into files. The program can then be explicitly assembled from the files by including them into a master program file. Name spaces allow the identifiers that represent program entities to be organized into a hierarchy, which reduces the potential for name collisions when program fragments are combined. Grammar overrides [27] allow the user to make modifications to a reusable base grammar by overriding particular type definitions. An override can specify an entirely new set of productions for a type, or it can extend the existing set of productions with additional cases. All of these features facilitate grammar modularity and reuse, and we must provide language constructs for them. The language constructs for grammar overrides must accommodate attribute definitions and commit points, which will both be affected by overrides.

7.3 Implementation

We have implemented our system and named it COLM, for COmputer Language Manipulation. We have licensed it as free software and the source code is available from the project's web page [93]. All features have been implemented and tested. However, the system is constantly being improved and extended, with frequent rewrites of major subsystems taking place. A large change can render specific language features non-functional. At the time of writing, a stable release is not available.

7.4 Conclusion

Transformation systems are very useful for the analysis and transformation of computer languages. They are powerful tools for parsing input, and traversing and rewriting trees.

When applied in practice a serious problem is evident. Some languages are difficult to express using the pure context-free language facilities that transformation systems provide. Context-dependent languages cause these systems trouble because the manipulation of global data in support of tasks such as identifier lookup or tag matching is incompatible with generalized parsing. We set out to solve this problem by designing a new transformation system with a generalized parsing engine that supports the manipulation of global data structures at parse time, thus enabling generalized context-dependent parsing. In the process we design a new transformation language and take the opportunity to create a language that is closer to general-purpose languages. The result of this work is COLM: a new transformation system that is useful for the analysis and transformation of a larger group of computer languages.

Bibliography

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26, May 1991.

[2] Tankut Akgul and Vincent J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Transactions Software on Engineerning Methodologies*, 13(2):149–198, April 2004.

[3] John Aycock. Why Bison is becoming extinct. *ACM Crossroads*, Xrds-7.5, 2002.

[4] John Aycock and Nigel Horspool. Faster generalized LR parsing. In *8th International Conference on Compiler Construction (CC'99)*, volume 1575 of *Lecture Notes in Computer Science*, pages 32–46, 1999.

[5] John Aycock and Nigel Horspool. Practical Earley parsing. *The Computer Journal*, 45(6):620–630, November 2002.

[6] John Aycock and R. Nigel Horspool. Directly-executable Earley parsing. In *10th International Conference on Compiler Construction (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 229–243, 2001.

[7] John Aycock and R. Nigel Horspool. Schrodinger's token. *Software: Practice and Experience*, 31(8):803–814, July 2001.

[8] John Aycock, R. Nigel Horspool, Jan Janousek, and Borivoj Melichar. Even faster generalised LR parsing. *Acta Informatica*, 37(9):633–651, June 2001.

[9] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Elco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In *3rd International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 65–75. IEEE Computer Society Press, 2003.

[10] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In *18th Conference on Rewriting Techniques and Applications (RTA'07)*, Lecture Notes in Computer Science, pages 36–47, 2007.

[11] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *26th International Conference on Software Engineering (ICSE'04)*, pages 625–634. IEEE Computer Society, 2004.

[12] Ralph Becket and Zoltan Somogyi. DCGs + memoing = packrat parsing but is it worth it? In *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *Lecture Notes in Computer Science*, pages 182–196, 2008.

[13] Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *27th Annual Meeting of the Association for Computational Linguistics (ACL'89)*, pages 143–151. Association for Computational Linguistics, 1989.

[14] Michael R. Birch, Christopher M. Boroni, Frances W. Goosey, Samuel D. Patton, David K. Poole, Craig M. Pratt, and Rockford J. Ross. DYNALAB: a dynamic computer science laboratory infrastructure featuring program animation. In *26th SIGCSE technical symposium on Computer science education (SIGCSE'95)*, pages 29–33. ACM Press, 1995.

[15] Bitan Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Notices*, 34(4):61–69, April 1999.

[16] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In *2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 55–70, 1998.

[17] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible markup language (XML) 1.0, 2006. http://www.w3.org/TR/2006/REC-xml-20060816/.

