Packrat Parsing:
a Practical Linear-Time Algorithm with Backtracking

by

Bryan Ford

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2002

Author ....
Department of Electrical Engineering and Computer Science
September 3, 2002

Certified by.......................... ....
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Packrat Parsing:
a Practical Linear-Time Algorithm with Backtracking

by
Bryan Ford

Submitted to the Department of Electrical Engineering and Computer Science
on September 3, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Packrat parsing is a novel and practical method for implementing linear-time parsers for grammars defined in Top-Down Parsing Language (TDPL). While TDPL was originally created as a formal model for top-down parsers with backtracking capability, this thesis extends TDPL into a powerful general-purpose notation for describing language syntax, providing a compelling alternative to traditional context-free grammars (CFGs). Common syntactic idioms that cannot be represented concisely in a CFG are easily expressed in TDPL, such as longest-match disambiguation and "syntactic predicates," making it possible to describe the complete lexical and grammatical syntax of a practical programming language in a single TDPL grammar.

Packrat parsing is an adaptation of a 30-year-old tabular parsing algorithm that was never put into practice until now. A packrat parser can recognize any string defined by a TDPL grammar in linear time, providing the power and flexibility of a backtracking recursive descent parser without the attendant risk of exponential parse time. A packrat parser can recognize any LL(k) or LR(k) language, as well as many languages requiring unlimited lookahead that cannot be parsed by shift/reduce parsers. Packrat parsing also provides better composition properties than LL/LR parsing, making it more suitable for dynamic or extensible languages. The primary disadvantage of packrat parsing is its storage cost, which is a constant multiple of the total input size rather than being proportional to the nesting depth of the syntactic constructs appearing in the input.

Monadic combinators and lazy evaluation enable elegant and direct implementations of packrat parsers in recent functional programming languages such as Haskell. Three different packrat parsers for the Java language are presented here, demonstrating the construction of packrat parsers in Haskell using primitive pattern matching, using monadic combinators, and by automatic generation from a declarative parser specification. The prototype packrat parser generator developed for the third case itself uses a packrat parser to read its parser specifications, and supports full TDPL notation extended with "semantic predicates," allowing parsing decisions to depend on the semantic values of other syntactic entities. Experimental results show that all of these packrat parsers run reliably in linear time, efficiently support "scannerless" parsing with integrated lexical analysis, and provide the user-friendly error-handling facilities necessary in practical applications.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank my advisor Frans Kaashock for his thoughtful and attentive guidance and infinite patience. I also wish to thank my colleagues Chuck Blake and Russ Cox for their helpful comments and suggestions.

Portions of this work appear in the proceedings of the 2002 International Conference on Functional Programming [8].

Contents

List of Figures

Chapter 1

Introduction

Practically all languages in common use today, both human-readable and machine-readable, are founded on the idea of expressing information in linear form, as sequences of written marks or symbols. Text in a written language is usually represented electronically as a *string*, or a sequence of characters chosen from some standardized character set. The character set in use may vary from one language or locale to another (e.g., ASCII versus Unicode), as well as other details such as the normal writing direction, the conventions for expressing relevant formatting information such as whitespace and line breaks, and the method of expressing composite symbols such as accented letters. Regardless of such details, however, the first task of any language processing application is to analyze these flat strings of characters into meaningful higher-level groupings such as words, phrases, sentences, expressions, or statements. This task of deriving useful high-level structural information from linear text is known as *syntax analysis* or *parsing*.

1.1 Expressing Syntax with Grammars

In order to create a parser for a particular language, or even just to reason formally about what kinds of strings are meaningful or well-formed in that language, we must have a way of expressing and understanding the language's syntactic structure. For this purpose we commonly use a *grammar*, which is a concise representation of the structure of one language, expressed in another (ideally very small and simple) language. The language whose structure the grammar is intended to represent—the language we are "talking about"—is known as the *object language*, whereas the language the grammar is *expressed in* is known as the *grammar definition language*. Being able to express the syntactic structure of a language concisely with a grammar is especially important for programming languages and other languages expressly designed for precision and machine-readability, because grammars can be used to reason about the properties of a language mathematically or with the help of mechanical tools.

The most common type of grammar in use today is the *context-free grammar* (CFG), typically expressed in the ubiquitous Backus-Naur Form (BNF). A context-free grammar essentially specifies a set of mutually recursive rules that describe how strings in the described language may be written. Each rule or *production* in a CFG specifies one way in which a syntactic variable or *nonterminal* can be expanded into a string. A nonterminal may expand to a string containing more nonterminals, which are expanded in turn until no nonterminals remain. Because there can be multiple ways to expand a particular nonterminal, the grammar can express an infinite set of strings having a well-defined hierarchical structure. Parsing a string whose syntax is specified by a CFG involves reversing this process: determining from a fully expanded string containing only atomic characters or *terminals*, what sequence (or sequences) of expansion steps, if any, can produce that string. This task is complicated by the fact that CFGs often contain ambiguities: both *local ambiguities*, in which the correct interpretation of a portion of a string can only be determined from the context in which it appears; and *global ambiguities*, in which a string as a whole may have multiple valid syntactic interpretations.

1.2 Top-Down Parsing Language (TDPL)

Another method of expressing syntax formally is through a set of rules describing how the strings in a language are to be *read* rather than written. Top-Down Parsing Language, or TDPL [3], is a formal scheme for describing language syntax that shares a close relationship with top-down or recursive-descent parsing. TDPL was developed at around the same time most of the classic CFG parsing algorithms were invented, but at that time it was used only as a formal model for the study of certain top-down parsing algorithms.

The first contribution of this thesis is to develop TDPL into a practical syntactic notation in its own right, and to demonstrate the advantages in expressiveness of TDPL over the CFG paradigm. Many machine-readable languages in use today are much easier to express in TDPL than with a pure CFG, particularly because of TDPL's natural support for the pervasive longest-match disambiguation policy and other common localized disambiguation rules. Both the low-level lexical and the high-level grammatical syntax of a language can be expressed in a single unified TDPL grammar, whereas in the CFG paradigm these two levels of syntax must usually be considered separately. Even when lexical analysis is separated from grammatical analysis, most CFG-based specifications of practical languages must include some "supplemental rules" describing how certain ambiguities in the CFG are to be resolved. Usually these supplemental rules are described either only informally, or in reference to the behavior of a specific parsing algorithm (e.g., a bottom-up shift/reduce parser) that is expected to be used to parse the language. In the TDPL paradigm, these rules fit cleanly into the basic syntactic notation and do not require any special extensions.

1.3 Packrat Parsing

The simplest method of parsing a string according to a TDPL grammar is using a top-down, recursive-descent parser with backtracking capability. This observation should come as no surprise since TDPL was created in the first place as a formalization for such parsers. Unfortunately, on many TDPL grammars an ordinary recursive-descent parser can take exponential runtime to parse input strings. The reason for this blow-up is that the backtracking recursive-descent parser may redundantly compute intermediate results multiple times. If memoization is used to prevent this redundant computation, however, then any TDPL grammar can be parsed in linear time, proportional to the size of the input string.

Although the theoretical existence of a linear-time TDPL parsing algorithm was previously known, it was apparently viewed as impractical and never implemented, because it involves storing a large table of intermediate results that grows in proportion to the length of the input string. Furthermore, the naïve approach to the construction of this table would compute many results that are never needed, even though no single result is computed more than once. On modern machines, the storage cost of this memoization table is no longer a serious problem for typical problem sizes, and lazy evaluation techniques make it possible to avoid computing results that are not needed.

The second main contribution of this thesis is *packrat parsing*, a version of the tabular linear-time TDPL parsing algorithm refined for practical use. Packrat parsing is particularly well-suited to implementation in modern lazy functional programming languages such as Haskell, though packrat parsing can of course be implemented in other languages. In a lazy functional language, packrat parsing requires only ordinary algebraic data types, with no explicit hash tables or other costly lookup structures. A packrat parser implemented in this way is almost identical in style and structure to a conventional recursive-descent parser with backtracking: the packrat parser essentially just uses a different method to "tie up" the mutually recursive calls that are made between the functions comprising the parser, so as to provide the memoization necessary to ensure parsing in linear time.

1.4 Automatic Generation of Packrat Parsers

Although packrat parsing is simple enough to implement by hand, particularly in a lazy functional language, constructing packrat parsers would be even easier with the help of an automatic parser generator along the lines of YACC in the C world or Happy [9] and Mímico [5] in the Haskell world. The third major contribution of this thesis is Pappy, a prototype packrat parser generator that takes declarative parser specifications and generates packrat parsers in Haskell.

The parser specification language accepted by Pappy is based on the TDPL notation developed in this thesis, with extensions to allow the parser to compute *semantic values* such as abstract syntax trees as an input string is parsed. As is the tradition for such parser generators, these semantic values are computed with the help of code fragments in the implementation language (Haskell) embedded in the parser specification. Pappy also extends the TDPL notation with support for *semantic predicates*, through which parsing decisions can depend on semantic values and not just on the success or failure of a syntactic rule. For example, using semantic predicates, syntactic character classes can be expressed in terms of Haskell functions such as `isAlpha` without requiring specialized notation.

1.5 Thesis Overview

The rest of this thesis is organized into the following chapters:

- Chapter 2 briefly reviews the two most common syntax notation schemes, regular expressions and context-free grammars, then presents the TDPL notation system used in this thesis and explores its expressiveness in comparison with CFG notation.

- Chapter 3 introduces the packrat parsing algorithm and demonstrates the implementation of packrat parsers in Haskell through a series of examples. The chapter also presents extended techniques such as integrated lexical analysis, packrat parsing with monadic combinators, and user-friendly error handling.

- Chapter 4 presents Pappy, the prototype packrat parser generator. The chapter first describes the TDPL-based specification language accepted by Pappy, then details its operation and the grammar transformations it performs in order to produce efficient parsers.

- Chapter 5 describes experimental results measuring the storage cost and performance of both hand-implemented and automatically-generated packrat parsers for the Java[1] programming language.
- Chapter 6 details relevant prior work. The chapter begins by reviewing the origins and background of TDPL, then provides detailed informal comparisons between packrat parsing and other algorithms such as LL/LR, noncanonical LR, and generalized CFG parsing.
- Finally, Chapter 7 concludes and points out possible directions for future work.

1.6 Code Availability

Complete versions of all of the examples and library code described in this thesis, as well as Pappy, the packrat parser generator, can be found at the following web site:

```
http://pdos.lcs.mit.edu/~baford/packrat/thesis/
```

[1] Java is a trademark of Sun Microsystems, Inc.

Chapter 2

Top-Down Parsing Language

The two most common methods of describing language syntax today are regular expressions and context-free grammars [2, 3]. These formalisms are by no means the only ways of precisely specifying the syntactic structure of a language, however. Another useful syntax description paradigm, known as *top-down parsing language* or TDPL, bears a close relationship to context-free grammars but also has fundamental differences. To provide a flavor of the relationship between these two paradigms, a context-free grammar is geared toward *writing* text in a specified language based on high-level, structured information, whereas a TDPL grammar is geared toward *reading* text strings and deriving structured information from them: hence the name "parsing language."

For the description of languages intended to be (mostly) written by humans but read by machines, TDPL is often a more suitable specification tool than a context-free grammar. Many common syntactic idioms in practical programming languages, such as longest-match rules, are more easily and naturally expressed in TDPL. In addition, TDPL grammars can always be parsed in linear time using the packrat parsing algorithm described in subsequent chapters, whereas only restricted subclasses of CFGs can be parsed in linear time. The rest of this chapter consists of a brief review of regular expressions and context-free grammars, followed by an in-depth exploration of the TDPL paradigm, and a comparison of its expressive power in comparison with conventional CFG notation.

2.1 Regular Expressions

Regular expressions provide a way of specifying simple languages whose structure may involve sequencing, choice, and repetition, but no recursive syntactic constructs. A regular expression can be built using the following rules:

- **Empty String:** '`()`' is a regular expression denoting the empty string. (In formal descriptions the empty string is often denoted as 'ϵ'.)

- **Literal:** If a is a single character, then 'a' is a regular expression denoting the corresponding single-character string.

- **Sequence:** If e_1, e_2, ..., e_n are regular expressions, then '$(e_1 e_2 \ldots e_n)$' is a regular expression denoting all strings that can be constructed by concatenating n strings, where the first substring matches e_1, the second substring matches e_2, etc.

- **Choice:** If e_1 and e_2 are regular expressions, then '$(e_1 \,|\, e_2)$' is a regular expression denoting all strings that match either e_1 or e_2 or both.
- **Repetition:** If e is a regular expression, then '$e*$' is a regular expression denoting all strings that can be constructed by concatenating any number of substrings together, where each substring matches the regular expression e.

Practical regular expression languages typically use precedence rules to dispense with unnecessary parentheses, and for convenience also usually support additional operators that can be implemented in terms of the primitives above, such as the following:

- **Optional:** If e is a regular expression, then '$e?$' is a regular expression denoting the empty string in addition to all strings matching e. '$e?$' is equivalent to '$(e\,|\,())$'.
- **One-or-more:** If e is a regular expression, then '$e+$' is a regular expression denoting all strings that can be constructed by concatenating *one or more* substrings together, each of which matches e. '$e+$' is equivalent to '$(ee*)$'.

A regular expression essentially provides simple, concise, rules for *writing* strings in the language denoted by the regular expression. For example, the regular expression '`(a(b|c)*)`' is a shorthand for the following "recipe" for generating strings consisting of an '`a`' followed by any number of '`b`'s and/or '`c`'s:

1. Process the subexpression '`a`' by generating the single character '`a`'.
2. To process the subexpression '`(b|c)*`':
   (a) Decide how many repetitions of the subexpression '`(b|c)`' to generate.
   (b) For each individual repetition:
      i. Choose which of the alternative subexpressions, '`b`' or '`c`', to generate
      ii. If the first alternative is chosen, generate the single character '`b`'. If the second alternative is chosen, generate the single character '`c`'.

If the strings in this trivial language were intended to convey useful information of some kind, then the various choices made in the recipe above would be determined by the content to be transmitted. For example, each of the '`b`' versus '`c`' choices might be intended to transmit one bit of a binary number, and the number of repetitions decided in step 2(a) might be determined by the number of bits in that binary number. But the important point is that the structure of regular expressions is defined fundamentally in terms of how strings are *produced* rather than how they are recognized or read.

There is no *a priori* guarantee that strings produced from a given regular expression will be unambiguous or readable in general. For example, the regular expression '`(a|a()*)`' is perfectly legitimate, but any information that may be used to direct the choice and repetition operators is lost in the generated string, since the expression can ultimately only yield the single string '`a`'.

Although recognizing strings defined by regular expressions is not nearly as straightforward as generating strings from them, the recognition problem is not that difficult and can always be done in linear time, proportional to the length of the input string. The relevant algorithms can be found in any compiler textbook [2]. Because of their conciseness,

$$\begin{array}{lcl}
\mathrm{E} & \rightarrow & \mathrm{N} \\
 & | & \text{`('}\ \mathrm{E}\ \text{`+'}\ \mathrm{E}\ \text{`)'} \\
 & | & \text{`('}\ \mathrm{E}\ \text{`-'}\ \mathrm{E}\ \text{`)'} \\
\mathrm{N} & \rightarrow & \mathrm{D} \\
 & | & \mathrm{D\ N} \\
\mathrm{D} & \rightarrow & \text{`0'} \mid \ldots \mid \text{`9'}
\end{array}$$

Figure 2-1: Context-free grammar for a trivial arithmetic expression language

simplicity, and efficiency, regular expressions have become the *de facto* basis for generic searching and text manipulation tools of all kinds, as well as for defining the lexical stage of many conventional language processors. Issues of ambiguity and information preservation are not usually considered highly important for regular expression processing, since in practice regular expressions are primarily used merely to *find* or *identify* strings matching a pattern for subsequent (manual or automatic) processing by other methods, and not to extract information from those strings directly.

2.2 Context-Free Grammars

The most important limitation of regular expressions is that they cannot express recursive syntactic constructs. The syntax of most practical languages inherently involves recursive rules such as, "If e_1 and e_2 are expressions, then 'e1 + e2' is an expression," or "If e is an expression and s_1 and s_2 are statements, then 'if (e) then s_1 else s_2' is a statement." Even the informal definition of regular expressions above uses such rules; regular expressions are not powerful enough even to express their own syntax effectively.

A *context-free grammar* (CFG) solves this problem by representing syntax not by a single rule but by a set of rules that can refer to each other recursively. Two classes of symbols are utilized in a CFG: *terminals*, which represent atomic syntactic elements and correspond to literal characters in regular expressions; and *nonterminals*, which represent higher-level, composite constructs such as expressions and statements. In the most common practical representation for context-free grammars, known as Backus-Naur Form (BNF), each rule has the form '$n \rightarrow v_1 | \ldots | v_n$', where n is the nonterminal symbol that the rule defines, and each v_i on the right-hand side is a string consisting of any number of terminals and/or nonterminals. Each v_i in the rule represents an alternative syntactic expansion or *derivation* for the nonterminal on the left. No significance is attached to the order in which multiple alternatives appear in a rule, or to the order in which rules appear in the grammar.

As with regular expressions, a context-free grammar essentially provides a recipe for writing strings in the language expressed by the grammar. For example, Figure 2-1 shows a context-free grammar for a trivial expression language featuring addition, subtraction, and decimal numbers. The grammar uses three nonterminal symbols: E represents expressions, N represents decimal numbers consisting of one or more digits, and D represents individual decimal digits. If we want to generate an expression (E), the grammar gives us three choices: either write a number using the rule for N, or write a string of the form '(e_1+e_2)' or '(e_1-e_2)', where e_1 and e_2 are expression strings likewise generated by the E rule. Similarly, to generate a number (N), there are two choices: either just write a single digit (the "base case"), or prepend a digit (D) to another number generated by this same rule.

Although BNF notation is often extended in practice with repetition operators such as '$*$' and '$+$', repetition is not usually considered "primitive" in context-free grammars since it can be easily expressed in terms of recursion, as demonstrated by the N nonterminal in the example grammar.

As with regular expressions, ambiguity and information preservation are issues for context-free grammars. Ambiguity is a much more important problem in practice for CFGs than for regular expressions, because the primary goal is usually not just to *recognize* whether a string conforms to a particular CFG, but to *parse* the string, effectively reconstructing the full set of decisions (that would have been) used to generate the string from the CFG.

Both recognizing and parsing CFGs is a much harder problem than recognizing regular expressions. Parsing arbitrary CFGs is equivalent in complexity to boolean matrix multiplication [13], which is generally believed to lie between $O(n^2)$ and $O(n^3)$. The fastest known algorithm to recognize arbitrary *unambiguous* CFGs is $O(n^2)$, but the question of whether an arbitrary CFG is unambiguous is undecidable in general [3].

Various linear-time algorithms exist to recognize useful but often restrictive classes of CFGs. Top-down, *predictive* parsing algorithms start with high-level syntactic constructs as "goals" and break them into progressively smaller subgoals while recognizing a string, whereas bottom-up, *shift/reduce* algorithms start with the atomic units (terminals) of the string and progressively clump or "reduce" them into higher-level constructs (nonterminals). These algorithms are explored and contrasted with packrat parsing later in Chapter 6.

2.3 TDPL and Parsing Grammars (PGs)

A different but equally legitimate approach to expressing language syntax is to represent it not as a set of rules for *writing* strings in the language, but as a set of rules for *reading* them. A *top-down parsing language* (TDPL) is one notation that can be used to express grammars in this way. As its name implies, TDPL is geared toward a top-down style of parsing: it can be seen as a simple programming language for writing top-down parsers. A grammar in TDPL or a similar "parsing-oriented" notation will be referred to here as a *top-down parsing grammar*, or just *parsing grammar* (PG) for short.

2.3.1 Definition of Parsing Grammars

As with CFGs, a parsing grammar makes use of both terminal and nonterminal symbols, and consists of a set of rules to provide definitions for each nonterminal. Each rule can refer to other rules in the grammar recursively. Parsing grammar definitions will be given the notation '$n \leftarrow e$', where n is a nonterminal and e is a *parsing expression* to be defined below. The use of a left-arrow instead of a right-arrow expresses the conceptual difference in "natural information flow" between a PG and a CFG. Whereas the rules of a CFG most directly represent "expansions" from nonterminals to right-hand-side strings, the rules of a PG most directly represent "reductions" from right-hand-side parsing expressions to nonterminals. Furthermore, whereas the expansions expressed in a CFG represent operations on *whole strings*, the reductions expressed in a TDPL grammar represent operations on *prefixes* of an input string.

For example, the CFG rule '$\mathrm{A} \rightarrow \mathrm{B}\ \mathrm{C}$' can be read as, "To produce a string matching nonterminal A, first generate a string matching B and a string matching C, and concatenate the two to form the result." The interpretation of the similar-looking PG rule 'A ← B C' is quite different: "To read an instance of nonterminal A from a string, look for an instance of nonterminal B followed by an instance of nonterminal C, possibly followed by additional, unrecognized input. If both B and C are found, then succeed and consume the corresponding portion of the input string. Otherwise, fail and do not consume anything." The subtle but important difference between these interpretations is elaborated below.

In the TDPL notation used in this thesis, parsing expressions are constructed as follows:

- **Empty String:** '`()`' is a parsing expression denoting the empty string. Its interpretation is, "Don't try to read anything: just trivially succeed without consuming any input."

- **Terminal:** If a is a terminal symbol (e.g., a single character), then 'a' is a parsing expression whose interpretation is, "If the next input terminal is a, then consume that one terminal and succeed. Otherwise, fail without consuming anything."

- **Nonterminal:** If A is a nonterminal symbol, then 'A' is a parsing expression whose interpretation is, "Try to read input according to the grammar rule for nonterminal A, and succeed or fail accordingly."

- **Sequence:** If e_1, e_2, ..., e_n are parsing expressions, then '$(e_1e_2 \ldots e_n)$' is a parsing expression whose interpretation is as follows: "First try to read a string matching e_1. If e_1 succeeds, then try to read a string matching e_2, starting with the remaining input text left unconsumed by e_1. If e_2 succeeds, then continue with e_3, and so on up to e_n. If all n expressions are successfully recognized consecutively, then succeed and consume all of the corresponding input text. If any sub-expression fails, then the sequence as a whole fails without consuming any text."

- **Ordered Choice:** If e_1, e_2, ..., e_n are parsing expressions, then '$(e_1/e_2/\ldots/e_n)$' is a parsing expression whose interpretation is as follows: "First try to read a string matching e_1. If that succeeds, then the choice expression succeeds and consumes the corresponding input text. Otherwise, try e_2 with the original input text, then e_3, and so on, in order up to e_n, stopping with the first matching alternative. If none of the n alternatives match, then fail without consuming any text." The parsing expression '(e_1/e_2)' can be read, "e_1 or else e_2." We use the forward slash symbol ('/') to denote choice in TDPL in place of the vertical bar symbol ('|') used in CFGs, in order to emphasize the critical difference that choice in a CFG is symmetric (order is not important), whereas choice in TDPL is asymmetric and implies a priority relationship.

- **Greedy Repetition:** If e is a parsing expression, then 'e*' is a parsing expression with this interpretation: "Apply expression e repeatedly to the input text, consuming input text progressively with each iteration for as long as it continues to succeed. At the first failure, consume all of the successfully matched text and succeed. If e did not match even once, then succeed anyway but do not consume anything."

- **Greedy Positive Repetition:** If e is a parsing expression, then 'e+' is a parsing expression with this interpretation: "Apply expression e repeatedly, then succeed and consume all matched text as long as at least one instance of e is recognized. If not, then fail without consuming anything."

```
E  ←  N
   /  '(' E '+' E ')'
   /  '(' E '-' E ')'
N  ←  D N
   /  D
D  ←  '0' | ... | '9'
```

Figure 2-2: Parsing grammar for a trivial arithmetic expression language

- **Optional:** If e is a parsing expression, then 'e?' is a parsing expression with this interpretation: "Try to apply expression e to the input. If it succeeds, then consume the matched text and succeed. If e fails, then succeed anyway but do not consume any input."

- **Followed-By Predicate:** If e is a parsing expression, then '&(e)' is a parsing expression with this interpretation: "Try to apply expression e to the input. If it matches, then succeed *but do not consume any input* (i.e., back up to the original position before e was applied). If e fails, then fail."

- **Not-Followed-By Predicate:** If e is a parsing expression, then '!(e)' is a parsing expression with this interpretation: "Try to apply expression e to the input. If it matches, then fail without consuming any input. If e fails, then succeed without consuming any input."

Despite the richness of this vocabulary, all of the TDPL constructs defined above can easily be reduced to a small "kernel" of primitive constructs. Since the most appropriate choice of primitives depends on the intended use of the language (e.g., for formal analysis versus practical parser implementation), this issue will be dealt with later.

2.3.2 An Example Parsing Grammar

Figure 2-2 shows a parsing grammar for the same trivial arithmetic expression language that the CFG in Figure 2-1 represents. The structure is essentially the same, except for the order of the alternatives for the N construct; this difference is important and will be explained shortly.

Figure 2-3 shows an illustration of how the string '(12-3)' can be read according to the parsing grammar in Figure 2-2. We start by trying to read an expression (E) starting from the beginning of the string. The first alternative for E is N, so we try both alternatives of N in succession at this position, but neither of them matches because the first character is an opening parenthesis and not a digit. Next we try the second alternative for E, the rule for addition expressions. This rule successfully matches the opening parenthesis, and directs us to read a (sub-)expression starting at position 2. To read this sub-expression, we again first try the N alternative. In this case, the first alternative of N successfully matches a digit at position 2, then recursively looks for a second instance of N at position 3. The first alternative of this second instance of N (D N) fails because the digit at position 3 is not followed by any more digits, but the second alternative (D) succeeds in matching that digit, producing the result labeled N_3 in the figure. This success result enables the earlier attempt

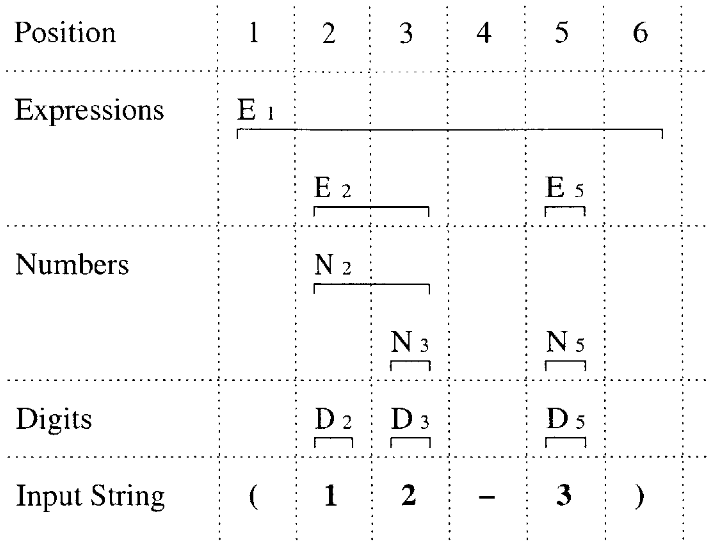

Figure 2-3: Reading the string '(12-3)' according to the grammar in Figure 2-2

to read N starting at position 2 to succeed, yielding in a two-character-wide instance of N at position 2, labeled N_2, and this result in turn leads to the corresponding two-character expression E_2. Returning to the attempt to read an expression at position 1, the second alternative now fails because expression E_2 is followed in the input by a '-' instead of a '+'. However, upon subsequently attempting the third alternative for E, this alternative succeeds because it matches the opening parenthesis and E_2 in the same way that the first alternative did, then continues to match the '-', the single-digit expression E_5 at position 5, and the closing parenthesis. Thus, expression E_1 is generated, which matches the entire input string.

2.3.3 Greedy versus Nondeterministic Repetition

The rule for the N nonterminal above illustrates one of the most important differences between a parsing grammar and a context-free grammar. In the CFG in Figure 2-1, the order of the two alternatives for N is unimportant because the choice between them is nondeterministic and is oriented toward writing strings rather than reading them. In the PG of Figure 2-2, the order of the two alternatives is critical: if we attempted the shorter alternative (D) first, then the longer one would *never* be used, because in a PG the first alternative to succeed is always the one used. The resulting grammar would not be able to read the example input string '(12-3)' because the reading of nonterminal N at position 2 would only consume the '1' in the number '12', leaving the '2' to befuddle subsequent attempts to parse a larger expression starting at position 1.

Because of this difference, repetition constructs in a parsing grammar are naturally "greedy" rather than nondeterministic: a repetitive construct by default always consumes as much text as it can, regardless of the context in which it is used. The nonterminal N in the example parsing grammar can be eliminated by replacing the first alternative of E with 'D+', but the result is exactly the same, which is why the + operator in a parsing grammar is known as "greedy positive repetition."

In practice, greedy repetition is almost always what we want when dealing with languages intended to be unambiguous and machine-readable. It is easy to create "pathological" examples of context-free grammars that are unambiguous but depend on nondeterministic choice, such as the following one:

$$\begin{array}{lcl} S & \rightarrow & A\ a \\ A & \rightarrow & a\ A\ |\ a \end{array}$$

If this CFG is converted directly to a PG in the obvious way, by reversing the arrows and changing the '|' to a '/', then the resulting grammar will not match any string at all. The nonterminal A in the PG will always consume as many consecutive 'a's as it can and cause the subsequent attempt by S to match a trailing 'a' to fail. But even in pathological cases such as these, usually a PG can be written to achieve the desired effect. For example, the following PG is equivalent to the above CFG:

$$\begin{array}{lcl} S & \leftarrow & A\ a \\ A & \leftarrow & a\ A\ \&(a)\ |\ a \end{array}$$

The use of the "followed-by" operator in the first alternative for A ensures that exactly one 'a' is left unconsumed following the text matched by A, so that S will be able to succeed.

2.3.4 Parsing Complete Strings

If the grammar in Figure 2-2 is used to read the string '(12-3)XYZ' starting with the nonterminal E, then the result is a success, but only the '(12-3)' part of the string is actually read, leaving 'XYZ' as an "unconsumed" remainder. When the intention is to read an input string of a known length, it is usually desirable that the parsing process "succeed" only if the *entire* input is accepted, not just part of it. Fortunately, this behavior is easy to accomplish by adding a special "start symbol" S, as follows:

$$\begin{array}{lcl} S & \leftarrow & E\ !(C) \\ E & \leftarrow & \ldots \\ \ldots & & \\ C & \leftarrow & \text{any single character} \end{array}$$

The start symbol looks for an expression E, then uses the "not-followed-by" operator to ensure that nothing follows the recognized expression in the input. If there is more text following the E, then the C will match, causing S to fail; otherwise C fails and S succeeds.

C in this example is a nonterminal representing a *character class*. As long as the character set in use is finite, nonterminals representing character classes can in theory be written out directly as a list of individual alternatives. Particularly with the shift from ASCII and its 8-bit relatives to Unicode, however, it is usually easier to consider character classes to be "primitive." More specialized character classes typically include letters, digits, whitespace characters, control characters, punctuation, and symbols.

2.3.5 Left Recursion

In a context-free grammar, both *left recursion* and *right recursion* are allowed. A left-recursive nonterminal is one that, after being expanded one or more times, can yield a string starting with the same nonterminal. Similarly, a right-recursive nonterminal is one that can expand to a string *ending* in the same nonterminal. For example, it is typical to express the syntax of left-associative operators in terms of left-recursive CFG definitions, and right-associative operators in terms of right-recursive definitions:

```
Unary      →  '+' Unary
           |  '–' Unary
           |  Primary
Additive   →  Additive '+' Unary
           |  Additive '–' Unary
           |  Unary
```

In the right-recursive definition of Unary, the '+' and '–' symbols serve as right-associative unary operators (e.g., sign flags under the standard interpretation). The right-recursive self-reference of Unary from its own definition allows multiple unary operators to precede a Primary expression. For example, if p is a Primary expression, then the string '+–p' is a Unary '+' expression containing a nested Unary '–' expression: i.e., '+[–p]', In contrast, the definition for Additive is left-recursive, with the result that successive expansions of this nonterminal will build toward the *right* from a leading Unary expression. For example, if e_1, e_2, and e_3 are Unary expressions, then 'e_1+e_2–e_3' can be naturally interpreted as a binary '–' expression whose left-hand operand is a nested binary '+' expression: i.e., '[a+b]+c'.

In TDPL, while right recursion works in much the same way as it does in a CFG, a left-recursive definition is considered erroneous, because its interpretation under TDPL rules leads to a degenerate self-reference. For example, changing the definition of Additive above directly into a TDPL definition in the obvious way would lead to this interpretation: "To read an Additive expression, first try to read an Additive expression followed by ..." There would be no way to read an Additive expression at all, because doing so would require already having read an Additive expression starting at that same position. Recursive nonterminals in TDPL function properly only if some "progress" is made rightwards through the input string before the next recursive invocation of the same nonterminal. For example, the definition of Unary above can be converted directly into a valid right-recursive TDPL definition whose interpretation is as follows: "To read a Unary expression, first try to read a '+' sign, then read another Unary expression *following the '+' sign...*"

In a CFG, left recursion can be convenient but is not essential to the expression of language syntax, because any CFG involving left recursion can be rewritten into a CFG representing the same language without left recursion. In TDPL, it is usually more convenient and concise to use the repetition operators '∗' and '+' instead of either left or right recursion. For example, the above CFG can be written in TDPL as follows:

```
Unary     ←  ('+' / –')* Primary
Additive  ←  Unary (('+' / '–') Unary)*
```

2.4 The Expressiveness of CFGs Versus PGs

For the purpose of expressing languages that are intended to be unambiguous and machine-readable, parsing grammars are both more powerful and more convenient than CFGs in practice. Part of this expressiveness arises from TDPL's natural support for reading text with a longest-match ("greedy") policy, which is pervasively used in practical languages. Another important part of this expressiveness is provided by the '&' and '!' operators demonstrated above. These operators represent *syntactic predicates* [16], which allow arbitrary syntactic patterns to be used to direct decisions without actually consuming any input text. The following subsections provide a few practical examples of the expressive power of PGs in comparison with CFGs.

2.4.1 Tokens and Whitespace

In most programming languages, the lowest "level" of syntax apart from atomic characters themselves is the level of *tokens.* A token corresponds to a string of one or more *directly consecutive* characters, such as a word or a decimal number. Tokens may be separated by *whitespace* characters such as spaces, tabs, and newlines, but whitespace padding never appears *within* a token. Often tokens *must* be separated by whitespace characters in order to make them readable. For example, the two consecutive words '`bar`' and '`fly`' must be separated by at least one whitespace character in order for them to be distinguishable from the single word '`barfly`'.

Despite its apparent simplicity and obviousness, standard tokenization rules are difficult to express in a pure context-free grammar. For this reason, CFGs are usually not used to express language syntax at the character level, but instead only down to the token level. Converting strings of characters into strings of tokens is usually left for a separate preprocessing phase before parsing, known as *lexical analysis.* This difficulty is also the primary historical reason the atomic symbols used in a context-free grammar are often referred to as "tokens" or "terminals" rather than "characters."

Consider a simple "S-expression" language, in which an expression is either an identifier or a parenthesized sequence of consecutive expressions. If identifiers (IDENT) and parentheses (OPEN and CLOSE) are assumed to be atomic terminals produced by a separate lexical analysis phase, then this language is easily expressed as a CFG:

$$
\begin{array}{lcl}
\text{Expr} & \rightarrow & \text{IDENT} \\
 & | & \text{OPEN Exprs CLOSE} \\
\text{Exprs} & \rightarrow & \textit{(empty)} \\
 & | & \text{Expr Exprs}
\end{array}
$$

Suppose we want to express tokenization directly in the CFG however. According to standard rules, identifiers cannot contain parentheses, so whitespace padding is optional between two parentheses or between a parenthesis and an identifier. Two consecutive identifiers, however, must be separated by at least one whitespace character. To express these tokenization rules, the original S-expression grammar must be "unfolded" in some way so that these contexts can be distinguished. For example, here is one way the grammar can be rewritten:

$$
\begin{array}{lcl}
\text{Expr} & \rightarrow & \text{Identifier SpacesOpt} \\
 & | & \text{`(' SpacesOpt Exprs `)' SpacesOpt} \\
\text{Exprs} & \rightarrow & \text{ParenExprs} \\
 & | & \text{IdentExprs} \\
\text{ParenExprs} & \rightarrow & \textit{(empty)} \\
 & | & \text{`(' SpacesOpt Exprs `)' SpacesOpt Exprs} \\
\text{IdentExprs} & \rightarrow & \text{Identifier SpacesOpt ParenExprs} \\
 & | & \text{Identifier Spaces IdentExprs} \\
\text{Identifier} & \rightarrow & \text{Letter} \\
 & | & \text{Letter Identifier} \\
\text{SpacesOpt} & \rightarrow & \textit{(empty)} \\
 & | & \text{Spaces} \\
\text{Spaces} & \rightarrow & \text{Space} \\
 & | & \text{Space Spaces}
\end{array}
$$

In this example, Letter and Space are assumed to denote character classes representing identifier-letters and whitespace characters, respectively. In order to isolate the case in which two identifiers appear consecutively, the Exprs nonterminal has been factored into two other nonterminals, one (ParenExprs) for empty expression lists and expression lists starting with a parenthesized subexpression, and the other for expression lists starting with an identifier. The rule for IdentExprs can then make whitespace mandatory between an Identifier and a nested IdentExprs "tail," and optional between an Identifier and a ParenExprs "tail."

Even after this unfolding, however, the tokenization rules are still expressed only implicitly and in a rather cumbersome way, through the uses of the Spaces and SpacesOpt nonterminals sprinkled throughout the grammar. Furthermore, in the process of unfolding the grammar, the simplicity, elegance, and clarity of the original grammar has been obscured considerably. In a language even marginally more complex, expressing tokenization rules implicitly in this way quickly becomes impractical.

In contrast, because of its natural predisposition for longest-match repetition, tokenization can easily be expressed in a natural and modular fashion in a parsing grammar. Here is a PG for the S-expression language above with integrated tokenization:

```
Expr        ←  Identifier
            /  Open Expr* Close
Identifier  ←  Letter+ Space*
Open        ←  ’(’ Space*
Close       ←  ’)’ Space*
```

The Identifier nonterminal, for example, has a very simple and natural informal interpretation: "To parse an identifier, read as many consecutive letters as possible, ensuring that at least one is present, followed by as many consecutive space characters as possible." The implicit longest-match policy directly yields the desired behavior. Rules for any number of other kinds of tokens can be expressed similarly.

2.4.2 Reserved Words

The possibility of expressing the syntax of a conventional programming language "top-to-bottom" in a context-free grammar becomes even more remote if the language includes *reserved words*, as most programming languages do. A reserved word is a syntactic construct that takes the basic form of an identifier, typically consisting of a fixed sequence of consecutive letters, but which serves some completely different purpose in the higher-level syntax. Reserved words generally serve as syntactic markers used to introduce or delineate specific kinds of statements, definitions, or other programming language constructs. In order to serve as syntactic markers reliably, it is usually critical that they never be mistaken for identifiers. Therefore, the nonterminal in a syntax representing identifiers must *not* match reserved words: reserved words must actually be "reserved" syntactically from the space of identifiers.

Unfortunately, a pure CFG provides no direct way to express the notion that we want a nonterminal such as Identifier to represent all strings in some large class *except for* a few specific strings. Instead the definition for Identifier would have to be unfolded so as to "feel around the edges" of the set of reserved words. For example, if a language has only a single reserved word, '`foo`', then the CFG rules for recognizing these tokens could be written as follows:

FOO	→	'foo'
Identifier	→	NotF LettersOpt
	|	'f' NotO LettersOpt
	|	'f' 'o' NotO LettersOpt
	|	'f' 'o' 'o' Letter LettersOpt
LettersOpt	→	(empty)
	|	Letter LettersOpt
NotF	→	any letter except 'f'
NotO	→	any letter except 'o'

This approach is obviously too cumbersome to be practical, especially as the number of reserved words increases.

In contrast, reserved words are easy to express in a parsing grammar using the "not-followed-by" syntactic predicate operator. For example, a parsing grammar might use the following rules to express the tokenization of identifiers and reserved words, including the handling of trailing whitespace:

Identifier	←	!(ReservedWord) Letter+ Space*
ReservedWord	←	IF / ELSE / WHILE / DO / ...
IF	←	'if' !(Letter) Space*
ELSE	←	'else' !(Letter) Space*
WHILE	←	'while' !(Letter) Space*
DO	←	'do' !(Letter) Space*
...		

The definition of Identifier can be read as follows: "To read an Identifier, first check that there is no ReservedWord at this position, and if there is, fail. Otherwise, read one or more letters followed by any number of spaces." Each reserved word has its own corresponding nonterminal, named in all capitals by historical convention. The nonterminal for a reserved word first matches the appropriate string, then checks that this word is not followed by any more letters (since the recognized sequence could be the first part of a longer identifier such as '`iffy`'), and finally consumes any whitespace padding following the reserved word.

2.4.3 Operators with Variable Precedence

Although the examples above have focused on tokenization issues, parsing grammars are often more convenient than CFGs for expressing practical higher-level language constructs as well. For example, in C-like languages, there is a classic ambiguity relating to the `if` statement, which allows an optional `else` clause:

Statement	→	IF '(' Expression ')' Statement
	|	IF '(' Expression ')' Statement ELSE Statement
	|	...

When reading a statement of the form, '`if (`e_1`) if (`e_2`)` s_1 `else` s_2', the above CFG leaves it undecided whether the final '`else` s_2' part of the statement is associated with the outer `if` or the inner one. The Haskell language [12] includes more severe examples of the same kind of ambiguity, because of its '\' (lambda), '`let`', and '`if`' operators, which have a high precedence themselves but accept a low-precedence expression as a "tail," unguarded by any syntactic marker to indicate the end of this tail:

$$\begin{array}{lcl}
\text{Exp} & \rightarrow & \text{low-precedence expressions} \\
 & \vdots & \\
\text{Exp10} & \rightarrow & \text{`}\texttt{\textbackslash}\text{'} \ldots \text{`}\texttt{->}\text{' Exp} \\
 & | & \text{`}\texttt{let}\text{'} \ldots \text{`}\texttt{in}\text{' Exp} \\
 & | & \text{`}\texttt{if}\text{' Exp `}\texttt{then}\text{' Exp `}\texttt{else}\text{' Exp} \\
 & | & \text{other high-precedence expressions}
\end{array}$$

The above CFG does not specify whether a string of the form ‘`let ... in x + 1`’ should be interpreted as ‘`let ... in (x + 1)`’ or as ‘`(let ... in x) + 1`’.

In essentially all practical situations of this kind, the desired behavior is for the “dangling tail” to bind to the *innermost* possible construct. For example, the dangling-`else` example above should be resolved ‘$\texttt{if (}e_1\texttt{) \{ if (}e_2\texttt{) }s_1\texttt{ else }s_2\texttt{ \}}$’, and the Haskell example should be resolved ‘`let ... in (x + 1)`’. This rule makes some intuitive sense because the innermost construct is the one closest to the dangling tail, and thus is probably the one the programmer had been thinking about “most recently” when the dangling tail was written. This “innermost-binding” rule does cause occasional confusion in practice, but it is well-accepted and its use can provide substantial notational flexibility.

This disambiguation rule is just a higher-level variant of the longest-match policy that is needed to parse tokens and whitespace effectively, and for this reason this policy is extremely natural in a PG. Both of the above examples, if converted directly to a PG in the obvious way, yield the desired behavior.

In the earlier dangling-`else` case it is possible to rewrite the original CFG to resolve the ambiguity according to the innermost-binding rule, by unfolding and duplicating the Statement nonterminal. In the Haskell case, however, the number of operators contributing to the ambiguity, and the large difference between the precedences of the relevant operators and the precedences of their tail expressions, make the prospect of disambiguating the CFG in this way inconceivable in practice. As a result, the Haskell specification, as with the specifications of most other programming languages in similar situations, simply provide an ambiguous CFG and an informal side-note specifying how a *parser* for the language should resolve the ambiguity. In essence, the language specification is “stepping across the gap” from the CFG paradigm of describing how phrases are written, into the TDPL paradigm of describing how phrases are read. The frequency with which this gap must be crossed in practice might be taken as a compelling argument for starting on the TDPL side in the first place.

2.4.4 Limitations of Parsing Grammars

The primary limitation of the TDPL notation and parsing grammars is that they *cannot* express ambiguous syntax even if there is a need to. For expressing natural languages, for example, in which ambiguity is a fact of life, context-free grammars are clearly more appropriate because they enable a parser to enumerate *all of the possible ways* of reading a string. In natural language, there are usually no simple, purely local syntactic rules like the innermost-binding rule above to resolve various ambiguities consistently. Instead, a language recognizer must rely on global syntactic structure or, most often, on semantics (i.e., which readings make “sense”).

If attention is restricted to unambiguous context-free grammars, there are pathological CFGs such as the following, for which there appear to be no PG recognizing the same language:

$$S \rightarrow a\,S\,a \mid a\,S\,b \mid b\,S\,a \mid b\,S\,b \mid a$$

However, pathologies of this sort seem rather unlikely to occur in practical programming languages designed to be legible to humans as well as machines.

It might be argued that specifying grammars via CFGs is useful because automatic tools can then check the grammar for unintended ambiguities. Checking grammars in this way is indeed a worthwhile goal, but in current practice it is undermined by several practical problems. First, since it is undecidable whether an arbitrary CFG is unambiguous, these tools at best check that the grammar is within some conservative class of grammars that is known to be "safe," such as LALR(1). Second, the grammars for useful programming languages often have so many *intended* ambiguities that any unintended ones are easily overlooked amid the warnings. At best you might be lucky to notice that the number of shift/reduce conflicts reported by the parser generator has increased since the last compile.

It should be possible to extend TDPL notation with an "unordered choice" operator, which expresses the intention that all of the alternatives are supposed to be unambiguously differentiable from each other (i.e., non-overlapping). This "claim" could then be checked automatically using methods similar to those used by tools for CFGs, and then reduced to simple ordered choice once the claim has been verified. This approach would have the notable benefit of still allowing the ordered choice operator to be used in parts of the grammar in which ambiguity is expected and the desire is to resolve it with an explicit priority relationship. Although an unordered choice operator with this interpretation is easy to add to TDPL, appropriate algorithms to check for overlap between alternatives in a PG are not as obvious, and so this extension is left for future work.

Chapter 3

Packrat Parsing

In this chapter we address the problem of efficiently parsing strings whose syntax is specified in TDPL. We will use the programming language Haskell [12] as the implementation language, because it is currently the most popular and well-established of the non-strict functional programming languages. As we shall see, the lazy evaluation capabilities of non-strict languages such as Haskell makes the implementation of packrat parsers direct and elegant. Of course, there is no reason a packrat parser could not or should not be implemented in other languages; doing so would merely be slightly less straightforward because of the need to hand-code the lazy evaluation functionality that Haskell provides as part of the language.

In the first section of this chapter, we describe the basic principles behind the packrat parsing algorithm by first creating a backtracking recursive-descent for a trivial language, then converting that parser into a packrat parser. In the second section, we present useful extensions to this basic parser to demonstrate practical methods of handling of left recursion, integrated lexical analysis, constructing parsers with monadic combinators, and user-friendly error handling. The third section explores some practical issues and limitations of packrat parsers: localized backtracking, limited state, and high space consumption.

3.1 Building a Packrat Parser

Packrat parsing is essentially a top-down parsing strategy, and as such packrat parsers are closely related to recursive descent parsers. For this reason, we will first build a recursive descent parser for a trivial language and then convert it into a packrat parser.