[18] R. A. Brooker. A note: Top-to-bottom parsing rehabilitated? *Communications of the ACM*, 10(4):223–224, April 1967.

[19] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[20] Ruknet Cezzar. The design of a processor architecture capable of forward and reverse execution. *IEEE Southeastcon '91*, 2:885–890, 1991.

[21] K. Mani Chandy and Chittoor V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21(6):546–556, June 1972.

[22] John Cocke. *Programming languages and their compilers: Preliminary notes*. Courant Institute of Mathematical Sciences, New York University, 1969.

[23] Jonathan J. Cook. Reverse execution of java bytecode. *The Computer Journal*, 45(6):608–619, November 2002.

[24] Rafael Corchuelo, José A. Pérez, Antonio Ruiz, and Miguel Toro. Repairing syntax errors in LR parsers. *ACM Transactions on Programming Languages and Systems*, 24(6):698–710, November 2002.

[25] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[26] Mark Crispin. RFC 3501: Internet message access protocol - version 4rev1, 2003.

[27] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Agile parsing in TXL. *Journal of Automated Software Engineering*, 10(4):311–336, October 2003.

[28] Franklin L. DeRemer. *Practical translators for LR(k) languages*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, 1969.

[29] Jay Earley. An efficient context-free algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[30] Torbjörn Ekman and Görel Hedin. The Jastadd extensible java compiler. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA'07)*, volume 42(10) of *SIGPLAN Notices*, pages 1–18, 2007.

[31] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1, 1999.

[32] Kathleen Fisher and Robert Gruber. PADS: A domain-specific language for processing ad hoc data. In *2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, volume 40(6) of *SIGPLAN Notices*, pages 295–304, 2005.

[33] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 2008.

[34] Robert W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, October 1967.

[35] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming (ICFP'02)*, pages 36–47. ACM Press, 2002.

[36] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'04)*, pages 111–122. ACM Press, 2004.

[37] Richard A. Frost. Guarded attribute grammars: top down parsing and left recursive productions. *SIGPLAN Not.*, 27(6):72–75, 1992.

[38] Mahadevan Ganapathi. Semantic predicates in parser generations. *Computer Languages*, 14(1):25–33, 1989.

[39] Robert Grimm. Better extensibility through modular syntax. In *2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI'06)*, pages 38–51. ACM Press, 2006.

[40] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques - A Practical Guide*. Ellis Horwood, Chichester, England, 1990.

[41] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[42] Freeman Y. Huang. *Type-safe Computation with Heterogeneous Data*. PhD thesis, Queen's University, School of Computing, 2007.

[43] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[44] The International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. American National Standards Institute, First edition, September 1998.

[45] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[46] Steven C. Johnson. YACC: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, 1979.

[47] Adrian Johnstone and Elizabeth Scott. Generalised recursive descent parsing and follow determinism. In *7th International Conference on Compiler Construction (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 16–30, 1998.

[48] Adrian Johnstone and Elizabeth Scott. Generalised regular parsing. In *12th International Conference on Compiler Construction (CC'03)*, volume 2622 of *Lecture Notes in Computer Science*, pages 232–246. Springer-Verlag, 2003.

[49] Adrian Johnstone and Elizabeth Scott. Recursion engineering for reduction incorporated parsers. In *5th Workshop on Language Descriptions, Tools, and Applications (LDTA'05)*, volume 141(4) of *Electronic Notes in Theoretical Computer Science*, pages 143–160, 2005.

[50] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Generalised parsing: Some costs. In *13th International Conference on Compiler Construction (CC'04)*, volume 2985 of *Lecture Notes in Computer Science*, pages 89–103, 2004.

[51] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. The grammar tool box: A case study comparing GLR parsing algorithms. In *4th Workshop on Language Descriptions, Tools, and Applications (LDTA'04)*, volume 110 of *Electronic Notes in Theoretical Computer Science*, pages 97–113, 2004.

[52] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Evaluating GLR parsing algorithms. *Science of Computer Programming*, 61(3):228–244, August 2006.

[53] Adrian Johnstone, Elizabethi Scott, and Giorgios Economopoulos. The GTB and PAT tools. In *4th Workshop on Language Descriptions, Tools, and Applications (LDTA'04)*, volume 110 of *Electronic Notes in Theoretical Computer Science*, pages 173–175, 2004.