Additive	←	Multitive ‘+’ Additive / Multitive
Multitive	←	Primary ‘*’ Multitive / Primary
Primary	←	‘(’ Additive ‘)’ / Decimal
Decimal	←	‘0’ / . . . / ‘9’

Figure 3-1: Grammar for a trivial language

3.1.1 Recursive-Descent Parsing

Consider the TDPL grammar shown in Figure 3-1 for a trivial arithmetic expression language. This grammar can be viewed as a direct short-hand representation for a recursive-descent parser with packtracking. To construct a recursive-descent parser for this grammar, we define four functions, one for each of the nonterminals in the grammar. Each function takes takes the string to be parsed, attempts to recognize some prefix of the input string as a derivation of the corresponding nonterminal, and returns either a "success" or "failure" result. On success, the function returns the remainder of the input string immediately following the part that was recognized, along with some semantic value computed from the recognized part. Each function can recursively call itself and the other functions in order to recognize the nonterminals appearing on the right-hand sides of its corresponding grammar rules.

Since the purpose of a practical parser is usually not just to recognize a string but to derive meaningful semantic information from its structure, the example parser to be developed here will also act as a simple calculator by computing the integer value of an arithmetic expression it recognizes.

To implement this recursive-descent parser in Haskell, we first need a type describing the result of a parsing function:

```
data Result v = Parsed v String
              | NoParse
```

In order to make this type generic for different parsing functions producing different kinds of semantic values, the Result type takes a type parameter v representing the type of the associated semantic value. A success result is built with the Parsed constructor and contains a semantic value (of type v) and the remainder of the input text (of type String). A failure result is represented by the simple value NoParse. In this particular parser, each of the four parsing functions takes a String and produces a Result with a semantic value of type Int:

```
pAdditive   :: String -> Result Int
pMultitive  :: String -> Result Int
pPrimary    :: String -> Result Int
pDecimal    :: String -> Result Int
```

The definitions of these functions have the following general structure, directly reflecting the mutual recursion expressed by the grammar in Figure 3-1:

```
pAdditive  s = ... (calls itself and pMultitive) ...
pMultitive s = ... (calls itself and pPrimary) ...
pPrimary   s = ... (calls pAdditive and pDecimal) ...
pDecimal   s = ...
```

For example, the pAdditive function can be coded as follows, using only primitive Haskell pattern matching constructs:

```
-- Parse an additive-precedence expression
pAdditive :: String -> Result Int
pAdditive s = alt1 where

    -- Additive <- Multitive '+' Additive
    alt1 = case pMultitive s of
             Parsed vleft s' ->
               case s' of
                 ('+':s'') ->
                   case pAdditive s'' of
                     Parsed vright s''' ->
                       Parsed (vleft + vright) s'''
                     _ -> alt2
                 _ -> alt2
             _ -> alt2

    -- Additive <- Multitive
    alt2 = case pMultitive s of
             Parsed v s' -> Parsed v s'
             NoParse -> NoParse
```

To compute the result of pAdditive, we first compute the value of alt1, representing the first alternative for this grammar rule. This alternative in turn calls pMultitive to recognize a multiplicative-precedence expression. If pMultitive succeeds, it returns the semantic value vleft of that expression and the remaining input s' following the recognized portion of input. We then check for a '+' operator at position s', which if successful produces the string s'' representing the remaining input after the '+' operator. Finally, we recursively call pAdditive itself to recognize another additive-precedence expression at position s'', which if successful yields the right-hand-side result vright and the final remainder string s'''. If *all three* of these matches were successful, then we return as the result of the initial call to pAdditive the semantic value of the addition, vleft + vright, along with the final remainder string s'''. If any of these matches failed, we fall back on alt2, the second alternative, which merely attempts to recognize a single multiplicative-precedence expression at the original input position s and returns that result verbatim, whether success or failure.

The other three parsing functions are constructed similarly, in direct correspondence with the grammar. Of course, there are easier and more concise ways to write these parsing functions, using an appropriate library of helper functions or combinators. These techniques will be discussed later in Section 3.2.3, but for clarity we will stick to simple pattern matching for now.

3.1.2 Backtracking Versus Prediction

The parser developed above is a *backtracking* parser. If alt1 in the pAdditive function fails, for example, then the parser effectively "backtracks" to the initial position, starting over with the original input string s in the second alternative alt2, regardless of whether the first alternative failed to match during its first, second, or third stage. If the input s consists of only a single multiplicative expression, then the pMultitive function will be called twice on the same string: once in the first alternative, which will fail while trying to match a nonexistent '+' operator, and then again while successfully applying the second alternative. This backtracking and redundant evaluation of parsing functions can lead to parse times that grow exponentially with the size of the input, and this blow-up is the principal reason why a "naïve" backtracking strategy such as the one above is never used in realistic parsers for inputs of substantial size.

The standard strategy for making top-down parsers practical is to design them so that they can "predict" which of several alternative rules to apply *before* actually making any recursive calls. In this way it can be guaranteed that parsing functions are never called redundantly and that any input can be parsed in linear time. For example, although the grammar in Figure 3-1 is not directly suitable for a predictive parser, it can be converted into the TDPL equivalent of an LL(1) grammar, suitable for prediction with one lookahead token, by "left-factoring" the Additive and Multitive nonterminals as follows:

Additive	$\leftarrow$	Multitive AdditiveSuffix
AdditiveSuffix	$\leftarrow$	'+' Additive / ()
Multitive	$\leftarrow$	Primary MultitiveSuffix
MultitiveSuffix	$\leftarrow$	'*' Multitive / ()

Now the decision between the two alternatives for AdditiveSuffix can be made before making any recursive calls simply by checking whether the next input character is a '+'. However, because the prediction mechanism only has "raw" input tokens (characters in this case) to work with, and must itself operate in constant time, only a restricted class of grammars can be parsed predictively. Care must also be taken to keep the prediction mechanism consistent with the grammar, which can be difficult to do manually and highly sensitive to global properties of the language. For example, the prediction mechanism for MultitiveSuffix would have to be adjusted if a higher-precedence exponentiation operator '**' was added to the language; otherwise the exponentiation operator would falsely trigger the predictor for multiplication expressions and cause the parser to fail on valid input.

Some top-down parsers use prediction for most decisions but fall back on full backtracking when more flexibility is needed. This strategy often yields a good combination of flexibility and performance in practice, but it still suffers the additional complexity of prediction, and it requires the parser designer to be intimately aware of where prediction can be used and when backtracking is required.

3.1.3 Tabular Top-Down Parsing

As pointed out by Birman and Ullman [4], a backtracking top-down parser of the kind presented in Section 3.1.1 can be made to operate in linear time without the added complexity or constraints of prediction. The basic reason the backtracking parser can take super-linear time is because of redundant calls to the same parsing function on the same input substring, and these redundant calls can be eliminated through memoization.

Each parsing function in the example is dependent *only* on its single parameter, the input string. Whenever a parsing function makes a recursive call to itself or to another parsing function, it always supplies either *the same* input string it was given (e.g., for the call by `pAdditive` to `pMultitive`), or a *suffix* of the original input string (e.g., for the recursive call by `pAdditive` to itself after matching a '+' operator). If the input string is of length n, then there are only $n + 1$ distinct suffixes that might be used in these recursive calls, counting the original input string itself and the empty string. Since there are only four

column	C1	C2	C3	C4	C5	C6	C7	C8
pAdditive			⇡	(7,C7)	X	(4,C7)	X	X
pMultitive		⇠		(3,C5)	X	(4,C7)	X	X
pPrimary			?	(3,C5)	X	(4,C7)	X	X
pDecimal			X	(3,C5)	X	(4,C7)	X	X
input	'2'	'*'	'('	'3'	'+'	'4'	')'	(end)

Figure 3-2: Matrix of parsing results for string '2*(3+4)'

parsing functions, there are at most $4(n+1)$ distinct intermediate results that the parsing process might require.

We can avoid computing any of these intermediate results multiple times by storing them in a table. This table has one row for each of the four parsing functions and one column for each distinct position in the input string. We fill the table with the results of each parsing function for each input position, starting at the *right* end of the input string and working towards the left, column by column. Within each column, we start from the bottommost cell and work upwards. By the time we compute the result for a given cell, the results of all would-be recursive calls in the corresponding parsing function will have already computed and recorded elsewhere in the table; we merely need to look up and use the appropriate results.

Figure 3-2 illustrates a partially-completed result table for the input string '2*(3+4)'. For brevity, Parsed results are indicated as (v,c), where v is the semantic value and c is the column number at which the associated remainder suffix begins. Columns are labeled C1, C2, and so on, to avoid confusion with the integer semantic values. NoParse results are indicated with an X in the cell. The next cell to be filled is the one for pPrimary at column C3, indicated with a circled question mark.

The rule for Primary expressions has two alternatives: a parenthesized Additive expression or a Decimal digit. If we try the alternatives in the order expressed in the grammar, pPrimary will first check for a parenthesized Additive expression. To do so, pPrimary first attempts to match an opening '(' in column C3, which succeeds and yields as its remainder string the input suffix starting at column C4, namely '3+4)'. In the simple recursive-descent parser pPrimary would now recursively call pAdditive on this remainder string. However, because we have the table we can simply look up the result for pAdditive at column C4 in the table, which is (7,C7). This entry indicates a semantic value of 7—the result of the addition expression '3+4'—and a remainder suffix of ')' starting in column C7. Since this match is a success, pPrimary finally attempts to match the closing parenthesis at position C7, which succeeds and yields the empty string C8 as the remainder. The result entered for pPrimary at column C3 is thus (7,C8).

Although for a long input string and a complex grammar this result table may be large, it grows only in proportion to the size of the input string. As long as the computation of each cell looks up only a limited number of previously-recorded cells in the matrix and completes in constant time, the parsing process as a whole completes in linear time.

Due to the "forward pointers" embedded in the results table, the computation of a given result may examine cells that are widely spaced in the matrix. For example, computing the result for pPrimary at C3 above made use of results from columns C3, C4, and C7. This ability to skip ahead arbitrary distances while making parsing decisions is the source of the algorithm's unlimited lookahead capability, making the algorithm more powerful than linear-time predictive parsers or LR parsers.

3.1.4 Packrat Parsing

An obvious practical problem with the tabular right-to-left parsing algorithm above is that it computes many results that are never needed. An additional inconvenience is that we must carefully determine the order in which the results for a particular column are computed, so that parsing functions such as pAdditive and pMultitive that depend on other results from the same column will work correctly.

Packrat parsing is essentially a lazy version of the tabular algorithm above that solves both of these problems. A packrat parser computes results only as they are needed, in the same order as the original recursive descent parser would. However, once a result is computed for the first time, it is stored for future use by subsequent calls.

A non-strict functional programming language such as Haskell provides an ideal implementation platform for a packrat parser. In fact, packrat parsing in Haskell is particularly straightforward because it does not require arrays or any other explicit lookup structures other than the language's ordinary algebraic data types.

Types for Packrat Parsing

First we will need a new type to represent a single column of the parsing result matrix, which we will call Derivs ("derivations"). This type is merely a tuple with one component for each nonterminal in the grammar. Each component's type is the result type of the corresponding parsing function. The Derivs type also contains one additional component, which we will call dvChar, to represent "raw" characters of the input string as if they were themselves the results of some parsing function. The Derivs type for our example parser can be conveniently declared in Haskell as follows:

```
data Derivs = Derivs {
                dvAdditive  :: Result Int,
                dvMultitive :: Result Int,
                dvPrimary   :: Result Int,
                dvDecimal   :: Result Int,
                dvChar      :: Result Char}
```

This Haskell syntax declares the type Derivs to have a single constructor, also named Derivs, with five components of the specified types. The declaration also automatically creates a corresponding data-accessor function for each component: dvAdditive can be used as a function of type Derivs → Result Int, which extracts the first component of a Derivs tuple, and so on.

Next we modify the Result type so that the "remainder" component of a success result is not a plain String, but is instead an instance of Derivs:

```
data Result v = Parsed v Derivs
              | NoParse
```

The Derivs and Result types are now mutually recursive: the success results in one Derivs instance act as links to other Derivs instances. These result values in fact provide the *only* linkage we need between different columns in the matrix of parsing results.

Packrat Parsing Functions

Now we modify the original recursive-descent parsing functions so that each function takes a Derivs instead of a String as its parameter:

```
pAdditive  :: Derivs -> Result Int
pMultitive :: Derivs -> Result Int
pPrimary   :: Derivs -> Result Int
pDecimal   :: Derivs -> Result Int
```

Wherever one of the original parsing functions examined input characters directly, the new parsing function instead refers to the dvChar component of the Derivs object. Wherever one of the original functions made a recursive call to itself or another parsing function, in order to match a nonterminal in the grammar, the new parsing function instead instead uses the Derivs accessor function corresponding to that nonterminal. Sequences of terminals and nonterminals are matched by following chains of success results through multiple Derivs instances. For example, the new pAdditive function uses the dvMultitive, dvChar, and dvAdditive accessors as follows, without making any direct recursive calls:

```
-- Parse an additive-precedence expression
pAdditive :: Derivs -> Result Int
pAdditive d = alt1 where

    -- Additive <- Multitive '+' Additive
    alt1 = case dvMultitive d of
             Parsed vleft d' ->
               case dvChar d' of
                 Parsed '+' d'' ->
                   case dvAdditive d'' of
                     Parsed vright d''' ->
                       Parsed (vleft + vright) d'''
                     _ -> alt2
                 _ -> alt2
             _ -> alt2

    -- Additive <- Multitive
    alt2 = dvMultitive d
```

The Recursive Tie-Up Function

Finally, we create a special "top-level" function, parse, to produce instances of the Derivs type and "tie up" the recursion between all of the individual parsing functions:

```
-- Create a result matrix for an input string
parse :: String -> Derivs
parse s = d where
    d    = Derivs add mult prim dec chr
    add  = pAdditive d
    mult = pMultitive d
    prim = pPrimary d
    dec  = pDecimal d
    chr  = case s of
             (c:s') -> Parsed c (parse s')
             [] -> NoParse
```

The "magic" of the packrat parser is in this doubly-recursive function. The first level of recursion is produced by the parse function's reference to itself within the case statement. This relatively conventional form of recursion is used to iterate over the input string one character at a time, producing one Derivs instance for each input position. The final Derivs instance, representing the empty string, is assigned a dvChar result of NoParse, which effectively terminates the list of columns in the result matrix.

The second level of recursion is via the symbol d. This identifier names the Derivs instance to be constructed and returned by the parse function, but it is also the parameter to each of the individual parsing functions. These parsing functions, in turn, produce the rest of the components forming this very Derivs object.

This form of *data recursion* works only in a non-strict programming language, which allows some components of an object to be accessed before other parts of the same object are available. In any Derivs instance created by the above function, for example, the dvChar component can be accessed before any of the other components of the tuple are available. Attempting to access the dvDecimal component of this tuple will cause pDecimal to be invoked, which in turn uses the dvChar component but does not require any of the other "higher-level" components. Accessing the dvPrimary component will similarly invoke pPrimary, which may access dvChar and dvAdditive. Although in the latter case pPrimary is accessing a "higher-level" component, doing so does not create a cyclic dependency in this case because it only ever invokes dvAdditive on a *different* Derivs object from the one it was called with: namely the one for the position following the opening parenthesis. Every component of every Derivs object produced by parse can be lazily evaluated in this fashion.

Data and Evaluation in a Packrat Parser

Figure 3-3 illustrates the data structure produced by the parser for the example input text '2*(3+4)', as it would appear in memory under a modern functional evaluator after fully reducing every cell. Each vertical column represents a Derivs instance with its five Result components. For results of the form 'Parsed *v* *d*', the semantic value *v* is shown in the appropriate cell, along with an arrow representing the "remainder" pointer leading to another Derivs instance in the matrix. In any modern lazy language implementation that

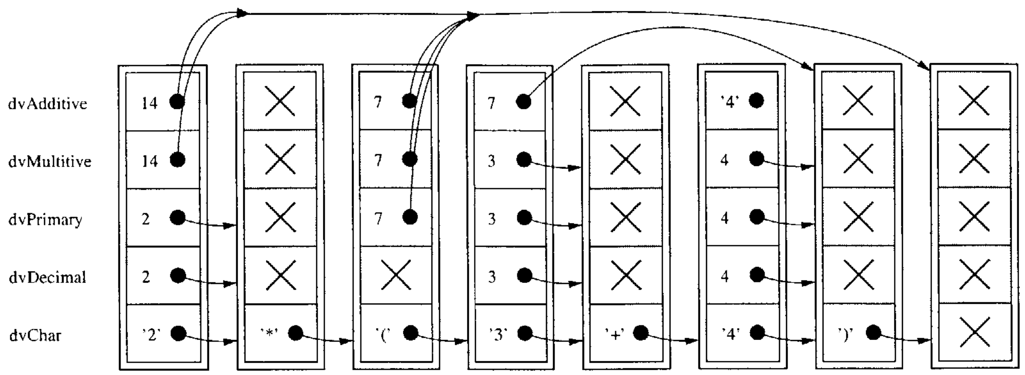

Figure 3-3: `Derivs` data structure produced by parsing the string '`2*(3+4)`'

properly preserves sharing relationships during evaluation, the arrows in the diagram will literally correspond to pointers in the heap, and a given cell in the structure will never be evaluated twice. Shaded boxes represent cells that would never be evaluated at all in the likely case that the `dvAdditive` result in the leftmost column is the only value ultimately needed by the application.

This illustration should make it clear why this algorithm can run in $O(n)$ time under a lazy evaluator for an input string of length n. The top-level `parse` function is the *only* function that creates instances of the `Derivs` type, and it always creates exactly $n + 1$ instances. The parsing functions only access entries in this structure instead of making direct calls to each other, and each function examines at most a fixed number of other cells while computing a given result. Since the lazy evaluator ensures that each cell is evaluated at most once, the parser has the critical memoization property necessary to guarantee linear parse time, even though it evaluates these results in a completely different order from the tabular, right-to-left, bottom-to-top algorithm presented earlier.

3.2 Practical Extensions to the Algorithm

The previous section provided the basic principles and tools required to create a packrat parser, but building parsers for real applications involves many additional details, some of which are affected by the packrat parsing paradigm. In this section we will explore some of the more important practical issues, while incrementally building on the example packrat parser developed above. We first examine the annoying but straightforward problem of left recursion. Next we address the issue of lexical analysis, seamlessly integrating this task into the packrat parser. Finally, we explore the use of monadic combinators to express packrat parsers more concisely.

3.2.1 Left Recursion

One limitation packrat parsing inherits from TDPL and shares with other top-down parsing schemes is that it does not directly support *left recursion*. For example, suppose we wanted to add a subtraction operator to the above example and have addition and subtraction be properly left-associative. In a context-free grammar, the natural approach would be to use a left-recursive rules such as the following one:

$$\begin{array}{lcl} \text{Additive} & \rightarrow & \text{Additive '+' Multitive} \\ & | & \text{Additive '-' Multitive} \\ & | & \text{Multitive} \end{array}$$

If m_1, m_2, and m_3 are Multitive expressions, then the string 'm_1-m_2-m_3' would be correctly interpreted as '[m_1-m_2]-m_3' and never as 'm_1-[m_2-m_3]'.

However, as described earlier in Section 2.3.5, a rule of this form is erroneous in TDPL because it expresses a degenerate cycle. In a recursive descent parser constructed from such a rule, the `pAdditive` function would recursively invoke itself with the same input it was provided, and therefore would get into an infinite cycle of recursive function calls. In a packrat parser with such a rule, `pAdditive` would attempt to access the `dvAdditive` component of *its own* `Derivs` tuple—the same component it is supposed to compute—and thus create a *circular data dependency.* In either case the parser fails, although the packrat parser's failure mode might be viewed as slightly "friendlier" since modern lazy evaluators often detect circular data dependencies at run-time but cannot detect infinite recursion.

Fortunately, a left-recursive context-free grammar can always be rewritten into an equivalent right-recursive one [2], from which it is generally easy to construct a corresponding TDPL grammar. The desired left-associative semantic behavior is easily reconstructed by using higher-order functions as intermediate parser results.

For example, to make additive expressions left-associative in the example parser, we can split the definition for Additive into two nonterminals, Additive and AdditiveSuffix, yielding the following TDPL definitions:

$$\begin{array}{lcl} \text{Additive} & \leftarrow & \text{Multitive AdditiveSuffix} \\ \text{AdditiveSuffix} & \leftarrow & \text{'+' Multitive AdditiveSuffix} \\ & / & \text{'-' Multitive AdditiveSuffix} \\ & / & () \end{array}$$

Figure 3-4 shows how the corresponding functions in a packrat parser can be defined. The `pAdditiveSuffix` parsing function collects a series of infix operators and right-hand-side operands, and builds a semantic value of function type 'Int → Int'. This higher-order function value, in turn, takes an argument to be used as an initial left-hand-side operand, and computes a result according to the addition/subtraction operators and right-hand-side operands represented by the suffix. The `pAdditive` function recognizes a single Multitive expression followed by an AdditiveSuffix, and then uses the higher-order function produced by `pAdditiveSuffix` as a "modifier" to perform the operations indicated by the suffix on the left-hand-side value.

The `alt3` case in the `pAdditiveSuffix` function also demonstrates how to implement "empty string" rules in TDPL notation: simply return the original `Derivs` object as the "remainder" without parsing anything. Care must be taken that the use of such rules does not introduce subtle left recursion elsewhere in the parser. For example, the definition 'Foo ← Bar Foo' becomes left-recursive if nonterminal Bar can match the empty string.

3.2.2 Integrated Lexical Analysis

Traditional parsing algorithms usually assume that the "raw" input text has already been partially digested by a separate *lexical analyzer* into a stream of tokens. The parser then treats these tokens as atomic units even though each may represent multiple consecutive input characters. This separation is usually necessary because conventional linear-time

```
-- Additive <- Multitive AdditiveSuffix
pAdditive :: Derivs -> Result Int
pAdditive d = case dvMultitive d of
                Parsed vl d’ ->
                  case dvAdditiveSuffix d’ of
                    Parsed f d’’ ->
                      Parsed (f vl) d’’
                    _ -> NoParse
                _ -> NoParse

-- AdditiveSuffix <-
--           ’+’ Multitive AdditiveSuffix
--         / ’-’ Multitive AdditiveSuffix
--         / ()
pAdditiveSuffix :: Derivs -> Result (Int -> Int)
pAdditiveSuffix d = alt1 where

  -- Alternative 1: ’+’ Multitive AdditiveSuffix
  alt1 = case dvChar d of
           Parsed ’+’ d’ ->
             case dvMultitive d’ of
               Parsed vr d’’ ->
                 case dvAdditiveSuffix d’’ of
                   Parsed f d’’’ ->
                     Parsed (\vl -> f (vl + vr))
                            d’’’
                   _ -> alt2
               _ -> alt2
           _ -> alt2

  -- Alternative 2: ’-’ Multitive AdditiveSuffix
  alt1 = case dvChar d of
           Parsed ’-’ d’ ->
             case dvMultitive d’ of
               Parsed vr d’’ ->
                 case dvAdditiveSuffix d’’ of
                   Parsed f d’’’ ->
                     Parsed (\vl -> f (vl - vr))
                            d’’’
                   _ -> alt3
               _ -> alt3
           _ -> alt3

  -- Alternative 3: (empty string)
  alt3 = Parsed (\v -> v) d
```

Figure 3-4: Packrat parsing functions for left-associative addition and subtraction

parsers can only use primitive terminals in their lookahead decisions and cannot refer to higher-level nonterminals. This limitation was explained in Section 3.1.2 for predictive top-down parsers, but bottom-up LR parsers also depend on a similar token-based lookahead mechanism sharing the same problem. If a parser can only use atomic tokens in its lookahead decisions, then parsing becomes much easier if those tokens represent whole keywords, identifiers, and literals rather than raw characters.

Packrat parsing suffers from no such lookahead limitation, however. Because a packrat parser reflects a true backtracking model, decisions between alternatives in one parsing function can depend on *complete results* produced by other parsing functions. For this reason, lexical analysis can be integrated seamlessly into a packrat parser with no special treatment.

Implementing Tokens

To extend the packrat parser example with "real" lexical analysis, we add some new nonterminals to the Derivs type:

```
data Derivs = Derivs {
                -- Expressions
                dvAdditive   :: Result Int,
                ...

                -- Lexical tokens
                dvDigits     :: Result (Int, Int),
                dvDigit      :: Result Int,
                dvSymbol     :: Result Char,
                dvSpacing    :: Result (),

                -- Raw input
                dvChar       :: Result Char}
```

The pSpacing parsing function consumes any whitespace that may separate lexical tokens:

```
pSpacing :: Derivs -> Result ()
pSpacing d = case dvChar d of
                 Parsed c d' ->
                   if isSpace c
                   then pSpacing d'
                   else Parsed () d
                 _ -> Parsed () d
```

The pSpacing function illustrates the handling of longest-match disambiguation, as described earlier in Section 2.4.1. In a more complete language, this function might have the task of "eating" comments as well as space characters. Since the full expressive power of TDPL notation and packrat parsing is available for lexical analysis, comments may have a complex hierarchical structure of their own, such as nesting or markups for literate programming.

Continuing with the lexical analysis example, the function pSymbol recognizes "operator tokens" consisting of an operator character followed by optional whitespace:

```
-- Parse an operator followed by optional whitespace
pSymbol :: Derivs -> Result Char
pSymbol d = case dvChar d of
                Parsed c d' ->
                  if c `elem` "+-*/%()"
                  then case dvSpacing d' of
                         Parsed _ d'' -> Parsed c d''
                         _ -> NoParse
                  else NoParse
                _ -> NoParse
```

Using Tokens in High-Level Constructs

To make use of the lexical analysis defined functions above, we now modify the higher-level parsing functions for expressions to invoke dvSymbol instead of dvChar to scan for operators and parentheses. For example, pPrimary can be implemented as follows:

```
-- Parse a primary expression
pPrimary :: Derivs -> Result Int
pPrimary d = alt1 where

    -- Primary <- '(' Additive ')'
    alt1 = case dvSymbol d of
             Parsed '(' d' ->
               case dvAdditive d' of
                 Parsed v d'' ->
                   case dvSymbol d'' of
                     Parsed ')' d''' -> Parsed v d'''
                     _ -> alt2
                 _ -> alt2
             _ -> alt2

    -- Primary <- Decimal
    alt2 = dvDecimal d
```

This function demonstrates how parsing decisions can depend not only on the *existence* of a match at a given position for a nonterminal such as Symbol, but also on the *semantic value* associated with that nonterminal. In this case, even though all symbol tokens are parsed together and treated uniformly by pSymbol, other rules such as pPrimary can still distinguish between particular symbols. In a more sophisticated language with multi-character operators, identifiers, and reserved words, the semantic values produced by the token parsers might be of type String instead of Char, but these values can be matched in the same way. Such dependencies of syntax on semantic values, known as *semantic predicates* [16], provide a powerful and useful capability in practice. As with the syntactic predicates expressed by the & and ! operators in TDPL, semantic predicates require unlimited lookahead in general and cannot be implemented by conventional parsing algorithms without giving up their linear time guarantee.

There is no direct equivalent to the pPrimary function in basic TDPL notation, because basic TDPL cannot express semantic values or semantic predicates. However, the next chapter will present a "semantic extension" of TDPL in which semantic values and semantic predicates can be expressed.

Advanced Lexical Analysis Capabilities

In a packrat parser with an integrated lexical analyzer, the link between lexical and high-level syntax does not need to be unidirectional. Not only can high-level syntax make reference to lexical elements, but lexical parsing functions can potentially refer "upwards" to the parsing functions for high-level syntactic constructs. For example, a language's syntax could allow identifiers or code fragments embedded within comments to be demarked so the parser can find and analyze them as actual expressions or statements, making intelligent software engineering tools more effective. Similarly, escape sequences in string literals could contain generic expressions representing static or dynamic substitutions.

Another example of the use of lexical tokens with a complex hierarchical structure will be seen in the next chapter, where structured fragments of code in one language are embedded as "literals" in another language.

3.2.3 Monadic Packrat Parsing

A popular method of constructing parsers in functional languages such as Haskell is using monadic combinators [11, 14]. Unfortunately, the monadic combinator approach usually comes with a performance penalty, and with packrat parsing this tradeoff presents a difficult choice. Implementing a packrat parser as described so far assumes that the set of nonterminals and their corresponding result types is known statically, so that they can be bound together in a single fixed tuple to form the Derivs type. Constructing entire packrat parsers dynamically from other packrat parsers via combinators would require making the Derivs type represent a dynamic lookup structure, associating a variable set of nonterminals with corresponding results. This approach would be much slower and less space-efficient.

A more practical strategy, which provides most of the convenience of combinators with a less significant performance penalty, is to use combinators to define the individual parsing *functions* comprising a packrat parser, while keeping the Derivs type and the "top-level" recursion structure static.

The Parser Type

Since we would like our combinators to build the parsing functions we need directly, the obvious method would be to make the combinators work with a simple type alias:

```
type Parser v = Derivs -> Result v
```

Unfortunately, in order to take advantage of Haskell's useful do syntax, the combinators must use a type of the special class Monad, and simple aliases cannot be assigned type classes. We must instead wrap the parsing functions with a "real" user-defined type:

```
newtype Parser v = Parser (Derivs -> Result v)
```

Basic Combinators

We can now implement Haskell's standard sequencing (>>=), result-producing (return), and error-producing combinators:

```
instance Monad Parser where

        (Parser p1) >>= f2 = Parser pre
                where pre d = post (p1 d)
                      post (Parsed v d') = p2 d'
                        where Parser p2 = f2 v
                      post (NoParse) = NoParse

        return x = Parser (\d -> Parsed x d)

        fail msg = Parser (\d -> NoParse)
```

Finally, for parsing we need an ordered choice combinator:

```
(</>) :: Parser v -> Parser v -> Parser v
(Parser p1) </> (Parser p2) = Parser pre
                where pre d = post d (p1 d)
                      post d NoParse = p2 d
                      post d r = r
```

With these combinators in addition to a trivial one to recognize specific characters, the pAdditive function in the original packrat parser example can be written as follows:

```
Parser pAdditive =
            (do vleft <- Parser dvMultitive
                char '+'
                vright <- Parser dvAdditive
                return (vleft + vright))
        </> (do Parser dvMultitive)
```

Iterative Combinators

It is tempting to build additional combinators for higher-level idioms such as repetition and infix expressions. However, using iterative combinators within packrat parsing functions violates the assumption that each cell in the result matrix can be computed in constant time once the results from any other cells it depends on are available. Iterative combinators effectively create "hidden" recursion whose intermediate results are not memoized in the result matrix, potentially making the parser run in super-linear time.

For example, suppose the rule 'A ← x*' is implemented using a "zero-or-more repetitions" combinator, and applied to an input string consisting of n 'x's. If we compute the parsing matrix cell for nonterminal A for each of the $n+1$ possible positions in this string, the computation of each result would start "from scratch" and iterate through the remaining part of the string, making the computation of each cell run in $O(n)$ and making the computation of the entire parsing matrix run in $O(n^2)$. In contrast, if the same grammar is written in terms of primitive, constant-time combinators using the rule 'A ← x A / ()', then the result for each input position builds on the results for positions farther right, and the $O(n)$ time guarantee is preserved.

This problem is not necessarily serious in practice, as the experimental results in Chapter 5 will show. When iterative combinators are used in practical grammars, often only the result for the cell at the beginning of the sequence is ever needed. In this case, lazy evaluation ensures that none of the result cells for the "tails" of this sequence is computed at all, and the effective amount of work done is the same. What this situation amounts to is a partial regression to the more conventional functional backtracking approach, and for many practical grammars occasional backtracking is acceptable. However, it may be difficult to predict whether there are combinations of input strings that might lead to exponential parse times in a complex grammar, and what forms those input strings might take.

A Packrat Parser Combinator Library

The on-line examples for this thesis include a full-featured monadic combinator library that can be used to build large packrat parsers conveniently. This library is substantially inspired by the PARSEC combinator library [14], which is designed for the construction of top-down predictive parsers with support for occasional backtracking. The combinators in the packrat parsing library are much simpler than those in PARSEC, however, since they do not have to treat lexical analysis as a separate phase or implement the one-token-lookahead mechanism used by traditional predictive parsers. The full packrat parsing combinator library provides a variety of "safe" constant-time combinators, as well as a few "dangerous" iterative ones, which are convenient but not necessary to construct useful parsers. The combinator library can be used simultaneously by multiple parsers with different `Derivs` types, and supports user-friendly error detection and reporting as described next.

3.2.4 Error Handling

Graceful error handling is critical for practical parsers that are expected to interpret source files of nontrivial size written by humans. Error handling techniques for conventional LL and LR parsers and their variants are well-studied, but these techniques are not directly applicable to packrat parsers because they generally assume that the parser performs a deterministic left-to-right scan on the input and can simply stop and report an error whenever it gets "stuck." With packrat parsing it is somewhat more difficult to localize or determine the true cause of an error, because most "failures" that occur in the parsing process do not represent errors but merely cause backtracking to an alternate path. Even a "success" result might indicate an error condition: whether a particular result actually contributes to an error fundamentally depends on the context in which the result is used.

Consider the following definition, which might appear in a grammar for an imperative language such as C:

$$\text{Block} \quad \leftarrow \quad \text{`\{' Statement* `\}'}$$

Suppose that the third Statement in a Block in the input text is malformed. In this case the 'Statement*' portion of the rule succeeds anyway, because a '$*$' operator *always* succeeds even if it matches nothing. Since only the first two Statements in the block have been consumed, however, the next character after the matched portion is not be the expected closing brace ('}'), and the Block fails to match. What position should be reported as the location of the error? If we use the position at which at which we expected to find the closing brace, then the error will be reported at the *beginning* of the invalid statement. The invalid statement may itself be a complex construct many lines long, however, such as a nested Block containing many embedded statements. If the actual error is in one of these embedded statements, then reporting the position of the beginning of the block may

```
data Pos = Pos {
                posFile :: String,
                posLine :: Int,
                posCol  :: Int}

data ErrorDescriptor =
          Expected String
        | Message String

data ParseError =
  ParseError {
                errorPos        :: Pos,
                errorDescrs     :: [ErrorDescriptor]}

data Result d v =
          Parsed v d ParseError
        | NoParse ParseError
```

Figure 3-5: `Result` type augmented with error information

provide little useful information. In order to locate the actual error precisely, we must be able to "descend into" the apparent success result generated by the 'Statement*' subrule, and discover *why that subrule did not match more text than it did.*

Representing Error Information

The error handling method presented here is inspired by the method used in the PARSEC combinator library mentioned above. In this solution, *all* results generated by the parsing functions, both success and failure, are augmented with error information, as shown in Figure 3-5. The error information in each `Result` contains two components: a position in the input text (e.g., file name, line number, and column number), and a set of *error descriptors.* Each error descriptor indicates *one possible reason* for any error that may be detected at this position. It is legal for a `ParseError` instance to have no error descriptors; this situation simply means that no information is available (yet) about an error other than the position.

There are two kinds of error descriptors. The first variant, of the form '`Expected` s', indicates that a syntactic element named or described by the string s was scanned for starting at this position, but not found. For example, s may name a specific keyword, operator, or punctuation symbol, or it may name a larger composite construct such as '`expression`' or '`declaration`'. This first kind of error descriptor is by far the most common. The second variant, of the form '`Message` s', indicates a more general syntax error of some kind described by the error message s. This variant allows more specific information to be provided about the cause of an error, which is particularly useful if some amount of "semantic checking" is performed by the parser in addition to pure syntax analysis. For example, if a parsing function for a decimal number successfully reads a string of digits, but discovers that the value of the number is out of the legal range for numeric literals, it might produce

```
-- Two positions are ordered by line number, then column number.
instance Ord Pos where
        Pos f1 l1 c1 <= Pos f2 l2 c2 =
                (l1 < l2) || (l1 == l2 && c1 <= c2)

-- Join two ParseErrors, giving preference to the one farthest right,
-- or merging their descriptor sets if they are at the same position.
joinErrors :: ParseError -> ParseError -> ParseError
joinErrors (e @ (ParseError p m)) (e’ @ (ParseError p’ m’)) =
        if p’ > p || null m then e’
        else if p > p’ || null m’ then e
        else ParseError p (m ‘union‘ m’)
```

Figure 3-6: Joining sets of error descriptors

a `Message` error descriptor with the string ‘`Numeric literal out of range.`’

Propagating Error Information

Because of the inherently speculative nature of packrat parsing, it is generally never known at the time a `ParseError` instance is produced whether the information it contains actually indicates an error. Instead, the error information produced by each parsing function is filtered, combined, and propagated upwards through other parsing functions according to well-defined rules. These rules are defined so that *if* there is an error in the input string, then the most relevant information will “trickle upward” and eventually emerge as part of the failure result produced by the top-level parsing function.

Though there is probably no perfect method of deciding exactly what information is the “most relevant” to an error, a simple heuristic that provides good results in practice is simply to prefer information produced at positions farthest to the right in the input stream. Since TDPL and packrat parsing are inherently oriented toward reading text from left to right, error information corresponding to positions farthest “forward” is generally the most specific and the closest to the actual error.

The core of the error information combination and filtering process is implemented by the `joinErrors` function shown in Figure 3-6. This function takes two `ParseError` instances and combines them into a single instance. If the two instances indicate errors at different positions, then the one farthest forward is returned. If the two instances are associated with the same position, however, then their respective sets of error descriptors are combined with Haskell’s `union` function, which merges the elements of two lists while discarding duplicates.

With this function, we now modify the monadic combinators defined in the last section as shown in Figure 3-7. The `return` combinator produces a `ParseError` with no error information other than the position at which the result was produced, and the `fail` combinator produces a `ParseError` with the supplied message in the obvious way.

The sequencing combinator operates as follows. First, the combinator invokes the left-hand-side parser `p1`. If `p1` fails, then the combinator returns this result verbatim including the error information it contains. If `p1` succeeds, however, then the combinator’s `first`

```
-- Standard monadic combinators
instance Derivs d => Monad (Parser d) where

        -- Sequencing combinator
        (Parser p1) >>= f = Parser parse

                where parse dvs = first (p1 dvs)

                      first (Parsed val rem err) =
                        let Parser p2 = f val
                        in second err (p2 rem)
                      first (NoParse err) = NoParse err

                      second err1 (Parsed val rem err) =
                        Parsed val rem (joinErrors err1 err)
                      second err1 (NoParse err) =
                        NoParse (joinErrors err1 err)

        -- Result-producing combinator
        return x = Parser (\dvs -> Parsed x dvs (ParseError (dvPos dvs) []))

        -- Failure combinator
        fail msg = Parser (\dvs -> NoParse (ParseError (dvPos dvs)
                                                       [Message msg]))

-- Ordered choice
(</>) :: Derivs d => Parser d v -> Parser d v -> Parser d v
(Parser p1) </> (Parser p2) = Parser parse

                where parse dvs = first dvs (p1 dvs)

                      first dvs (result @ (Parsed val rem err)) = result
                      first dvs (NoParse err) = second err (p2 dvs)

                      second err1 (Parsed val rem err) =
                        Parsed val rem (joinErrors err1 err)
                      second err1 (NoParse err) =
                        NoParse (joinErrors err1 err)
```

Figure 3-7: Monadic parser combinators with error handling

function invokes the right-hand-side parser p2 and calls second with p2's result, passing along the error information generated by p1. The function second, in turn, combines the error information from p1 with the error information from p2 in generating the sequencing combinator's final result, *regardless of whether* p2 *succeeds or fails.*

Suppose for example that p1 is the parsing function for the 'Statement*' subrule in the Block construct presented above, and p2 is the parsing function for the closing brace. If there is a syntax error in one of the statements in the sequence parsed by p1, then p1 still produces a success (Parsed) result, but that result includes error information describing why another Statement could not be parsed following the text successfully consumed. The result subsequently produced by p2 will indicate an error at the *beginning* of the invalid statement, such as 'Expected "}"'. If the error detected by p1 indicates a position farther right than this (i.e., if some portion of the invalid statement was successfully parsed before the failure occurred), then the error information from p1 overrides the information from p2.

If, on the other hand, p1 did not get any farther than the end of the last successfully parsed statement (e.g., if the sequence of statements is correct but the block was accidentally terminated with a ']' instead of a '}'), then p1's ParseError will have the same position as p2's, and the sequencing combinator will merge their error descriptors. For example, if the error information from p1 indicates 'Expected "statement"', then the combined ParseError contains the descriptor list '[Expected "statement", Expected "}"]' which might produce the descriptive error message '*(position)*: Expected statement or }'.

The ordered choice operator combines error information in a similar fashion. It first invokes the parser for the first alternative, p1, and returns p1's result directly if successful, including any error information p1 might have produced about why it could not parse a longer string than it did. If p1 failed, then p2 is invoked, and the error information in its result is combined with the error information from p1 regardless of whether p2 succeeded or failed. It is obvious that the two ParseErrors should be combined when p1 and p2 both fail: e.g., if p1 looks for a statement beginning with the keyword 'for', and p2 looks for a statement beginning with the 'while', then if neither is found the parser might produce an error message such as '*(position)*: Expected "for" or "while"'.

To see why the two sets of ParseErrors should be combined even if p2 succeeds, consider the parsing of the actual 'Statement*' subrule in the statement-block example above. This subrule is functionally equivalent to the following right-recursive definition:

$$\begin{array}{rcl} \text{Statements} & \leftarrow & \text{Statement Statements} \\ & / & () \end{array}$$

Suppose p1 is the parsing function for the first alternative, 'Statement Statements', and p2 is the parsing function for the second alternative, which merely parses the empty string. Parser p2 *always* succeeds, producing a ParseError with no error descriptors (since no "error" occurred and nothing was "expected"). Therefore, if some statement in the sequence is invalid, then the information about that error must come from p1 even though p2 succeeded.

Injecting Error Information

The final basic error-handling primitive we need is a convenient way to "inject" error information in the first place. The fail combinator can be used to inject general-purpose error messages. However, error information is more commonly produced through the use of the *error annotation combinator* '<&>', shown in Figure 3-8. This combinator associates a

```
(<?>) :: Derivs d => Parser d v -> String -> Parser d v
(Parser p) <?> desc = Parser (\dvs -> munge dvs (p dvs))

                        where munge dvs (Parsed v rem err) =
                                Parsed v rem (fix dvs err)
                              munge dvs (NoParse err) =
                                NoParse (fix dvs err)

                              fix dvs (err @ (ParseError p ms)) =
                                if p > dvPos dvs then err
                                else ParseError (dvPos dvs) [Expected desc]
```

Figure 3-8: Error annotation combinator

human-readable name with the syntactic construct a parsing function is looking for. When a parser annotated with this combinator generates a result, the combinator compares the position of the ParseError in that result against the starting position of the annotated construct. If the error information in the result is to the right of that starting position, the combinator assumes that it represents more detailed information about a syntactic element *within* the annotated construct, and returns the parser's result unmodified. Otherwise, the combinator replaces the error information from the annotated parser's result with its own Expected descriptor containing the name attached to the combinator.

For example, consider the following annotated monadic parsing function, corresponding to the TDPL definition 'Expression ← '(' Expression '+' Expression ')' / Decimal':

```
pExpression :: Derivs -> Result ArithDerivs Int
Parser pExpression =
            (do char '('
                l <- Parser dvExpression
                char '+'
                r <- Parser dvExpression
                char ')'
                return (l + r))
        </> (do Parser dvDecimal)
        <?> "expression"
```

Suppose pExpression is invoked, and it finds neither an opening parenthesis ('(') nor a decimal digit at that position. Then both alternatives of the choice combinator (</>) fail, the first one indicating 'Expected "("' at the starting position, the second perhaps indicating 'Expected "decimal digit"' at the same input position. Since neither alternative got any farther in the input text than the beginning of the expression, the annotation combinator replaces the error information they produced with the single "higher-level" error descriptor, 'Expected "expression"'.

On the other hand, suppose that the input text at this position starts with '(1+x...'. The first three characters in this string appear to be the beginning of a expression, but the 'x' represents a syntax error. In this case, the first alternative of pExpression matches the '(1+' part of the string and fails at the 'x', yielding a ParseError indicating the position of the 'x'.

The choice combinator then invokes the second alternative, which fails without successfully parsing *any* text, and its error information is therefore overridden by the `ParseError` from the first alternative. The error annotation combinator likewise passes on this `ParseError` information without modification, because its position indicates that it represents more detailed information about an error *within* the expected expression.

Even if most the parsers for high-level constructs in a language are not annotated in this way, useful and detailed error messages can often be produced merely by using appropriate library functions to parse low-level lexical tokens such as keywords and operators. The monadic combinator library includes the following function for this purpose:

```
-- 'string <s>' matches all the characters in <s> in sequence.
string :: Derivs d => String -> Parser d String
string str = p str <?> show str

        where p [] = return str
              p (ch:chs) = do { char ch; p chs }
```

For example, '`string "for"`' creates a `Parser` that scans for the keyword '`for`', and if not found, produces the error descriptor '`Expected "\"for\""`'. The extra pair of double-quotes in the string literal is generated by the use of Haskell's `show` function in `string`, and helps to clarify to the user that the sequence of characters `f-o-r` was expected, and not some kind of syntactic construct known as a "for."

Maintaining Input Position Information

The error handling facilities described above depend on being able to determine the input position corresponding to any `Derivs` tuple. For this purpose we add an additional element to this tuple, with an accessor named `dvPos`:

```
data Derivs = Derivs {
                ...
                dvChar                  :: Result Char,
                dvPos                   :: Pos}
```

These position values are produced by the top-level "tie-up" function for the packrat parser, which we modify as follows:

```
parse :: Pos -> String -> Derivs
parse pos s = d where
    d   = Derivs ... chr pos
    ...
    chr = case s of
            (c:s') -> Parsed c (parse (nextPos pos c) s')
                                (ParseError pos [])
            [] -> NoParse (ParseError pos
                                 [Message "unexpected end of input"])
```

The `parse` function now takes as an additional argument the position at which to start "counting" at the beginning of the input text (e.g., '`Pos` *filename* `1 1`'), and increments it appropriately after each character using the function `nextPos`:

```
-- Incrementally compute the next position in a text file
-- if 'c' is the character at the current position.
-- Follows the standard convention of 8-character tab stops.
nextPos (Pos file line col) c =
        if c == '\n' then Pos file (line + 1) 1
        else if c == '\t' then Pos file line
                                   ((div (col + 8 - 1) 8) * 8 + 1)
        else Pos file line (col + 1)
```

3.2.5 Packrat Parsing with State

The final extension to the packrat parsing algorithm that we sometimes need in practical situations is the ability to parse *context-sensitive grammars*, in which parsing decisions can depend on some kind of *state* built up incrementally throughout the parsing process. The most well-known example of this requirement is in the grammars for C and C++, in which various constructs can be disambiguated properly only through a knowledge of which identifiers are the names of types and which represent ordinary variables. Since new type names can be declared throughout a source file, a symbol table must be kept during the parsing process and updated each time a new type is declared, so that constructs following these declarations can be parsed properly.

State is inherently a problem for packrat parsing, because the algorithm assumes that there is "only one way" to parse a given nonterminal at any given input position. The presence of state violates this assumption because it means there can be many (usually an infinite number of) ways to parse a nonterminal for the same input text, and the state used in parsing one region of the text may be different from that used in parsing another. This problem is not necessarily insurmountable; it just means that wherever a state change occurs, the parser must create and switch to a new derivations structure in mid-stream.

Maintaining State

To add state to a packrat parser, we first incorporate the desired state element as a component in the "derivations" tuple. In addition, efficiency is improved considerably during state changes if we also record in each Derivs instance the original input string supplied to the parse function that created that Derivs instance. For example, if the state element we want to maintain is of type SymbolTable, then we can extend the Derivs type used in the last section as follows:

```
data Derivs = Derivs {
                 ...
                 dvChar                :: Result Char,
                 dvPos                 :: Pos,
                 dvInput               :: String,
                 dvState               :: SymbolTable}
```

We now add a parameter to the parse function specifying the initial state value to be used at the beginning of the parsing process:

```
parse :: Pos -> String -> SymbolTable -> Derivs
parse pos str symtab = d where
    d   = Derivs ... chr pos str symtab
    ...
    chr = case str of
            (c:s') -> Parsed c (parse (nextPos pos c) s' symtab)
                             (ParseError pos [])
            [] -> NoParse (ParseError pos
                                [Message "unexpected end of input"])
```

Referencing State

The parsing functions in the packrat parser can now reference the "current state" simply by using the dvState accessor on the appropriate Derivs instance. In parsing functions defined using monadic combinators, the following combinator makes it easy to access the current state:

```
getState :: Parser SymbolTable
getState = Parser (\d -> dvState d)
```

With this combinator, the current state can be accessed in a Haskell do block with a statement of the form, 'symtab <- getState', after which the identifier symtab is bound to the state in effect at the the getState combinator was invoked.