[54] Karl Trygve Kalleberg and Eelco Visser. Fusing a transformation language with an open compiler. In *7th Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, Electronic Notes in Theoretical Computer Science, pages 18–31, 2007.

[55] Tadao Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass., 1965.

[56] John C. Klensin. RFC 2821: Simple mail transfer protocol, 2001.

[57] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *1st ASMICS Workshop on Parsing Theory*, pages 1–20, 1994.

[58] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.

[59] Jan Kort and Ralf Lammel. Parse-tree annotations meet re-engineering concerns. In *3rd International Workshop on Source Code Analysis and Manipulation, (SCAM'03)*, pages 161–170. IEEE Computer Society, 2003.

[60] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In *2nd Colloquium on Automata, Languages and Programming (ICALP'74)*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269, 1974.

[61] Michael E. Lesk and Eric Schmidt. Lex – A lexical analyzer generator. Technical report, Bell Laboratories, 1975. CS Technical Report No. 39.

[62] Philip M. Lewis II and Richard E. Stearns. Syntax-directed transduction. *Journal of the ACM*, 15(3):465–488, July 1968.

[63] Wolfgang Lohmann, Günter Riedewald, and Markus Stoy. Semantics-preserving migration of semantic rules during left recursion removal in attribute grammars. In *4th Workshop on Language Descriptions, Tools, and Applications (LDTA'04)*, volume 110 of *Electronic Notes in Theoretical Computer Science*, pages 133–148, 2004.

[64] Andrew J. Malton. The denotational semantics of a functional tree-manipulation language. *Computer Languages*, 19(3):157–168, July 1993.

[65] Andrew J. Malton, Kevin A. Schneider, James R. Cordy, Thomas. R. Dean, Darren Cousineau, and Jason Reynolds. Processing software source text in automated design recovery and transformation. In *9th International Workshop on Program Comprehension (IWPC'01)*, pages 127–134. IEEE Computer Society, 2001.

[66] Sylvain Marquis, Thomas R. Dean, and Scott Knight. Packet decoding using context sensistive parsing. In *2006 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'06)*, pages 263–274. IBM Press, 2006.

[67] Philippe McLean and R. Nigel Horspool. A faster Earley parser. In *6th International Conference on Compiler Construction (CC'96)*, volume 1060 of *Lecture Notes in Computer Science*, pages 281–293, 1996.

[68] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *13th International Conference on Compiler Construction (CC'04)*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88, 2004.

[69] Gary H. Merrill. Parsing non-LR(k) grammars with Yacc. *Software, Practice and Experience*, 23(8):829–850, August 1993.

[70] Paul V. Mockapetris. RFC 1035: Domain names - Implementation and specification, 1987.

[71] Leon Moonen. Generating robust parsers using island grammars. In *8th Working Conference on Reverse Engineering, (WCRE'01)*, pages 13–22. IEEE Computer Society, 2001.

[72] Pierre-Étienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *12th International Conference on Compiler Construction (CC'03)*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76, 2003.

[73] Mark-Jan Nederhof and Janos J. Sarbo. Increasing the applicability of LR parsing. In H. Bunt and M Tomita, editors, *Recent advances in parsing technology*, pages 37–57. Kluwer Academic Publishers, Netherlands, 1996.

[74] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 61–75. Kluwer Academic Publishers, Netherlands, 1991.

[75] Richard A. O'Keefe. O(1) reversible tree navigation without cycles. *Theory and Practice of Logic Programming*, 1(5):617–630, September 2001.

[76] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. Binpac: a yacc for writing application protocol parsers. In *6th ACM SIGCOMM conference on Internet measurement (IMC '06)*, pages 289–300. ACM Press, 2006.

[77] Terence J. Parr and Russell W. Quong. ANTLR: A predicated LL(k) parser generator. *Software, Practice and Experience*, 25(7):789–810, July 1995.

[78] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, May 1980.

[79] James F. Power and Brian A. Malloy. Exploiting metrics to facilitate grammar transformation into LALR format. In *2001 ACM Symposium on Applied Computing (SAC'01)*, pages 636–640. ACM Press, 2001.