Changing State

Since every Derivs instance has an associated dvState element, and the top-level parse function always replicates the state value it was invoked with throughout *all* of the Derivs instances it creates up to the end of the input string, we cannot use any of these Derivs instances after a state change. Instead, a parsing function making a state change must create an entirely new result matrix starting from the point at which the state change occurred. The parsing function then returns a Derivs instance from this new result matrix in the "remainder" portion of its result. Since this parsing function was probably called in order to produce some cell in the *original* result matrix, this result cell in the original matrix becomes a direct "link" into the new one. Other parsing functions in the original result matrix that subsequently use the remainder part of this result as a continuation point for further parsing will be automatically "forwarded" into the new result matrix, causing these subsequent actions to occur in the context of the new state.

State changes can be most easily implemented with the help of the following combinator, which complements the getState combinator above:

```
setState :: SymbolTable -> Parser ()
setState newstate = Parser p
        where   p d = Parsed () d' (ParseError pos [])
                        where   pos = dvPos d
                                input = dvInput d
                                d' = parse pos input newstate
```

The setState function extracts the position and the input string from the original Derivs instance, and calls the top-level parse function with this position and input string but with the new state value, in order to create a `Derivs` instance representing the first column in the result matrix for the new state. The parsing function `p` then returns the new `Derivs` instance as the "remainder" part of its success result.

The Dangers of State

It should be obvious that making state changes in a packrat parser is potentially an expensive operation, in terms of both speed and storage consumption. Lazy evaluation is crucial in a stateful packrat parser, since without it we would end up computing a complete parsing matrix for the entire remainder of the input text at every state change. With lazy evaluation, only the cells in each result matrix that are needed by parsing actions *that actually occur in the corresponding state* are computed. A stateful packrat parser in effect computes no more results than a backtracking recursive-descent parser would compute. State changes create the risk, however, that result cells for the same nonterminal and input position may be computed multiple times in different parsing matrices for different states. Results computed in parallel states in this way can defeat the packrat parser's memoization capability and create the potential for super-linear runtime. If state changes occur too often, then packrat parsing is likely to yield no practical benefit over simple backtracking but instead just consume additional storage. For this reason, the viability of packrat parsing with state depends on the properties of the specific language being parsed.

3.3 General Issues and Limitations of Packrat Parsers

Although packrat parsing is powerful and efficient enough for many applications, there are three main issues that can make it inappropriate in some situations. First, packrat parsing only supports *localized backtracking* in which each parsing function produces at most one result. Second, a packrat parser depends for its efficiency on being mostly or completely *stateless*. Finally, due to its reliance on memoization, packrat parsing is inherently space-intensive. These three issues are discussed in this section.

3.3.1 Localized Backtracking

An important assumption we have made so far is that each of the mutually recursive parsing functions from which a packrat parser is built will return *at most one result*. If there are any ambiguities in the grammar the parser is built from, then the parsing functions must be able to resolve them locally, within that parsing function. In the example parsers developed in this thesis, multiple alternatives have always been implicitly disambiguated by the order in which they are tested: the first alternative to match successfully is the one used, independent of whether any other alternatives may also match. This behavior is both easy to implement and useful for performing longest-match and other forms of explicit local disambiguation. A parsing function could even try each of several possible alternatives and produce a failure result if more than one alternative matches. What parsing functions in a packrat parser *cannot* do is return *multiple* results to be used in parallel or disambiguated later by some global strategy.

In languages designed for machine consumption, the requirement that multiple matching alternatives be disambiguated locally is not much of a problem in practice because ambiguity is usually undesirable in the first place, and localized disambiguation rules are preferred over global ones because they are easier for humans to understand. However, for parsing natural languages or other grammars in which global ambiguity is expected, packrat parsing (and TDPL notation) is less likely to be useful. Although a classic generalized top-down parser for context-free grammars in which the parsing functions return lists of results [26, 10, 7] could be memoized in a similar way, the resulting parser would not be linear time, and would likely be comparable to existing tabular algorithms for ambiguous context-free grammars [3, 23]. Since generalized CFG parsing is equivalent in computational complexity to boolean matrix multiplication [13], a linear-time solution to this more difficult problem is unlikely to be found.

3.3.2 Limited State

A second limitation of packrat parsing is that it is fundamentally geared toward stateless parsing. A packrat parser's memoization system assumes that the parsing function for each nonterminal depends only on the input string, and not on any other information accumulated during the parsing process. Although Section 3.2.5 demonstrated how stateful packrat parsers can be implemented, if state changes occur too frequently the algorithm may become inefficient, or under certain conditions even "blow up" and take exponential time *and* space. In contrast, Traditional top-down (LL) and bottom-up (LR) parsers have little trouble maintaining state while parsing. Since these algorithms perform only a single left-to-right scan of the input and never look ahead more than one or at most a few tokens, nothing is "lost" when a state change occurs.

3.3.3 Space Consumption

Probably the most striking characteristic of a packrat parser is the fact that it literally squirrels away *everything* it has ever computed about the input text, including the entire input text itself. For this reason packrat parsing always has storage requirements equal to some possibly substantial constant multiple of the input size. In contrast, LL(k), LR(k), and backtracking recursive-descent parsers can be designed so that space consumption grows only with the *maximum nesting depth* of the syntactic constructs appearing in the input. This nesting depth in practice is often orders of magnitude smaller than the total size of the text. Although LL(k) and LR(k) parsers for any non-regular language still have linear space requirements in the worst case, this "average-case" difference can be important in practice.

One way to reduce the space requirements of the derivations structure, especially in parsers for grammars with many nonterminals, is by splitting up the `Derivs` type into multiple levels. For example, suppose the nonterminals of a language can be grouped into several broad categories, such as lexical tokens, expressions, statements, and declarations. Then the `Derivs` tuple itself might have only four components in addition to `dvChar`, one for each of these nonterminal categories. Each of these components is in turn a tuple containing the results for all of the nonterminals in that category. For the majority of the `Derivs` instances, representing character positions "between tokens," none of the components representing the categories of nonterminals will ever be evaluated, and only the small top-level object and the unevaluated closures for its components occupy space. Even for `Derivs` instances corresponding to the beginning of a token, often the results from only one or two categories are needed depending on what kind of language construct is located at that position.

Even with such optimizations a packrat parser can consume many times more working storage than the size of the original input text. For this reason there are some application areas in which packrat parsing is probably not the best choice. For example, for parsing XML streams, which have a fairly simple syntax but often encode large amounts of relatively flat, machine-generated data, the power and flexibility of packrat parsing is not needed and its storage cost is not justified.

On the other hand, for parsing complex modern programming languages in which the source code is usually written by humans and the top priority is the power and expressiveness of the language, the space cost of packrat parsing is probably reasonable. Standard programming practice involves breaking up large programs into modules of manageable size that can be independently compiled, and the main memory sizes of modern machines leave at least three orders of magnitude in "headroom" for expansion of a typical 10–100KB source file during parsing. Even when parsing larger source files, the working set may still be relatively small due to the strong structural locality properties of realistic languages. Finally, since the entire derivations structure can be thrown away after parsing is complete, the parser's space consumption is likely to be irrelevant if its result is fed into some other complex computation, such as a global optimizer, that requires as much space as the packrat parser used. Chapter 5 will present evidence that this space consumption is reasonable in practical applications.

3.4 A Packrat Parser for Java

In order to illustrate the construction of a full-scale packrat parser for a practical programming language, a parser for the Java programming language is available on-line along with the smaller examples developed in this chapter. This parser is not described here in detail, however, since it is structurally and functionally equivalent to the parser specification that will be explored in the next chapter: only the notational details are different.

Chapter 4

Pappy: A Packrat Parser Generator for Haskell

The last chapter demonstrated how packrat parsers can be expressed directly and concisely in a non-strict functional language such as Haskell, using only the standard features of the language. Since non-strict functional languages are at the present time still little more than a curiosity outside the academic research community, however, we would like to be able to implement packrat parsers easily in more conventional languages as well. For such languages, the obvious approach is to create a "compiler-compiler" tool along the lines of YACC in the C world, which would accept a grammar in a concise notation and produce a working packrat parser in the target language.

Even in the Haskell world, there is practical benefit in using an automatic parser generator instead of coding packrat parsers directly. For example, a grammar compiler can generate not only the parsing functions themselves, but also the appropriate "derivations" type declaration and the top-level recursive "tie-up" function, making it easier to add or remove nonterminals in the grammar. The compiler can also rewrite left-recursive rules in terms of right-recursive ones to make it easier to express left-associative constructs in a grammar, and reduce iterative notations such as the '+' and '*' repetition operators into a low-level grammar that uses only primitive constant-time operations to ensure that the linear parse time guarantee is preserved. Finally, using a specialized high-level notation to express language syntax makes it possible to check and analyze the grammar automatically (e.g., to verify that it does not contain any illegal recursion cycles), and makes it easier to re-use the grammar in different applications.

This chapter presents Pappy, a prototype packrat parser generator that performs all of the functions described above. Pappy is written in Haskell and currently generates only Haskell parsers. Pappy is designed to be retargetable in order to generate parsers for other languages such as Java and C++, but extending it in this way is left for future work. The next section describes Pappy's parser specification language, and the following sections describe the operation of the parser generator and present an example Pappy parser specification for the Java language.

4.1 Parser Specification Language

Naturally, the parser specification language accepted by Pappy is based on TDPL. In addition, the parser specification language will be presented here *in terms of* TDPL. Although

```
Start           ←  Spacing Grammar EOF

Identifier      ←  IdentStart IdentCont* Spacing
IdentStart      ←  Letter / ‘_’
IdentCont       ←  IdentStart / Digit / ‘'’

CharLit         ←  ‘'’ (!(‘'’) QuotedChar) ‘'’ Spacing
StringLit       ←  ‘"’ (!(‘"’) QuotedChar)* ‘"’ Spacing
QuotedChar      ←  ‘\n’ / ‘\r’ / ‘\t’ / ‘\\’ / ‘\'’ / ‘\"’
                /  !(‘\’) Char

Spacing         ←  (SpaceChar / LineComment)*
LineComment     ←  ‘--’ (!(LineTerminator) Char)* LineTerminator

SpaceChar       ←  ‘ ’ / TAB / CR / LF
LineTerminator  ←  CR LF / CR / LF

EOF             ←  !(Char)
```

Figure 4-1: Syntax of basic lexical elements in Pappy parser specifications

the details of the specification language's syntax are not of primary importance, a complete definition of the syntax is included here in order to provide a practical example of a full-scale TDPL syntax specification, complementing the "toy" grammars presented in Chapter 2. The TDPL syntax of Pappy's specification language is included in fragments throughout this section, and can be found as a single unified listing in Appendix B.

4.1.1 Basic Lexical Elements

Figure 4-1 presents the syntax of the basic lexical elements used throughout Pappy's parser specification language. By convention, each construct representing a lexical token is responsible for "consuming" any whitespace and/or comments following the token; the nonterminal Spacing serves this purpose. Comments in Pappy specifications have the same syntax as Haskell comments: they are indicated with a double dash (‘--’), and continue through the end of the line.

The special nonterminal Start is the "top-level" symbol representing a complete Pappy specification file. This nonterminal is merely a "wrapper" for Grammar, to be defined later, which represents the high-level global structure of a Pappy specification. The Start symbol first invokes Spacing in order to allow comments and whitespace to precede the first regular token, and after the complete Grammar has been parsed it checks that the end of file (EOF) has been reached as described in Section 2.3.4.

4.1.2 Semantic Values and Haskell Code Blocks

As with most practical parser generators, Pappy is designed to create parsers that not only recognize the language specified by a grammar, but also compute *semantic values* for recognized constructs. These semantic values can then be used for subsequent processing by code written in the target programming language (in this case Haskell). The semantic values generated by a parser are most commonly used to form an *abstract syntax tree* (AST) representing the the parsed string. An AST represents the high-level structure of the input in a convenient hierarchical form, eliminating irrelevant syntactic details such as textual layout, comments, and syntactic markers such as parentheses and keywords.

In order for an automatically-generated parser to produce meaningful semantic values, the user of the parser generator must have some way to specify how those semantic values are generated. The most common and flexible method of specifying semantic values is to allow actual fragments of code in the target language (Haskell) to be included in the parser specification itself at well-defined points. These code fragments are completely uninterpreted by the parser generator, but instead are simply "pasted" into the generated parser at the appropriate locations. This traditional solution is the one Pappy adopts. In effect, the parser specification language accepted by Pappy is not "pure" TDPL, but rather a language supporting TDPL-like rules containing fragments of Haskell code to specify how semantic values are generated.

The advantage of this approach is that the full power of the target language is available for the computation of semantic values, and the code to compute these values is conveniently located with the corresponding syntax definitions. There are two main disadvantages to this scheme, however:

- Using target-language code fragments in a parser specification makes the specification dependent on the target language and on the type of semantic values to be computed. For example, with this approach a single parser specification cannot be used directly to generate parsers in multiple target languages.

- Since it is usually impractical to re-implement the target language's full syntax and type checking in the parser generator, any erroneous target-language code fragments in a parser specification will generally not be caught by the parser generator. Instead, a source file in the target language will be "successfully" generated, which the target language compiler will subsequently fail to compile. The actual cause of the error in the specification may not be obvious from the error message produced by the target language compiler. This problem is compounded in Pappy because the Haskell target language has no equivalent for the '`#line`' preprocessor directive in C and C++, which can be placed in automatically-generated C or C++ source files to indicate the original source of a code fragment (e.g., a particular line in the parser specification file).

The syntax of Haskell code blocks embedded in Pappy specifications is shown in Figure 4-2. Although the contents of a code block is "mostly" uninterpreted, the syntax of the Haskell code it contains must be examined enough to determine where the block ends—i.e., where the closing brace is. Since the code block itself may contain nested brace pairs as part of Haskell expressions, these nested braces must be matched properly while reading the code block. Since Haskell character and string literals contained in the code blocks may contain *unmatched* curly braces, character and string literals and character escape sequences within them must in turn be detected properly. Finally, since Haskell identifiers can include single-quote characters, all Haskell identifiers in the code block must be parsed as atomic units to prevent a quote character embedded in an identifier from being improperly interpreted as the beginning of a character literal.

```
RawCode                  ←  HaskellBlock Spacing

HaskellBlock             ←  ‘{’ HaskellToken* ‘}’
HaskellToken             ←  HaskellBlock
                         /  HaskellIdentifier
                         /  HaskellCharLiteral
                         /  HaskellStringLiteral
                         /  !(‘{’ / ‘}’ / ‘'’ / ‘"’) Char

HaskellIdentifier        ←  IdentStart IdentCont*

HaskellCharLiteral       ←  ‘'’ HaskellSingleQuoteChar* ‘'’
HaskellSingleQuoteChar   ←  ‘\’ Char
                         /  !(‘'’ / CR / LF) Char

HaskellStringLiteral     ←  ‘"’ HaskellDoubleQuoteChar* ‘"’
HaskellDoubleQuoteChar   ←  ‘\’ Char
                         /  !(‘"’ / CR / LF) Char
```

Figure 4-2: Syntax of raw Haskell code blocks embedded in Pappy specifications

All of this syntactic machinery is easily expressed in TDPL, and it serves as a prime example of the advantages of TDPL over traditional "two-stage" syntactic paradigms in which lexical analysis is considered separate from parsing. In Pappy's parser specification syntax, a Haskell code block is treated as a single lexical token, representing an uninterpreted string of characters to be deposited verbatim in the Haskell source file for the generated parser. Nevertheless, the *syntactic specification* of this special kind of "token" involves complex recursive structures such as HaskellBlock, which conventional lexical analyzer generators based on regular expressions could not be expected to handle. For this reason, the syntax of embedded code fragments of this kind is traditionally not expressed in a formal syntax at all, but is instead merely described informally and used to construct a hand-coded lexical analyzer.

4.1.3 Global Structure

Now that the lexical building blocks of Pappy's syntax have been established, we move to the global structure of a Pappy specification, which is summarized in Figure 4-3. A parser specification consists a text file containing of the following main components in order:

1. The first component is a header of the form, ‘`parser` *name*`:`’, where *name* is a legal Haskell identifier that will be used as a prefix in various other identifiers in the generated parser. It does not matter whether *name* starts with an uppercase or lowercase letter: Pappy will convert its first letter to uppercase when generating Haskell type or constructor names, and to lowercase when generating ordinary Haskell identifiers.

2. An optional block of Haskell code can be included following the header, enclosed in curly braces (`{}`), which will be inserted verbatim (without the braces), into the

```
Grammar       ←  Header RawCode? TopDecl Definition* RawCode?

Header        ←  PARSER Identifier COLON
TopDecl       ←  TOP Nonterminal (COMMA Nonterminal)*

Nonterminal   ←  Identifier

PARSER        ←  'parser' Spacing
TOP           ←  'top' Spacing
COLON         ←  ':' Spacing
COMMA         ←  ',' Spacing
```

Figure 4-3: Global structure of Pappy's parser specification language

generated parser as "top-level" Haskell code, before any other Haskell declarations produced by the parser generator. This code block is typically used to declare Haskell types and type classes for the semantic values to be generated by the parser.

3. Next comes a mandatory declaration of the form, '`top` *names*', where *names* is a list of one or more nonterminal names used in the grammar, separated by commas. Each nonterminal listed in this declaration will be considered by the parser generator to be a "top-level" nonterminal, which ensures that it will have a corresponding component in the "derivations" tuple for the generated packrat parser, with a corresponding Haskell accessor function. Other nonterminals not declared "top-level" in this way may be inlined or otherwise "optimized away" by Pappy before the parser is generated.

4. The main component of the specification is the grammar itself. The grammar is a sequence of definitions, one for each nonterminal, with an associated Haskell type and a corresponding right-hand-side expression indicating how that nonterminal is to be parsed and how any associated semantic value is to be generated.

5. Finally, a second optional block of Haskell code can be included at the end of the specification, again enclosed in curly braces (`{}`). This block is included in the generated parser in the same way as the one above.

4.1.4 Nonterminal Definitions

Figure 4-4 presents the syntax of nonterminal definitions, of which the bulk of a Pappy parser specification is composed. Each definition consists of a nonterminal name, a Haskell type, and a parsing rule expression.

The Haskell type in a definition describes the semantic value for the nonterminal, and can be expressed either as a single Haskell identifier or as a raw Haskell code block. A code block can be used to express complex Haskell types such as functions, tuples, lists, or user-defined parameterized types. For example, the following Pappy definition assigns the nonterminal Comment a Haskell type of '`()`', the empty tuple type:

```
Comment :: {()} =
        TraditionalComment
```

```
Definition     ←  Nonterminal DOUBLECOLON HaskellType EQUALS Rule

HaskellType    ←  Identifier
               /  RawCode

DOUBLECOLON    ←  '::' Spacing
EQUALS         ←  '=' Spacing
```

Figure 4-4: Syntax of nonterminal definitions in Pappy

```
            / EndOfLineComment
```

4.1.5 Parsing Rules

The syntax of Pappy parsing rules is shown in Figure 4-5. Rules have four precedence levels: in order from highest to lowest, primary rules, unary operators, sequencing, and ordered choice. Primary rules consist of nonterminal names, character and string literals for matching raw input characters, and rules enclosed in parentheses. The unary postfix operators are '?' (optional), '*' (zero-or-more), and '+' (one-or-more). In order to handle the construction of semantic values, the syntax of the sequencing operator in a Pappy parser specifications is somewhat different from basic TDPL notation, as described in the next section. Finally, the ordered choice operator '/' has lowest precedence.

All of these operators express parsing rules that produce semantic values, and since the target language is Haskell, each semantic value must have a Haskell type. The Haskell type for the semantic value of any given rule expression can be determined as follows:

- The Haskell type of a primary rule consisting of a simple nonterminal identifier is the Haskell type associated with that nonterminal in its definition.
- A character literal rule yields a semantic value of Haskell type '`Char`'.
- A string literal rule yields a semantic value of Haskell type '`String`'.
- If rule r has Haskell type t, then the rule 'r?' has Haskell type `Maybe` t (i.e., an optional instance of type t).
- If rule r has Haskell type t, then the rules 'r*' and 'r+' have Haskell type [t] (i.e., a list of elements of type t).
- If rules $r_1, \ldots, r_n$ all have Haskell type t, then the rule 'r_1 / ... / r_n' has type t.

Pappy does not perform full checking of Haskell types: doing so would be difficult in general because of the sequencing operator described below, which allows semantic values to be computed by arbitrary Haskell expressions. If there is a mismatch between a definition's declared type and the type of semantic value produced by the rule on the definition's right-hand side, or if different sub-rules in an alternation expression have different types, then the error will not be detected until the Haskell compiler attempts to compile the generated parser. These types must nevertheless be declared for the nonterminals in a Pappy specification, because they are required by the parser generator in order to produce a Haskell declaration for the `Derivs` tuple type in the resulting packrat parser.

```
PrimRule    ←  Nonterminal
             /  CharLiteral
             /  StringLiteral
             /  OPEN Rule CLOSE
UnaryRule   ←  PrimRule QUESTION
             /  PrimRule STAR
             /  PrimRule PLUS
             /  PrimRule
SeqRule     ←  Sequence
             /  UnaryRule
AltRule     ←  SeqRule (SLASH SeqRule)*
Rule        ←  AltRule

OPEN        ←  ‘(’ Spacing
CLOSE       ←  ‘)’ Spacing
QUESTION    ←  ‘?’ Spacing
STAR        ←  ‘*’ Spacing
PLUS        ←  ‘+’ Spacing
SLASH       ←  ‘/’ Spacing
```

Figure 4-5: Syntax of parsing rules in Pappy

4.1.6 The Sequencing Operator

The standard TDPL notation for sequencing, in which multiple subexpressions are merely adjoined end-to-end, is not directly suitable for Pappy specifications because it provides no indication of how a semantic result for the sequence is to be computed from the semantic results of the individual components of the sequence. For this reason, Pappy uses an explicit binary operator, ‘->’, to indicate sequencing. The syntax of this operator is shown in Figure 4-6.

On the left-hand side of a ‘->’ operator is a series of *matchers* specifying the elements of the sequence to match, and on the right-hand side is a *result* expression indicating how the final semantic value is to be computed. Matchers serve two primary purposes: first, to specify the parsing rule to be used to parse each component in the sequence; and second, to specify how the semantic value resulting from that parsing rule is to be matched and bound to Haskell identifiers so that it can be used in the computation of the result. In addition, matchers also serve the purpose of expressing syntactic and semantic predicates.

The semantic value to be computed by a sequence can be expressed in two ways. First, the result can be expressed as a single Haskell identifier, in which case that identifier must be bound (i.e., produced) by one of the matchers on the left-hand side of that sequence operator. Second, the result can be computed by an arbitrary Haskell expression enclosed in curly braces. In this case, the expression can refer to any number of semantic values bound to different Haskell identifiers by matchers on the left-hand side of the sequence operator.

```
Sequence     ←  SeqMatcher* ARROW SeqResult
SeqMatcher   ←  Identifier COLON UnaryRule
             /  RawCode COLON UnaryRule
             /  CharLiteral COLON UnaryRule
             /  StringLiteral COLON UnaryRule
             /  AND UnaryRule
             /  NOT UnaryRule
             /  AND RawCode
             /  UnaryRule
SeqResult    ←  Identifier
             /  RawCode

ARROW        ←  '->' Spacing
COLON        ←  ':' Spacing
AND          ←  '&' Spacing
NOT          ←  '!' Spacing
```

Figure 4-6: Syntax of the Pappy sequencing operator

Here is an example definition involving two sequencing operators:

```
ExponentPart :: Integer =
          ('e' | 'E') '-' v:Digits -> {-v}
        | ('e' | 'E') '+'? v:Digits -> v
```

Both sequences have three matchers on their left-hand sides. In the first sequence, the semantic result is computed by the Haskell expression '-v', where v is an identifier bound to the semantic value produced by the `Digits` component. In the second sequence, the result is expressed as the simple identifier v instead of a Haskell expression.