[80] The Python Software Foundation. The Python programming language. http://www.python.org/.

[81] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification, 1999. http://www.w3.org/TR/1999/REC-html401-19991224.

[82] Jan G. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, Wiskunde en Informatica, 1992.

[83] Pete Resnick. RFC 2822: Internet message format, 2001.

[84] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *ACM SIGPLAN 1989 Conference on Programming language design and implementation (PLDI'89)*, pages 170–178. ACM Press, 1989.

[85] Sylvain Schmitz. *Approximating Context-Free Grammars for Parsing and Verification*. PhD thesis, Université de Nice - Sophia Antipolis, Sciences et Technologies de lInformation et de la Communication, 2007.

[86] Sylvain Schmitz. An experimental ambiguity detection tool. In *7th Workshop on Language Descriptions, Tools and Applications (LDTA'07)*, volume 203(2) of *Electronic Notes in Theoretical Computer Science*, pages 69–84, 2008.

[87] Elizabeth Scott and Adrian Johnstone. Table based parsers with reduced stack activity. Technical Report CSD-TR-02-08, Royal Holloway, University of London, Department of Computer Science, May 2003.

[88] Elizabeth Scott and Adrian Johnstone. Generalized bottom up parsers with reduced stack activity. *The Computer Journal*, 48(5):565–587, September 2005.

[89] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618, July 2006.

[90] Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. BRNGLR: A cubic Tomita-style GLR parsing algorithm. *Acta Informatica*, 44(6):427–461, October 2007.

[91] Rok Sosič. History cache: hardware support for reverse execution. *SIGARCH Computer Architecture News*, 22(5):11–18, December 1994.

[92] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Robust multilingual parsing using island grammars. In *2003 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'03)*, pages 149–161. IBM Press, 2003.

[93] Adrian Thurston. The Colm programming language. http://www.complang.org/colm/.

[94] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, 1986.

[95] Stephen H. Unger. A global parser for context-free phrase structure grammars. *Communications of the ACM*, 11(4):240–247, 1968.

[96] Leslie G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):208–315, April 1975.

[97] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, July 2002.

[98] Mark G. J. van den Brand and Paul Klint. ATerms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, 49(1):55–64, 2007.

[99] Mark G. J. van den Brand, Paul Klint, and Jurgen J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, April 2003.

[100] Mark G. J. van den Brand, Steven Klusener, Leon Moonen, and Jurgen J. Vinju. Generalized parsing and term rewriting - semantics directed disambiguation. In *3rd Workshop on Language Descriptions, Tools and Applications (LDTA'03)*, volume 82(3) of *Electronic Notes in Theoretical Computer Science*, pages 575–591, 2003.

[101] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *11th International Conference on Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, 2002.

[102] Mark G. J. van den Brand, Alex Sellink, and Chris Verhoef. Current parsing techniques in software renovation considered harmful. In *6th International Workshop on Program Comprehension (IWPC'98)*, pages 108–117. IEEE Computer Society, 1998.

[103] Eric R. Van Wyk and August C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *6th International Conference on Generative Programming and Component Engineering, (GPCE'07)*, pages 63–72. ACM Press, 2007.

[104] Krishnamurti Vijay-Shanker and David J. Weir. Polynomial time parsing of combinatory categorial grammars. In *28th International Meeting of the Association for Computational Linguistics (ACL'90)*, pages 1–8. Association for Computational Linguistics, 1990.

[105] Eelco Visser. *Syntax definition for language prototyping.* PhD thesis, University of Amsterdam, Wiskunde en Informatica, 1997.

[106] Eelco Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, 1999.

[107] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238, 2004.

[108] Elco Visser and James R. Cordy. Tiny imperative language. http://www.program-transformation.org/Sts/TinyImperativeLanguage.

[109] Daniel Waddington and Bin Yao. High-fidelity C/C++ code transformation. *Science of Computer Programming*, 68(2):64–78, 2007.

[110] Philip Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture (FPCA'85)*, pages 113–128. Springer-Verlag, 1985.

[111] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *ACM SIGPLAN 1997 conference on Programming language design and implementation (PLDI'97)*, pages 31–43. ACM Press, 1997.

[112] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.