Pappy supports the following kinds of matchers in a sequence operator:

- An *anonymous matcher* consists of an unadorned UnaryRule (the final, "default" alternative in the SeqMatcher definition in Figure 4-6). An anonymous matcher causes the rule to be invoked in the sequence, but "throws away" the resulting semantic value without binding it to any Haskell identifier.

  The first component of both sequences in the example above is an anonymous matcher consisting of the sub-rule '`('e' | 'E')`', which matches an 'e' or 'E' character in the input without binding the semantic value of the subrule to any Haskell identifier. The second component of each sequence is also an anonymous matcher, matching a '-' sign in the first case and an optional '+' sign in the second.

- An *identifier matcher* takes the form i:r, where i is a Haskell identifier and r is a unary parsing rule. An identifier matcher invokes the rule and binds its semantic value to the Haskell identifier i for use in the result computation for the sequence.

  The third component of both sequences above is the identifier matcher '`v:Digits`', which binds the Haskell identifier v to the semantic value produced by the subrule

Digits. This semantic value of Digits should be of Haskell type Integer, in order to be compatible with the declared type of ExponentPart.

- A *pattern matcher* takes the form $\{p\}$:r, where p is an arbitrary Haskell pattern and r is a parsing rule. The Haskell pattern p may bind any number of identifiers, and will be used to match against the semantic result produced by the rule r in order to assign values to those identifiers. For example, if the semantic value of nonterminal Foo is a pair, then the sequence expression '{(x,y)}:Foo -> x' invokes the nonterminal Foo and returns the first component of its semantic value as the semantic value of the sequence.

  The Haskell pattern used in a pattern matcher does not need to be exhaustive ("irrefutable"). If the rule r succeeds but the pattern p fails to match its semantic result, then the entire sequence will fail exactly as if the rule r itself had failed. Thus, using a refutable pattern in a matcher is a simple way to express a semantic predicate, allowing parsing decisions for one syntactic construct to depend on the semantic values computed for other constructs. For example, if nonterminal Bar yields a semantic value of Haskell type Maybe Integer, then the sequence rule '{Just x}:Bar -> x' will succeed only if the rule for nonterminal Bar succeeds *and* generates a value matching the Just constructor. If Bar fails or if it produces the value Nothing, then the sequence as a whole fails to match.

- A *character matcher* has the form 'c':r, where c is a single character or a Haskell character escape sequence and r is a unary parsing rule. The rule r must produce a semantic value of Haskell type Char; if the semantic result of r is not the designated character, then the sequence as a whole fails. This notation is essentially a shorthand for the Haskell pattern matcher {'c'}:r.

- A *string matcher* has the form "s":r, where s is any number of characters suitable for a Haskell string literal, and r is a unary parsing rule. As with a character matcher, a string matcher is just a shorthand notation for the pattern matcher {"s"}:r.

- An *"and-followed-by" matcher* has the form &r, where r is a unary rule. This matcher implements syntactic predicates: it causes the rule r to be invoked at the appropriate position in the sequence, and if it succeeds, the input position is backed up to the position before r was invoked, acting as if r had not consumed any input text. If r fails, then the sequence as a whole fails.

- A *"not-followed-by" matcher* has the form !r, where r is a unary rule, and implements the negative form of syntactic predicate: if r succeeds, then the sequence as a whole fails; but if r fails, then the matcher succeeds without consuming any input text, and parsing of the sequence is allowed to continue.

- The final type of matcher, an *"and-predicate" matcher*, implements semantic predicates in a general form. An and-predicate matcher takes the form &$\{e\}$, where e is arbitrary Haskell expression producing a value of type Bool. If the predicate expression evaluates to True, then parsing of the sequence is allowed to continue; if the semantic predicate evaluates to False, then the sequence fails.

  The expression e can refer to any Haskell identifiers bound by matchers in this sequence up to the point at which this matcher appears. For example, in the sequence 'x:Foo &{*e*} y:Bar -> y', the semantic predicate expression *e* can make reference to x but not to y.

One simple example of how semantic predicates can be used is to implement character classes generically in terms of built-in Haskell functions:

```
Letter :: Char = c:Char &{isAlpha c} -> c
Digit :: Char = c:Char &{isDigit c} -> c
...
```

4.2 Parser Specification Reduction and Validation

After reading a parser specification, Pappy automatically performs several basic validity checks on the specification. For example, it ensures that no nonterminal is defined more than once, and that every nonterminal referenced in a rule has a definition. The parser generator then performs two transformations on the specification, in order to rewrite it into a form that can be used to implement a linear-time packrat parser directly. First, simple left-recursive definitions are rewritten into right-recursive ones. Second, iterative '*' and '+' rules are rewritten in terms of simple right-recursive definitions. These two transformations and their implications for parser specifications are described in detail below.

4.2.1 Rewriting Left-Recursive Definitions

As described earlier in Section 2.3.5, left recursion in a TDPL grammar is normally considered erroneous. A TDPL definition of the form 'A ← A ...' cannot be used to read anything because it effectively means, "In order to read an A, first try to read an A ..." In ordinary TDPL notation whose purpose is only to express a language's syntax, there is usually little need for the equivalent of a left (or right) recursive CFG definition because the same effect can be achieved more concisely using the '*' and '+' repetition operators.

For a parser generator such as Pappy, however, in which the generated parser is expected not just to recognize a string but to compute semantic values based on its syntactic structure, left and right recursion can be useful because they allow the semantic value of a sequence of syntactic elements to be computed *incrementally* from the values of the individual elements in the sequence. The '*' and '+' operators always produce lists, which may not be the type of semantic value desired from the sequence.

For example, consider the following fragment of a trivial expression language:

Expression ← Number ('+' Number)*

If we just wanted the semantic value of Expression to be a list of the semantic values of the Numbers in the sequence, then we could convert this definition directly into the following Pappy definition:

```
Expression :: {[Integer]} =
          n:Number ns:('+' m:Number -> m)* -> {n : ns}
```

However, if what we really want is to *add the numbers together* and use the sum as the semantic value of the Expression, then using the * operator would mean building a useless list only to take it apart again:

```
Expression :: Integer =
          n:Number ns:('+' m:Number -> m)* -> {foldl (+) n ns}
```

This task becomes more cumbersome if there is more than one kind of operator:

```
Expression :: Integer =
          n:Number t:ExprOp* -> {foldl (\n' op -> op n') n ns}

ExprOp :: {Integer -> Integer}
          '+' m:Number -> {\n -> n + m}
        / '-' m:Number -> {\n -> n - m}
```

For right-associative operators, or operators such as addition or multiplication for which associativity does not matter, using right recursion makes it easy and natural to add up the numbers incrementally:

```
Expression :: Integer =
          n:Number '+' m:Expression -> {n + m}
        / Number
```

If we attempted to parse a binary subtraction operator using right recursion in this way, however, a string such as '4 - 3 - 2' would be improperly interpreted as '5 - (3 - 2)' instead of '(5 - 3) - 2', leading to the incorrect result 4 instead of 0. Clearly what we want is to be able to express left recursion directly.

Simple Left Recursion

Since left-recursive definitions would otherwise be invalid in a Pappy specification, we are free to assign a meaning that suits our pragmatic needs even if that meaning does not quite fit into the TDPL paradigm. Therefore, for convenience in expressing left-associative constructs, Pappy automatically rewrites certain kinds of left-recursive definitions in terms of right-recursive definitions. Only *direct* left recursion can be rewritten in this way: all left-recursive references in the parsing rule on the right-hand side must refer directly to the nonterminal on the left-hand side of the definition, and not indirectly through other nonterminals. The right-hand side of the left-recursive definition must be a choice construct, and all of the left-recursive alternatives in this construct must have exactly the form 'i:n ... -> ...' where i is a Haskell identifier and n is the nonterminal being defined. Alternatives on the right-hand side that are not left-recursive can take other forms. For example, here is a left-recursive definition for the simple language above with addition and subtraction:

```
Expression :: Integer =
          n:Expression '+' m:Number -> {n + m}
        / n:Expression '-' m:Number -> {n - m}
        / Number
```

To rewrite a definition such as this one, Pappy separates all of the left-recursive alternatives from the rest (the first two in this case), and transforms the definition into a pair of definitions like this:

```
Expression :: Integer =
          n:Number f:ExpressionTail -> {f n}

ExpressionTail :: Integer =
          ’+’ m:Number f:ExpressionTail -> {\n -> f (n + m)}
        / ’-’ m:Number f:ExpressionTail -> {\n -> f (n + m)}
        / -> {\n -> n}
```

In essence, Pappy automatically performs the "left factoring" that we did in Section 3.2.1 to construct a packrat parser manually for left-associative operators.

Due to the basic left-to-right asymmetry of the TDPL paradigm, left-recursive Pappy definitions that are transformed in this way do not behave exactly like "mirrored" right-recursive definitions. For example, because the left-recursive alternatives are separated out from the original definition, the relative order of a left-recursive alternative and a non-recursive alternative in the definition does not matter, although the relative order of the alternatives in each category might be important. For example, the following left-recursive definition is functionally equivalent to the previous one:

```
Expression :: Integer =
          Number
        / n:Expression ’+’ m:Number -> {n + m}
        / n:Expression ’-’ m:Number -> {n - m}
```

In a comparable right-recursive definition, the second and third alternatives would never match because the first alternative would always succeed first and override them. For practical purposes, however, this asymmetry should not usually be an issue.

Indirect Left Recursion

After simple left-recursive definitions have been rewritten as described above, Pappy checks the entire grammar for any less direct forms of left recursion, and signals an error if illegal recursion is found. This check ensures that the parsing process will terminate—assuming, of course, that all of the Haskell expressions that compute semantic values or predicates terminate.

An obvious question is whether Pappy's support for rewriting left-recursive definitions could be made more general, for example to handle indirect left recursion among multiple nonterminals. Although such a transformation should be possible to implement, it is not clear how the behavior of the transformed grammar could be characterized in an a clear and meaningful way in the TDPL paradigm. A "generalized" left-recursive TDPL construct can be expected to behave differently from a superficially similar left-recursive construct in a CFG, given the fundamental differences between the two paradigms. Furthermore, it is clear that due to the left-to-right asymmetry of TDPL noted above, such a construct could not be expected to behave like a mirrored version of a right-recursive TDPL construct. At least until left recursion in TDPL is studied further, utilizing such a feature would amount to opening a syntactic Pandora's Box, which clearly defeats the pragmatic purpose for which the simple left recursion transformation is provided. There is a clear danger that generalized left recursion rewriting would be triggered by language designers accidentally more often than intentionally, through subtle dependencies between nonterminals that unexpectedly lead to left recursion. In such cases, it is probably much more useful for the language designer to be informed about the problem up front than for the parser generator to "solve" it silently, in an obscure way that may lead to unexpected results.

4.2.2 Rewriting Iterative Rules

As pointed out in Section 3.2.3, it is not difficult to implement iterative syntactic constructs such as those expressed by the '*' and '+' operators via Haskell combinators or direct functional code, but implementing them in this way can invalidate the packrat parser's linear time guarantee. Pappy instead implements repetition operators in a parser specification by rewriting them in terms of basic sequencing, choice, and recursion.

To rewrite a subexpression '$r*$' within a parsing rule definition, Pappy creates a new right-recursive nonterminal to match a sequence of instances of r and build their semantic values into a list. The parser generator then substitutes the new nonterminal for the '$r*$' in the original definition. For example, consider the following definition:

```
Word :: String =
          c:Letter cs:LetterOrDigit* Spacing -> {c : cs}
```

Pappy expands this definition into a pair of definitions with the following structure:

```
Word :: String =
          c:Letter cs:LetterOrDigitStar Spacing -> {c : cs}

LetterOrDigitStar :: {[Char]} =
          c:LetterOrDigit cs:LetterOrDigitStar -> {c : cs}
        / -> {[]}
```

Instances of the one-or-more repetitions operator, '+', are rewritten similarly, except that the second alternative in the iterative rule (the "base case") expects one instance of the repeated subrule and produces a one-element list, rather than matching nothing and producing an empty list:

```
LetterOrDigitPlus :: {[Char]} =
          c:LetterOrDigit cs:LetterOrDigitPlus -> {c : cs}
        / c:LetterOrDigit -> {[c]}
```

Unfortunately a minor practical difficulty arises with rewriting rules in this way. The subrule r to which the repetition operator is applied can be an arbitrary parsing rule expression. In order to create the new iterative definition, however, Pappy must be able to determine the Haskell type of the semantic value it generates, so that it can associate the proper corresponding list type with the newly created nonterminal. (As mentioned earlier, Pappy must know the Haskell type associated with every nonterminal in the grammar in order to declare an appropriate `Derivs` tuple type in the generated parser.)

In the most common cases, Pappy can infer the Haskell type for the iterated subrule based on examination of the subrule. For example, in the above case where the subrule is the name of a nonterminal (`LetterOrDigit`), the Haskell type is simply taken from the definition of that nonterminal (`Char` in this case) and enclosed in brackets to create the corresponding Haskell list type (`[Char]`, which in Haskell is the same as `String`). However, Pappy does not have the machinery to parse, let alone type-infer, the arbitrary Haskell patterns and expressions that may be used to compute semantic values in a sequence rule. If the appropriate Haskell type for the subrule of a repetition operator cannot be inferred due to dependence on Haskell patterns or expressions, then Pappy aborts with an error. For example, Pappy can infer the appropriate Haskell type for the iterated subrule in the first of the following three examples, but not the other two:

```
TraditionalComment :: {()} =
        "/*" (!"*/" c:Char -> c)* "*/" -> {()}      -- Legal

TraditionalComment :: {()} =
        "/*" (!"*/" c:Char -> {c})* "*/" -> {()}    -- ILLEGAL

TraditionalComment :: {()} =
        "/*" (!"*/" {c}:Char -> c)* "*/" -> {()}    -- ILLEGAL
```

In the first case, it is clear to Pappy that the subrule '`(!"*/" c:Char -> c)`' generates a semantic value of Haskell type `Char`, because the result to be generated by the sequence is derived directly from the result of the second matcher in the sequence, and this binding is expressed via simple identifiers. In the second case, however, the result of the sequence is computed by a Haskell expression (albeit a seemingly trivial one), which Pappy is unable to interpret. In the third case, the result of the sequence is indicated with a simple identifier, but Pappy cannot determine how this identifier is bound because its binding is within a Haskell pattern. This limitation is occasionally inconvenient, but does not cause a serious problem in practice, because it can always be solved by separating out the iterated subrule into a nonterminal definition of its own, with an explicitly declared Haskell type.

4.3 A Pappy Parser Specification for Java

This section describes an example Pappy parser specification for the Java language, which is functionally equivalent to the example monadic packrat parser mentioned earlier in Section 3.4. While only fragments of the Pappy specification for Java will be presented here, the complete specification is available on-line.

This parser specification provides a good illustration of the power and expressiveness of the Pappy specification language. The Pappy specification is just over 700 lines long, significantly shorter than the monadic Haskell version, which is over 1000 lines. Apart from the trivial preprocessing of Unicode escape sequences, lexical analysis is seamlessly integrated into the parser specification, instead of being implemented in a separate stage as would be necessary with a traditional LR parser generator.

Figure 4-7 summarizes the header portion of the Pappy specification for Java. Aside from the declaration of the parser name ('`Java`') at the top and the declaration of the top-level nonterminal (`CompilationUnit`), the header primarily consists of raw Haskell code. The purpose of most of this code is to declare the Haskell data types used to represent Java abstract syntax trees. The `keywords` constant is a list of the reserved words in the Java language, and is used later in the parsing of identifiers.

Next in the parser specification comes the grammar proper, which begins with the definitions for nonterminals representing the lexical elements of the language. As we did in the earlier TDPL specification of the syntax of Pappy's parser specification language itself, the Java parser specification adopts the convention that the parsing rule for each token is

```
-- Pappy packrat parser specification for the Java language version 1.1
parser Java:

{
import Char
import System
import Numeric

-- Abstract syntax tree data types
type Identifier = String
type Name = [Identifier]

data Literal =    LitInt Integer
                | LitLong Integer
                | LitFloat Float
                | LitDouble Double
                | LitChar Char
                | LitString String
                | LitBool Bool
                | LitNull

data Expression = ExpLiteral Literal
                | ExpIdent Identifier
                | ...

... (other abstract syntax tree data types) ...

-- List of Java's reserved words, used in Keyword below
keywords = [
        "abstract",
        "boolean", "break", "byte",
        "case", "catch", "char", "class", "const", "continue",
        ...
    ]
}

top CompilationUnit
```

Figure 4-7: Header portion of Pappy parser specification for Java

```
Spacing :: {()} =
          Space*                                               -> {()}

Space :: {()} =
          WhiteSpace                                           -> {()}
        / Comment                                              -> {()}

WhiteSpace :: Char =
          ' ' / '\t' / '\f' / LineTerminator

Comment :: {()} =
          TraditionalComment
        / EndOfLineComment

TraditionalComment :: {()} =
        "/*" (!"*/" c:Char -> c)* "*/"                         -> {()}
EndOfLineComment :: {()} =
        "//" (!LineTerminator c:Char -> c)* '\n'               -> {()}
```

Figure 4-8: Parsing rules for Java whitespace and comments

responsible for consuming any whitespace and comments following the token itself. This function is performed by the Spacing nonterminal, defined in Figure 4-8. Since whitespace and comments are normally ignored in subsequent processing, most of these definitions simply produce the empty tuple (‘`()`’) as their semantic result value. However, if it was important to retain this information, for example in an application that reads a Java source file and then writes it back out as modified Java source, then the text comprising whitespace and comments could be retained in the abstract syntax tree as well.

Both traditional C-style comments (‘`/* comment */`’) and C++-style comments (‘`// comment extending to the end of line`’) are easily handled using the ‘`*`’ repetition operator. For example, between the ‘`/*`’ and ‘`*/`’, `TraditionalComment` expects any number of characters matching the subrule ‘`(!"*/" c:Char -> c)`’, which means, “any character as long as it is not the beginning of a ‘`*/`’ sequence.” Since Java disallows nested comments, any ‘`/*`’ sequences within the comment are ignored. Changing the definition to support nested comments would also be easy, in contrast with traditional lexical analyzer generators, which are usually limited to non-recursive regular expressions:

```
TraditionalComment :: {()} =
        "/*" (TraditionalComment -> {' '} / !"*/" c:Char -> c)* "*/"
                -> {()}
```

Figure 4-9 shows the rules for parsing Java reserved words (“keywords”) and identifiers. The use of semantic predicates makes possible an even simpler approach to this task than the one we outlined in Section 2.4.2 using pure “syntax-only” TDPL. In the approach adopted here, a single nonterminal, `Word`, is responsible for parsing “words” of any kind, whether they represent identifiers or keywords. The semantic value of a `Word` is the string of characters comprising the identifier or keyword. The `Keyword` nonterminal then uses a Haskell semantic

```
-- Keywords and identifiers

Identifier :: Identifier =
        !Keyword !BooleanLiteral !NullLiteral s:Word            -> s

Keyword :: String =
          s:Word       &{s `elem` keywords}                     -> s

Word :: String =
          c:JavaLetter cs:JavaLetterOrDigit* Spacing            -> {c : cs}

JavaLetter :: Char =
          c:Char                &{isAlpha c}                    -> c
        / '_'
        / '$'

JavaLetterOrDigit :: Char =
          c:Char                &{isAlphaNum c}                 -> c
        / '_'
        / '$'
```

Figure 4-9: Parsing rules for Java keywords and identifiers

predicate to recognize keywords in the `keywords` list defined earlier. Finally, the `Identifier` nonterminal uses negative syntactic predicates to prevent keywords or "word-literals" from being accepted as identifiers. (The Java language specification classifies the special words `true`, `false`, and `null` separately as "literals" rather than as "keywords," and the Pappy parser adopts the same convention for consistency with the language specification, even though treating these names as keywords might yield a slightly simpler and more efficient parser.)

Java operators and punctuation are handled together by the definitions in Figure 4-10. As with most practical programming languages, Java includes a number of operators consisting of multiple characters, and the standard "longest-match" rule applies when reading such operators. For example, the character sequence '>>' is *always* interpreted as a single logical right shift operator and never as two consecutive greater than ('>') operators. For this reason, all of the operators and punctuation symbols in Java are listed together in the definition of `SymChars`, with the longest ones first so that they have priority. The `Sym` nonterminal represents a "symbol token" and matches `SymChars` followed by optional `Spacing`. As with keywords, the semantic value of a `Sym` is the sequence of characters comprising the operator, and it is this semantic value that is used later in the grammar to distinguish different symbols from one another.

The parsing of integer, floating point, character, string, boolean, and `null` literals is straightforward but involves many details that are not important to this thesis, so the definitions for these lexical constructs are not described here. However, Pappy's left recursion rewriting comes in particularly useful in parsing numbers:

```
-- Symbols (operators and punctuation)
Sym :: String =
          s:SymChars Spacing                                      -> s

SymChars :: String =
          ">>>="
        / ">>=" / "<<=" / ">>>"
        / ">>" / "<<"
        / "+=" / "-=" / "*=" / "/=" / "%=" / "&=" / "^=" / "|="
        / "++" / "--" / "&&" / "||" / "<=" / ">=" / "==" / "!="
        / ";" / ":" / "," / "." / "{" / "}" / "(" / ")"
        / "[" / "]" / "!" / "~" / "+" / "-" / "*" / "/"
        / "%" / "<" / ">" / "=" / "&" / "^" / "|" / "?"
```

Figure 4-10: Parsing rules for Java symbols

```
Digits :: Integer =
          v:Digits d:Digit                  -> {v * 10 + toInteger d}
        / d:Digit                           -> {toInteger d}

Digit :: Int =
          c:Char        &{isDigit c}        -> {digitToInt c}
```

Figure 4-11 shows the parsing rules for type expressions, providing an example of how the basic lexical elements above such as identifiers, keywords, and symbols are used throughout the grammar. For example, the definition for `PrimitiveType` has eight alternatives, each of which refer to the same nonterminal `Word`. Each of these references is qualified using a specific string as a semantic predicate, however, in order to recognize the specific Java keywords denoting primitive types. The definition of `PrimitiveType` can be informally interpreted, "Look for a `Word` with the semantic value '`byte`', or else a `Word` with the semantic value '`short`', or else ..." The definition of `Dims` similarly uses string matchers to recognize the specific symbols '`[`' and '`]`'. Using semantic predicates in this way allows basic syntactic elements such as keywords and symbols to be treated uniformly at a low level (the lexical level in this case), but to be used in different ways in higher syntactic levels.

Figure 4-12 shows a sample of the rules for Java expressions, including both right-associative operators such as assignment (the lowest precedence level), and left-associative operators such as postfix array access and method calls, which make use of left recursion rewriting. Although traditional parser generators such as YACC often allow the precedence levels of infix operators to be specified using special-purpose "auxiliary" annotations that control the parser's disambiguation process, Pappy does not provide such a feature because the notion of "disambiguation" does not have the same meaning in TDPL as it does for CFGs, and it is not obvious how some appropriate equivalent could be expressed cleanly in the TDPL paradigm. At any rate, it is not that cumbersome to express each precedence level as a separate nonterminal.

The remaining definitions for statements, declarations, and related constructs, are all straightforward translations from the Java language specification, and therefore are not

```
TypeSpec :: DeclType =
          t:TypeName d:Dims                     -> {DtArray t d}
        / TypeName

TypeName :: DeclType =
          PrimitiveType
        / n:QualifiedName                       -> {DtName n}

PrimitiveType :: DeclType =
          "byte":Word                           -> {DtByte}
        / "short":Word                          -> {DtShort}
        / "char":Word                           -> {DtChar}
        / "int":Word                            -> {DtInt}
        / "long":Word                           -> {DtLong}
        / "float":Word                          -> {DtFloat}
        / "double":Word                         -> {DtDouble}
        / "boolean":Word                        -> {DtBoolean}

QualifiedName :: {[Identifier]} =
          i:Identifier is:(".":Sym i:Identifier -> i)*  -> {i : is}

DimsOpt :: Int =
          d:Dims                                -> d
        /                                       -> {0}

Dims :: Int =
          "[":Sym "]":Sym d:Dims                -> {d+1}
        / "[":Sym "]":Sym                       -> {1}
```

Figure 4-11: Parsing rules for Java type expressions

```
Expression :: Expression =
          l:CondExpr op:AssignmentOperator r:Expression -> {ExpBinary op l r}
        / CondExpr

AssignmentOperator :: String =
          "=":Sym                                       -> {"="}
        / "+=":Sym                                      -> {"+="}
        / "-=":Sym                                      -> {"-="}
        / ...

... (other precedence levels) ...

PostfixExpr :: Expression =
          l:PostfixExpr "[":Sym r:Expression? "]":Sym   -> {ExpArray l r}
        / l:PostfixExpr a:Arguments                     -> {ExpCall l a}
        / l:PostfixExpr ".":Sym r:PrimExpr              -> {ExpSelect l r}
        / l:PostfixExpr ".":Sym "class":Word            -> {ExpDotClass l}
        / l:PostfixExpr "++":Sym                        -> {ExpPostfix "++" l}
        / l:PostfixExpr "--":Sym                        -> {ExpPostfix "--" l}
        / PrimExpr

PrimExpr :: Expression =
          l:Literal                                     -> {ExpLiteral l}
        / i:Identifier                                  -> {ExpIdent i}
        / "(":Sym e:Expression ")":Sym                  -> e
        / ...
```

Figure 4-12: Parsing rules for Java expressions

discussed further here. For more details please see the full parser specification available on-line.

4.4 Internal Grammar Representations and Transformations

After accepting a parser specification and rewriting any repetition operators and left-recursive definitions, Pappy uses a three-stage "back-end" pipeline to produce a working parser from the specification. The first back-end stage simplifies the grammar using a variety of local and global optimizations. The second stage analyzes the grammar to determine which nonterminals should be memoized in the parser's `Derivs` tuple, and which can be more efficiently implemented in terms of simple functions. Finally, in the last stage, the code generator uses the simplified grammar and the results of memoization analysis to write the parser. The remainder of this section describes the important details of each these stages.

4.4.1 Grammar Simplification

The number of nonterminals that must be memoized in a packrat parser's `Derivs` tuple is of critical importance to the parser's space consumption, since the parser must store an instance of this tuple for every character position in the input string. Experimentation reveals that, at least in Haskell, the number of nonterminals is also an important factor in the packrat parser's performance. For this reason, Pappy implements a number of optimizations designed primarily to reduce the number of nonterminals in the grammar, and secondarily to reduce the complexity of the rules themselves whenever possible.

Since the application of one optimization may create further optimization opportunities, Pappy invokes all of these optimizations repeatedly until none of them can be applied anymore. This process terminates because all of the optimizations strictly reduce the size of the grammar, where the grammar's "size" is measured first by the number of nonterminals and second by the aggregate size of the rules in the definitions.

Peephole Optimizations

A simple "peephole optimizer" performs two simple local transformations to reduce the complexity of rules in the parser specification:

1. Redundant sequencing operators containing only one component, and choice operators containing only one alternative, are eliminated.

2. Sequence operators nested directly within other sequence operators, and choice operators nested directly within other choice operators, are "flattened." For example, '$r_1/(r_2/r_3)/r_4$' is rewritten '$r_1/r_2/r_3/r_4$'.

In addition, to improve lexical analysis performance, the peephole optimizer detects and "left-factors" constructs consisting of a choice between several literal characters or strings, such as the rule for `SymChars` in the Java parser specification above. When multiple alternatives start with common characters, the common prefix is factored out so that it needs to be tested only once in the sequence. For example, the rule ('>=' / '>' / '<=' / '<') would be rewritten in the form ('>' ('=' / ()) / '<' ('=' / ())).

In addition, rather than implementing these factored constructs in terms of normal sequencing and choice, Pappy's optimizer uses a special internal *switch* operator. This operator cannot be invoked directly by the user in a parser specification, but only generated internally through the left-factoring optimization. The use of the switch operator for implementation of these "prefix recognition trees" preserves the knowledge that all of the "branches" of the tree are disjoint: if one prefix matches the input stream, then none of the other prefixes need to be tested even if the subrule for the matched prefix subsequently fails. The result is that the recognition of simple operators and other literal symbols used in a language can be implemented directly and efficiently in terms of nested Haskell `case` expressions. For example, with this optimization the `SymChars` definition in the Java parser results in in a Haskell expression of this form:

```
javaParseSymChars :: JavaDerivs -> Result JavaDerivs (String)
javaParseSymChars d =
        case javaChar d of
          Parsed '>' d3 _ ->
            case javaChar d3 of
              Parsed '>' d5 _ ->
                case javaChar d5 of
                  Parsed '>' d7 _ ->
                    case javaChar d7 of
                      Parsed '=' d9 _ ->
                        Parsed (">>>=") d9 (...)
                      _ ->
                        Parsed (">>>") d7 (...)
                  Parsed '=' d7 _ ->
                    Parsed (">>=") d7 (...)
                  _ ->
                    Parsed (">>") d5 (...)
              Parsed '=' d5 _ ->
                Parsed (">=") d5 (...)
              _ ->
                Parsed (">") d3 (...)
          Parsed '<' d3 _ ->
            ...
```

Redundant Rule Elimination

When repetition operators are applied to the same subrule in multiple places in the user's original parser specification, the rewriting of these operators creates multiple nonterminals with equivalent definitions. Therefore, Pappy's optimizer scans for and combines any redundant nonterminals created in this way, as well as any that the original specification may have contained. The structural equivalence analysis is relatively primitive, eliminating only redundant nonterminals that can be identified one at a time. Multiple redundant *sets* of mutually recursive nonterminals are not detected, for example, but this limitation is not a problem in practice since the rewriting of repetition operators does not produce such sets and they are unlikely to occur naturally in typical grammars.

Inlining

Grammar specifications often have many nonterminals that have simple, non-recursive definitions. Pappy attempts to eliminate such nonterminals through the classic process of *inlining*: folding copies of their definitions directly into all of their call points in the grammar. As with inlining in general-purpose programming languages, the inlining of definitions in a Pappy grammar must be done cautiously to avoid a blow-up in the size and complexity of the grammar and the resulting parser. For this reason, Pappy inlines only two kinds of definitions: "tiny" definitions containing at most two operators on their right-hand side, and definitions that are only referenced once in the entire grammar. Experience demonstrates that even inlining conservatively in this way reduces the number of nonterminals in a grammar substantially, and can "open the door" for other local optimizations to be applied to the inlined parsing rules.

4.4.2 Memoization Analysis and Virtual Inlining

Although only the smallest nonterminal definitions can be safely inlined through direct substitution in the parser specification, many more nonterminals can be eliminated from the `Derivs` tuple through *virtual inlining.* A virtually-inlined nonterminal remains part of the grammar, and corresponds to a separate Haskell parsing function in the generated parser. A virtually-inlined nonterminal has no corresponding component in the `Derivs` tuple, however. Instead, at every location in the parser where that nonterminal is invoked, the parsing function for the nonterminal is used directly instead of a `Derivs` data-accessor function. In effect, virtually-inlined functions are simply "de-memoized."

In order to ensure that the packrat parser's linear time guarantee is not invalidated by this transformation, only nonterminals that *could* be inlined through direct substitution are ever selected for virtual inlining. Specifically, virtually-inlined definitions cannot be recursive, either directly or indirectly through other virtually-inlined definitions. As long as all recursion in the grammar is broken through memoization, each virtually-inlined definition still only accesses a fixed number of other "cells" in the parsing structure and performs a fixed number of primitive parsing operations in order to compute each result, though these constants could be increased by virtual inlining.

Although much larger definitions can be virtually-inlined safely than can be directly inlined, caution is still required to prevent the possibility of too much redundant computation. Pappy will therefore not virtually-inline any nonterminal whose definition consists of more than about 25 operators, *including* the size of the definitions of any other nonterminals that are invoked by the nonterminal under consideration and have already been selected for virtual-inlining.

Even with these limitations, judicious virtual inlining can substantially reduce the number of memoized nonterminals and hence the size of a packrat parser's `Derivs` tuple, leading to reduced storage consumption and increased performance due to lower garbage collection overhead. For example, the packrat parser generated for the example Java grammar above contains 90 nonterminals after rewriting and simplification, but only 51 of those nonterminals are memoized.

4.4.3 Code Generation

The final step in Pappy's internal pipeline is writing the Haskell code for the parser. Although code generation is a relatively straightforward process directly following the structure of the internal parser specification, and will not be described in detail, there is one important feature of the code generator that is worthy of note.

In order to reduce the space consumption further, Pappy's code generator uses the trick described in Section 3.3.3 of breaking the `Derivs` structure into two levels. The "top-level" `Derivs` tuple consists of `dvChar`, `dvPos`, and a set of "sub-tuples," each of which in turn contains the result components for a number of nonterminals. Pappy attempts to balance the number of sub-tuples in the top-level `Derivs` tuple against the number of components in each sub-tuple, and distributes the nonterminals in the specification evenly among the sub-tuples. No special attempt is made to "cluster" nonterminals in some appropriate way, but nonterminals are assigned to sub-tuples in the order in which they appear in the parser specification, preserving the natural proximity of of related nonterminals that tends to be present in specifications written by humans. In the Java parser specification, for example, all of the lexical analysis rules are clustered toward the beginning of the specification, with the result that their corresponding components in the `Derivs` tuple are memoized in the first one or two sub-tuples. Since the lexical analysis rules always consume complete tokens, the parser usually evaluates results in only one of these sub-tuples at "internal" character positions after the beginning of a multi-character token.

Experimentation reveals that the use of this two-level `Derivs` structure is critical in packrat parsers for Haskell, not only for space consumption but for performance. Using a simple "flat" `Derivs` tuple in the example Java parser makes the parser *over an order of magnitude slower* under GHC, the Glaskow Haskell Compiler. This effect is caused not by the size of the `Derivs` tuple itself, but by the size of the top-level `parse` function, which the lazy evaluator must "instantiate" once for each input position. This effect is is probably specific to Haskell and other languages with a similar lazy evaluation model; space consumption behavior is likely to be substantially different in other target programming languages.

Chapter 5

Experimental Results

This chapter presents preliminary experimental results to demonstrate the practicality of packrat parsing. These experiments study the performance of hand-coded and automatically-generated packrat parsers for the Java[1] programming language. Java was chosen because it has a rich and complex grammar, but nevertheless adopts a fairly clean syntactic paradigm. For example, Java does not require the parser to maintain state about declared types as C and C++ parsers do, or to perform special processing between lexical and hierarchical analysis as Haskell's layout scheme requires.

5.1 The Parsers

The experiments use three different functionally equivalent packrat parsers for Java. Apart from a trivial preprocessing stage to canonicalize line breaks and Java's Unicode escape sequences, lexical analysis for all three parsers is fully integrated into the parser. All three parsers use the technique described in Section 3.3.3 of splitting the `Derivs` tuple into two levels, in order to increase modularity and reduce space consumption.

The first two parsers are manually written in Haskell, using the methods described in Chapter 3. One of these parsers uses monadic combinators exclusively throughout the parser, whereas the other parser uses primitive pattern matching directly to code the parsing functions for the most frequently used lexical constructs such as keywords, operators, identifiers, and integer literals. Both parsers use monadic combinators, however, to construct all higher-level parsing functions. The second, "hybrid" monadic/pattern-matching parser was created in order to study the practical cost of using monadic combinators in packrat parsers. The third parser used in the experiments is the parser generated by Pappy from the parser specification presented in Section 4.3.

The test suite used in all of these experiments consists of 60 unmodified Java source files from the Cryptix library[2]. Cryptix was chosen primarily because it is well-known, freely available, and includes a substantial number of relatively large Java source files. (Java source files are small on average because the compilation model encourages programmers to place each class definition in a separate file.)

In the following sections, we study two aspects of the performance of these three Java parsers. First, we examine their space utilization (maximum heap size), in order to address

[1] Java is a trademark of Sun Microsystems, Inc.

[2] `http://www.cryptix.org/`

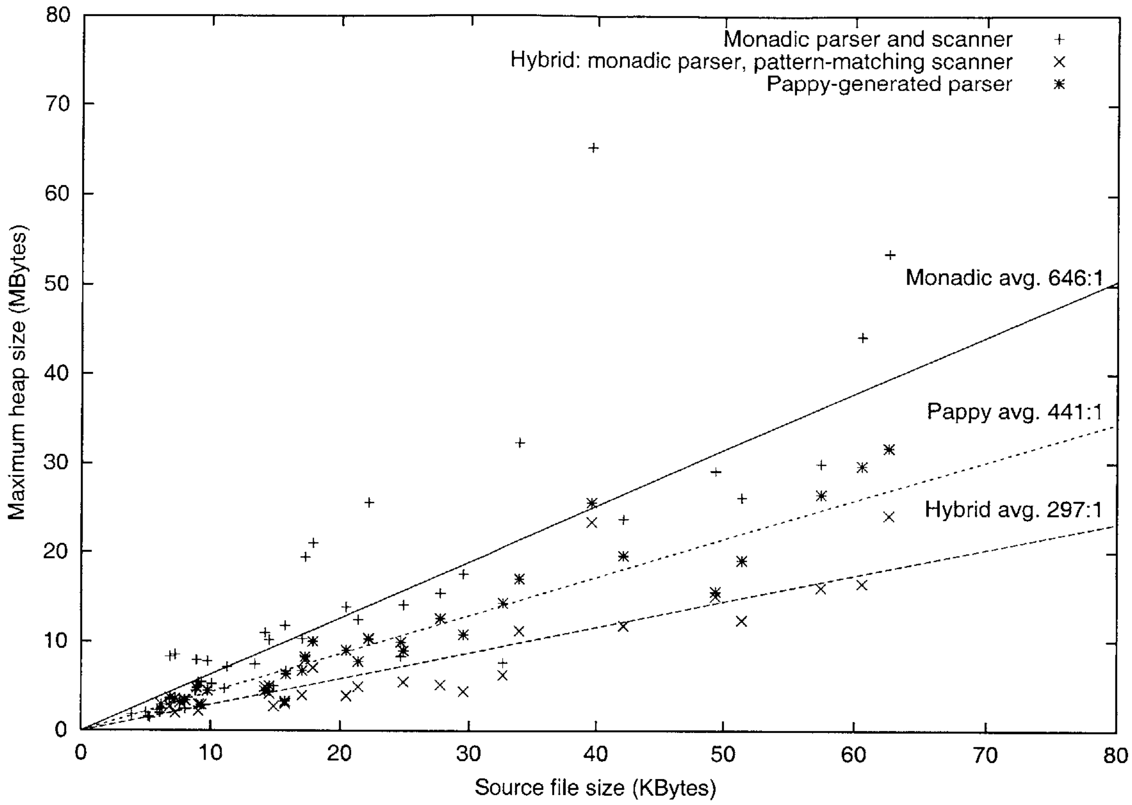

Figure 5-1: Maximum heap size versus input size

the important question of whether a packrat parser's result matrix consumes too much storage to make the algorithm practical. Second, we measure the absolute performance of the three parsers, in order to provide a feel for their general performance, and to compare the effects of the different methods of constructing packrat parsers.

5.2 Space Efficiency

In the first set of experiments, the three Java parsers were compiled with the Glasgow Haskell Compiler[3] version 5.04, with optimization and profiling enabled. GHC's heap profiling system was used to measure live heap utilization, which excludes unused heap space and collectible garbage when samples are taken.

Figure 5-1 shows a plot of each parser's maximum live heap size against the size of the input files being parsed. Because some of the smaller source files were parsed so quickly that garbage collection never occurred and the heap profiling mechanism did not yield any samples, the plot includes only 47 data points for the hand-coded parser that uses monadic combinators exclusively, 27 data points for the hybrid parser using a combination of monadic combinators and pattern matching, and 33 data points for the packrat parser generated by Pappy. One data point, for a Java source file 130 KB in size, was included in the analysis but not shown on the graph, because compressing the horizontal axis sufficiently to include it unacceptably obscures the more densely-packed portion of the graph to the left.

[3] `http://www.haskell.org/ghc/`

Averaged across the test suite, the fully monadic parser uses 646 bytes of live heap per byte of input, while the hybrid parser uses only 297 bytes of heap per input byte, and the Pappy-generated parser falls in the middle at 441 bytes of heap per input byte. In general, these results are encouraging: although packrat parsing can consume a substantial amount of space, a typical modern machine with 128KB or more of RAM should have no trouble parsing source files up to 100-200KB. Furthermore, even though the first two parsers use some iterative monadic combinators, which can break the linear time and space guarantee in theory, the space consumption of these parsers nevertheless appears to grow fairly linearly.

The use of monadic combinators clearly has a substantial penalty in terms of space efficiency. Modifying the parser to use direct pattern matching alone may yield further improvement, though the degree is difficult to predict since the cost of lexical analysis often dominates the rest of the parser. The parser generated by Pappy uses direct pattern matching exclusively in its parsing functions, but due to its many other differences from the hand-coded parsers, its performance cannot be considered directly indicative of the result of switching to pure pattern matching.

Although the Pappy-generated parser consumes more storage than the hand-coded pattern-matching parser, it is considerably more space-efficient than the monadic parser. The space consumption of the Pappy-generated parser is, furthermore, noticeably more *consistent* than either of the hand-coded parsers. Across the data points produced by this test suite, the standard deviation of the monadic parser from its average heap/input ratio is 301, the standard deviation for the hybrid parser is 106, and the standard deviation for the Pappy parser is 87. This difference is directly apparent from the graph, on which the points for the Pappy-generated parser (the '$*$'s) can be seen to follow the middle line, representing the average for that parser, much more closely than the data points for the two hand-coded parsers follow their corresponding lines. This improvement in predictability can be attributed to the fact that the parsing functions in the Pappy parser contain no hidden recursion as the other two parsers do. Because the repetition operators in the Pappy specification are rewritten in terms of constant-time primitives before the parser is generated, the Pappy parser can guarantee asymptotically linear execution time whereas the hand-coded parsers containing hidden recursion cannot. This result indicates that avoiding hidden recursion can indeed be important in practice, though not necessarily critical.

5.3 Parsing Performance

The second experiment compares the absolute execution time of the three packrat parsers. For this test, the parsers were compiled by GHC 5.04 with optimization but without profiling, and timed on a 1.28GHz AMD Athlon processor running Linux 2.4.17. Only the 28 source files in the test suite larger than 10KB were included in this test, because the smaller files were parsed so quickly that the Linux `time` command did not yield adequate precision. Again, the data point for the one 130 KB source file in the test suite was included in the analysis but omitted from the graph for purposes of clarity. Figure 5-2 shows the resulting execution times plotted against source file size. On these inputs the fully monadic parser averaged 26.3 KBytes per second with a standard deviation of 8.9 KB/s, while the hybrid parser averaged 52.1 KB/s with a standard deviation of 17.1 KB/s, and the Pappy parser averaged 34.4 KB/s, with a standard deviation of 5.9 KB/s.

In these tests, the parser generated by Pappy once again comes in at an intermediate level between the two hand-coded parsers, but its performance is noticeably more linear.

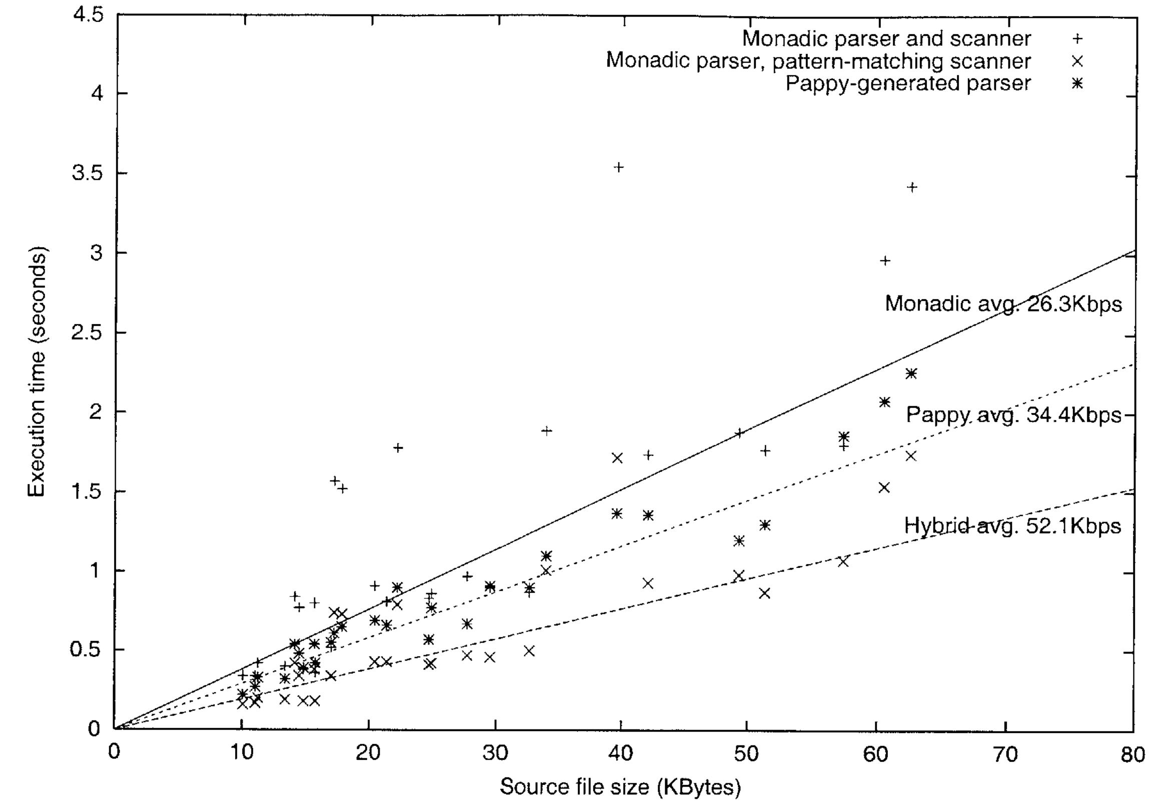

Figure 5-2: Execution time versus input size

Whereas the standard deviation in the performance of each of the hand-coded parsers represents about 33% of their respective averages, the standard deviation of the Pappy parser represents only 17% of its average performance. These numbers suggest that packrat parsing can provide dependable linear-time parsing of real source files in complex languages, especially when an automatic parser generator such as Pappy is used to rewrite repetition operators in the parser specification.

Chapter 6

Related Work

This chapter relates the TDPL notation developed in this thesis and the packrat parsing algorithm to relevant prior work. First we briefly review the historical background of TDPL and the theoretical results that have been derived. Second, we informally explore the relationship between packrat parsing and the ubiquitous LL and LR parsing algorithms. Next, we examine *noncanonical* extensions to the LR paradigm, which address some of the same limitations in LR that packrat parsing does, though in a different way. Finally, we discuss general CFG parsing algorithms and practical parser generators based on them.

6.1 TDPL Background

Birman and Ullman [4] first developed the formal properties of parsing algorithms with localized backtracking. This work was refined by Aho and Ullman [3] and classified as "top-down limited backtrack parsing," in reference to the restriction that each parsing function can produce at most one result and hence backtracking is localized to that function.

Two specific formal models of limited backtrack parsing were developed. The first model, referred to as the "TMG recognition scheme" (TS) in the original work and later renamed "top-down parsing language" (TDPL), was inspired by TMG, an early syntax-directed compiler generation system [15]. The second formal model, originally referred to as "generalized TS" (gTS) and later as "generalized TDPL" (GTDPL), was inspired by META II, another early compiler-compiler [20]. To keep the distinction clear between the overall TDPL notational and parsing paradigm used in this thesis, and the specific formal systems developed in the prior theoretical work, the latter formal systems will be referred to here as TS/TDPL and gTS/GTDPL respectively.

6.1.1 The TS/TDPL Formal System

In the formal model of TS/TDPL, the definitions of nonterminals are restricted to the following two primitive forms:

1. $A \leftarrow BC/D$, where A, B, C, and D are all nonterminals.
2. $A \leftarrow a$, where A is a nonterminal and a is either a terminal (character), the empty string ϵ, or a distinguished symbol f representing failure.

The first form of rule, $A \leftarrow BC/D$, embodies both sequencing and ordered choice. In its operational interpretation, B is first invoked. If B succeeds, then C is invoked on the

Description	Extended TDPL	Formal TS/TDPL
Empty string	$A \leftarrow ()$	$A \leftarrow \epsilon$
Terminal	$A \leftarrow a$	$A \leftarrow a$
Nonterminal	$A \leftarrow B$	$A \leftarrow BE/F$ $E \leftarrow \epsilon$ $F \leftarrow f$
n-ary sequence	$A \leftarrow r_1 r_2 \ldots r_n$	$A \leftarrow BC/F$ $B \leftarrow r_1$ $C \leftarrow r_2 \ldots r_n$ $F \leftarrow f$
n-ary ordered choice	$A \leftarrow r_1/r_2/\ldots/r_n$	$A \leftarrow BE/C$ $B \leftarrow r_1$ $C \leftarrow r_2/\ldots/r_n$ $E \leftarrow \epsilon$
Greedy repetition	$A \leftarrow r*$	$A \leftarrow BA/E$ $B \leftarrow r$ $E \leftarrow \epsilon$
Positive repetition	$A \leftarrow r+$	$A \leftarrow CB/F$ $B \leftarrow CB/E$ $C \leftarrow r$ $E \leftarrow \epsilon$ $F \leftarrow f$
Optional	$A \leftarrow r?$	$A \leftarrow BE/E$ $B \leftarrow r$ $E \leftarrow \epsilon$

Figure 6-1: Rules for rewriting extended TDPL definitions into formal TS/TDPL

input text unconsumed by B. If both B and C succeed, then the rule succeeds. However, if either B or C fail, then D is invoked at the original input position at which B was invoked, backtracking in the input string if necessary. The second form of the rule, $A \leftarrow a$, is used to match constant terminal symbols (characters), to produce unconditional success without consuming anything (ϵ), or to produce unconditional failure (f).

Any nonterminal definition in the extended TDPL notation used in this thesis can be reduced to the primitives supported by this formal model, provided that they do not contain syntactic predicate operators (& or !). Syntactic predicates appear to require the additional functionality of the gTS/GTDPL system, below. For reference, Figure 6-1 provides a set of rewrite rules that can be used to reduce extended TDPL notation to rules in the primitive TS/TDPL formal system. Many of the rules involve introducing new nonterminals into the grammar: any nonterminal letter appearing on the right side of a rewrite rule but not on the left is assumed to be a fresh nonterminal. In the rewrite rules containing variables appearing on both sides of the rule, representing parsing subrule expressions (r, r_1, r_2, etc.), the new definitions produced by the transformation may in turn have to be rewritten in order to decompose these subrule expressions. The primitive TS/TDPL grammars produced using these rules are by no means likely to be optimal.

Description	Extended TDPL	Formal gTS/GTDPL
TS/TDPL sequencing/choice	$A \leftarrow BC/D$	$A \leftarrow X[E,D]$ $X \leftarrow B[C,F]$ $E \leftarrow \epsilon$ $F \leftarrow f$
Not-followed-by predicate	$A \leftarrow\ !(r)$	$A \leftarrow B[F,E]$ $B \leftarrow r$ $E \leftarrow \epsilon$ $F \leftarrow f$
Followed-by predicate	$A \leftarrow \&(r)$	$A \leftarrow B[F,E]$ $B \leftarrow C[F,E]$ $C \leftarrow r$ $E \leftarrow \epsilon$ $F \leftarrow f$

Figure 6-2: Rules for rewriting formal TS/TDPL and extended TDPL syntactic predicates into formal gTS/GTDPL

6.1.2 The gTS/GTDPL Formal System

The gTS/GTDPL formal system uses primitive rules taking a slightly different form:

1. $A \leftarrow B[C,D]$, where A, B, C, and D are all nonterminals.

2. $A \leftarrow a$, where A is a nonterminal and a is a terminal, ϵ, or f.

Only the first form of rule differs from TS/TDPL. This form, though involving the same number of nonterminals as the sequencing/choice rule in the TS/TDPL system, is interpreted differently. Invoking A first causes B to be invoked. If B succeeds, then C is invoked starting at the position following the text consumed by B, and the result of C, whether success or failure, is used as the result for A. However, if B fails, then D is invoked instead at the same position at which B was called, and the result of the call to D becomes the result of A. The important point is that after B is invoked, *either* C or D is invoked, but never both. The gTS/GTPL rule '$A \leftarrow B[C,D]$' is functionally equivalent to the definition '$A \leftarrow B\ C\ /\ !(B)\ D$' in the extended TDPL notation of this thesis.

Birman and Ullman proved that any TS/TDPL grammar can be transformed into an equivalent gTS/GTDPL by rewriting the primitive sequence/choice rules in the TS/TDPL grammar using a rule such as the one shown in the first row of the table in Figure 6-2. Therefore, gTS/GTDPL is at least as powerful as TS/TDPL.

It appears that gTS/GTDPL is properly more powerful than TS/TDPL, though this conjecture has not yet been proven. The syntactic predicate operators of the extended TDPL notation used in this thesis can be rewritten in terms of GTDPL using the second and third rules shown in Figure 6-2. An example language that probably cannot be recognized by a well-formed TS/TDPL grammar is $\{a,b\}^* - \{a^n b^n | n > 0\}$. This language, consisting of all strings of 'a's and 'b's *except* those in which they form exactly matched pairs, is easily expressed by the following extended TDPL grammar:

$$\begin{array}{lcl} \text{S} & \leftarrow & !(\text{A EOF})\ (\texttt{a},\texttt{b})^* \\ \text{A} & \leftarrow & \texttt{a}\ \text{A}\ \texttt{b}\ /\ \texttt{a}\ \texttt{b} \\ \text{EOF} & \leftarrow & !(\text{any character}) \end{array}$$

gTS/GTDPL is more well-behaved than TS/TDPL in certain respects. It is possible to detect all potential "loop failures" caused by direct or indirect left recursion in a gTS/GTDPL grammar, and to rewrite the grammar into a form that recognizes the same language without causing loop failures. In contrast, there is strong evidence that eliminating loop failures in TS/TDPL grammars is impossible in general.

Birman and Ullman's left recursion elimination algorithm for gTS/GTDPL serves a different purpose from the one implemented in Pappy. The former algorithm rewrites the grammar so as to *fail gracefully* on strings in which the original grammar would get into an infinite loop, whereas the purpose of Pappy's left recursion elimination is to make the grammar *succeed* in a pragmatically useful way where it would otherwise fail. Pappy's left recursion elimination is intentionally *not* an equivalence-preserving transformation.

6.1.3 Relationship Between TDPL and CFGs

gTS/GTDPL is known to be powerful enough to recognize some non-context free languages. The language $\{\texttt{a}^n\texttt{b}^n\texttt{c}^n | n > 0\}$, for example, cannot be expressed by any CFG. The following parsing grammar recognizes this language, however:

$$\begin{array}{lcl} \text{S} & \leftarrow & \&(\text{A}\ \texttt{c})\ \texttt{a}+\ \text{B} \\ \text{A} & \leftarrow & \texttt{a}\ \text{A}\ \texttt{b}\ /\ \texttt{a}\ \texttt{b} \\ \text{B} & \leftarrow & \texttt{b}\ \text{B}\ \texttt{c}\ /\ \texttt{b}\ \texttt{c} \end{array}$$

This parsing grammar can be reduced to formal gTS/GTDPL using the rules above. The nonterminal S in this grammar first uses a syntactic predicate to check that the 'a's and 'b's are matched, without actually consuming them. The rule then skips past the 'a's and matches the 'b's with the 'c's.

It is also known that TDPL is powerful enough to simulate any push-down automaton, and can therefore recognize any LL(k) or LR(k) language.

Although not yet formally proven, it appears that the set of context-free languages and the set of languages expressible in TDPL are incomparable. For example, the pathological CFG presented earlier in Section 2.4.4 appears not to be expressible in TDPL:

$$\text{S} \rightarrow \text{a S a} \mid \text{a S b} \mid \text{b S a} \mid \text{b S b} \mid \text{a}$$

This CFG describes a language consisting of odd-length strings of 'a's and 'b's, in which the middle character is always an 'a'. The problem here is that in TDPL there is no way to "find the middle" of a homogeneous string containing no distinguishable syntactic "signposts," but the middle character must be found in this case in order to reject strings that have a 'b' at that position.

6.1.4 Practical TDPL Parsing

Birman and Ullman demonstrated the existence of a linear-time parsing algorithm for both TS/TDPL and gTS/GTDPL grammars, of the tabular variety described in Section 3.1.3. However, this linear-time algorithm was apparently never implemented, probably because computers at the time had much more limited RAM. Compilers in this period often had to operate in "streaming" fashion, generating output progressively as input text was consumed rather than keeping a complete intermediate representation in memory, so that large source files could be processed in near-constant space. Both of the compiler-compiler systems from which TDPL theory was inspired took the basic recursive descent approach; META II, in fact, did not even fully support backtracking as modeled by the formal system.

The formal TDPL work appears to have been largely neglected after its initial development, likely in part as a result of the excitement over the LR algorithms, which were newly-developed at the time and probably were seen as more practical. Adams [1] recently used TDPL in a modular language prototyping framework. In addition, many practical top-down parsing libraries and toolkits, including the popular ANTLR [17] and the PARSEC combinator library for Haskell [14], provide similar limited backtracking capabilities which the parser designer can invoke selectively in order to overcome the limitations of predictive parsing. In effect, by providing *ad hoc* backtracking support, many practical recursive descent parsers such as these effectively adopt the TDPL paradigm without explicitly recognizing it as such. However, these parsers still implement backtracking in the traditional recursive-descent fashion without memoization, creating the danger of exponential worst-case parse time, and thereby making it impractical in general to rely on backtracking as a substitute for prediction or to integrate lexical analysis with parsing.

6.2 LL and LR Parsing

Although LR parsing is commonly seen as "more powerful" than limited-lookahead top-down or LL parsing, the class of languages these parsers can recognize is the same [3]. As Pepper points out [18], LR parsing can be viewed simply as LL parsing with the grammar rewritten so as to eliminate left recursion and to delay all important parsing decisions as long as possible. The result is that LR provides more flexibility in the way grammars can be expressed, but no actual additional recognition power. For this reason, we will treat LL and LR parsers here as essentially equivalent.

6.2.1 Lookahead

The most critical practical difference between packrat parsing and LL/LR parsing is the lookahead mechanism. A packrat parser's decisions at any point can depend on all the text up to the end of the input string. Although the computation of an individual result in the parsing matrix can perform only a constant number of "basic operations," these basic operations include following forward pointers in the parsing matrix, each of which can skip over a large amount of text at once. Therefore, while LL and LR parsers can look ahead only a constant number of *terminals* in the input, packrat parsers can look ahead a constant number of *terminals and nonterminals* in any combination. This ability for parsing decisions to take arbitrary nonterminals into account is what gives packrat parsing its unlimited lookahead capability.

From a viewpoint of computational complexity, packrat parsing inherently requires, and takes advantage of, a more powerful computational model than LL or LR parsing. Whereas LL and LR parsers can be implemented by a simple stack machine or deterministic push-down automata [2], linear-time packrat parsing inherently requires a machine with random-access memory. Conversely, the claim that packrat parsing is a "linear-time" algorithm depends on the assumption that RAM can be accessed in constant time. With log-cost RAM, packrat parsing becomes $O(n \; log \; n)$.

To illustrate the difference in language recognition power between packrat parsing and LR, the following CFG expresses the language $\{\mathtt{a}^n\mathtt{b}^n | n > 0\} \cup \{\mathtt{a}^n\mathtt{b}^{2n} | n > 0\}$:

$$\begin{array}{lcl} \mathrm{S} & \longrightarrow & \mathrm{E} \mid \mathrm{F} \\ \mathrm{E} & \longrightarrow & \mathtt{a\ b} \mid \mathtt{a}\ \mathrm{E}\ \mathtt{b} \\ \mathrm{F} & \longrightarrow & \mathtt{a\ b\ b} \mid \mathtt{a}\ \mathrm{F}\ \mathtt{b\ b} \end{array}$$

This CFG is not LR(k) for any k, nor is there any other LR-class CFG that can recognize this language. Once an LR parser has encountered the boundary between the 'a's and 'b's in a string in this language, it must decide immediately whether to start reducing them by matching 'a's with 'b's one-to-one or by matching 'a's with pairs of 'b's. However, there is no way for the LR parser to make this decision until as many 'b's have been encountered as there were 'a's on the left-hand side.

This non-LR language can be recognized by a packrat parser by converting the above CFG directly into a TDPL grammar in the obvious way:

$$\begin{array}{lcl} \mathrm{S} & \longleftarrow & \mathrm{E} \,/\, \mathrm{F} \\ \mathrm{E} & \longleftarrow & \mathtt{a\ b} \,/\, \mathtt{a}\ \mathrm{E}\ \mathtt{b} \\ \mathrm{F} & \longleftarrow & \mathtt{a\ b\ b} \,/\, \mathtt{a}\ \mathrm{F}\ \mathtt{b\ b} \end{array}$$

A packrat parser for this grammar operates in a speculative fashion, potentially reducing the complete input string using both E and F *in parallel*. The ultimate decision between E and F is effectively delayed until the entire input string has been parsed, where the decision is merely a matter of checking which of E or F (if either) succeeded. Mirroring the above grammar left to right does not change the situation, making it clear that the difference is not merely a side-effect of the fact that an LR parser's decisions depend primarily on "leftward" context with limited "rightward" lookahead, whereas a packrat parser's decisions depend on unlimited "rightward" lookahead but have no leftward context.

6.2.2 Grammar Composition

The limitations of LR parsing due to fixed lookahead are frequently felt when designing parsers for practical languages, and many of these limitations stem from the fact that LL and LR grammars are not cleanly *composable*. For example, the following grammar represents a simple language with expressions and assignment, which only allows simple identifiers on the left side of an assignment:

$$\begin{array}{lcl} \mathrm{S} & \longleftarrow & \mathrm{R} \mid \mathrm{ID}\ \text{'='}\ \mathrm{R} \\ \mathrm{R} & \longleftarrow & \mathrm{A} \mid \mathrm{A\ EQ\ A} \mid \mathrm{A\ NE\ A} \\ \mathrm{A} & \longleftarrow & \mathrm{P} \mid \mathrm{P}\ \text{'+'}\ \mathrm{P} \mid \mathrm{P}\ \text{'-'}\ \mathrm{P} \\ \mathrm{P} & \longleftarrow & \mathrm{ID} \mid \text{'('}\ \mathrm{R}\ \text{')'} \end{array}$$

If the symbols ID, EQ, and NE are terminals, representing tokens produced by a separate lexical analysis phase, then an LR(1) parser has no trouble with this grammar. However, if we try to integrate this tokenization into the parser itself with the following simple rules, the grammar is no longer LR(1):

```
ID  ←  ’a’ | ’a’ ID
EQ  ←  ’=’ ’=’
NE  ←  ’!’ ’=’
```

The problem is that after scanning an identifier, an LR parser must decide immediately whether it is a primary expression or the left-hand side of an assignment, based only on the immediately following token. But if this token is an ‘=’, the parser has no way of knowing whether it is an assignment operator or the first half of an ‘==’ operator. In this particular case the grammar could be parsed by an LR(2) parser. In practice LR(k) and even LALR(k) parsers are uncommon for $k > 1$.

Even when lexical analysis is separated from parsing, the limitations of LR parsers often surface in other practical situations, frequently as a result of seemingly innocuous changes to an evolving grammar. For example, suppose we want to add simple array indexing to the language above, so that array indexing operators can appear on either the left or right side of an assignment. One possible approach is to add a new nonterminal, L, to represent left-side or "lvalue" expressions, and incorporate the array indexing operator into both types of expressions as shown below:

```
S  ←  R | L ‘=’ R
R  ←  A | A EQ A | A NE A
A  ←  P | P ‘+’ P | P ‘-’ P
P  ←  ID | ‘(’ R ‘)’ | P ‘[’ A ‘]’
L  ←  ID | ‘(’ L ‘)’ | L ‘[’ A ‘]’
```

Even if the ID, EQ, and NE symbols are again treated as terminals, this grammar is not LR(k) for any k, because after the parser sees an identifier it must immediately decide whether it is part of a P or L expression, but it has no way of knowing this until any following array indexing operators have been fully parsed. Again, a packrat parser has no trouble with this grammar because it effectively evaluates the P and L alternatives "in parallel" and has complete derivations to work with (or the knowledge of their absence) by the time the critical decision needs to be made.

In general, grammars for packrat parsers are composable because the lookahead a packrat parser uses to make decisions between alternatives can take account of arbitrary nonterminals, such as EQ in the first example or P and L in the second. Because a packrat parser does not give "primitive" syntactic constructs (terminals) any special significance as an LL or LR parser does, any terminal or fixed sequence of terminals appearing in a grammar can be substituted with a nonterminal without "breaking" the parser.

6.3 Noncanonical Bottom-Up Parsing Algorithms

Substantial efforts have been made to develop practical extensions to the LR class of parsing algorithms that solve some of the problems described above without giving up its linear time guarantee. Such algorithms are known as *noncanonical bottom-up algorithms* because they operate in the same general bottom-up fashion as an LR parser, successively reducing phrases in the input to their corresponding nonterminals, but unlike LR they can perform reductions on phrases *other than the leftmost one* in the stream at any given point. The effect of this modification is that under certain conditions, nonterminals as well as terminals can be used in lookahead decisions.

6.3.1 NSLR(1)

The only noncanonical bottom-up algorithm that is known to be practical is the NSLR(1) algorithm created by Tai [22]. The following example CFG by Tai expresses the language $\{\texttt{a}^n\texttt{b}^n\texttt{c} \mid n > 0\} \cup \{\texttt{a}^n\texttt{b}^{2n}\texttt{d} \mid n > 0\}$, which is not LR($k$) for any k but is NSLR(1):

$$\begin{array}{lcl}
\mathrm{S} & \rightarrow & \mathrm{E}\ \texttt{c} \mid \mathrm{F}\ \texttt{d} \\
\mathrm{E} & \rightarrow & \texttt{a}\ \mathrm{B} \mid \texttt{a}\ \mathrm{E}\ \mathrm{B} \\
\mathrm{B} & \rightarrow & \texttt{b} \\
\mathrm{F} & \rightarrow & \texttt{a}\ \mathrm{B'}\ \mathrm{B'} \mid \texttt{a}\ \mathrm{F}\ \mathrm{B'}\ \mathrm{B'} \\
\mathrm{B'} & \rightarrow & \texttt{b}
\end{array}$$

However, this grammar is also indicative of the limitations of the algorithm: if the seemingly redundant nonterminals B and B' are replaced with the simple terminal 'b', then the grammar is no longer NSLR(1). The reason for this limitation is that, like an LR(1) parser, after the boundary between the 'a's and the 'b's in the input string is found, the NSLR(1) parser must decide *with only one symbol of lookahead* whether to start reducing the matched 'a's and 'b's using nonterminal E or F. With the grammar shown above in which the distinct nonterminals B and B' represent "a 'b' to be used in E" and "a 'b' to be used in F" respectively, the NSLR(1) algorithm first reduces the last 'b' in the string to a B or a B' depending on whether the lookahead character is a 'c' or a 'd'. Then the parser reduces the next-to-last 'b' to a B or a B' depending on whether the lookahead symbol for *that* phrase (the 'b' just reduced) is a B or a B', and so on. Once all the 'b's are reduced, the algorithm can then start reductions via E or F. Without the B and B' nonterminals, however, the NSLR(1) algorithm has no way to "transmit" the information that the last character in the string is a 'c' or a 'd' back to the boundary between the 'a's and 'b's where it is needed to make the choice between E and F.

In contrast, a packrat parser can speculatively reduce instances of E and F and delay the decision between them until their results are needed by S. For this reason, the example language can be recognized by a packrat parser with a TDPL grammar constructed without the redundant B or B' nonterminals:

$$\begin{array}{lcl}
\mathrm{S} & \leftarrow & \mathrm{E}\ \texttt{c} \mid \mathrm{F}\ \texttt{d} \\
\mathrm{E} & \leftarrow & \texttt{a}\ \texttt{b} \mid \texttt{a}\ \mathrm{E}\ \texttt{b} \\
\mathrm{F} & \leftarrow & \texttt{a}\ \texttt{b}\ \texttt{b} \mid \texttt{a}\ \mathrm{F}\ \texttt{b}\ \texttt{b}
\end{array}$$

Despite this limitation, NSLR(1) has been shown by Salomon and Cormack [19] to be powerful enough to implement some types of "scannerless parsers," in which the lexical analyzer is integrated into the parser. Achieving this goal, however, involved augmenting the context-free grammar with a set of restrictive disambiguation rules of two kinds. First, *exclusion rules* serve a comparable function to the negative syntactic predicate ('!') operator in TDPL, by disallowing certain derivations that would otherwise be allowed. Exclusion rules are used by Salomon and Cormack to prevent reserved words from being recognized as identifiers, for example. Second, *adjacency restriction* rules disallow specific pairs of nonterminals from appearing directly adjacent to each other in the input string. This rule is used to handle the problem discussed in Section 2.4.1 of expressing the notion that whitespace is optional most of the time but mandatory between identifiers and keywords. In general the '!' operator in TDPL can be used to provide the functionality of adjacency rules as well: for example, 'Foo !(Bar)' matches Foo as long as it is not immediately followed by Bar. In the case of handling whitespace, however, it is not necessary to use syntactic predicates because of TDPL's inherent support for longest match parsing.

It is unknown whether there are NSLR(1) languages that cannot be recognized by a packrat parser: i.e., whether packrat parsing is more expressive than NSLR(1) or the two classes of languages are incomparable. Packrat parsing is clearly less restrictive of rightward lookahead, but NSLR(1) can also take leftward context into account when making reduction decisions.

In practice, NSLR(1) is probably more space-efficient, but packrat parsing is simpler and cleaner. As with LR grammars, the NSLR grammars represent a mathematically complicated subclass of the context-free grammars, which is unlikely to be closed under composition as TDPL rules are.

6.3.2 Bounded Context Parsable Grammars

The Bounded Context Parsable (BCP) grammars [27] represent the largest known class of languages that is decidable and can be parsed by a noncanonical bottom-up parser in linear time. Unfortunately, although a parsing algorithm exists for BCP grammars, it has never been put into practice because it requires an unmanageably large table of "parsing contexts." Since BCP grammars were developed and explored in 1972, it is possible that BCP grammar parsing would be manageable on modern machines, in the same way that tabular GTDPL parsing has become practical. However, even if BCP parsing has become practical, it still suffers from the limitation discussed above that a conventional bottom-up parser cannot make speculative reductions. For example, although the example language discussed in the last section is BCP, the language expressed by this simpler variant is not:

$$\begin{array}{lcl} \mathrm{S} & \rightarrow & \mathrm{E} \mid \mathrm{F} \\ \mathrm{E} & \rightarrow & \mathtt{a}\ \mathtt{b} \mid \mathtt{a}\ \mathrm{E}\ \mathtt{b} \\ \mathrm{F} & \rightarrow & \mathtt{a}\ \mathtt{b}\ \mathtt{b} \mid \mathtt{a}\ \mathrm{F}\ \mathtt{b}\ \mathtt{b} \end{array}$$

Without the trailing c or d in the former example, a noncanonical bottom-up parser has *no* syntactic markers anywhere in the input that allow it to determine whether to start reducing via E or F. The only way for a parser to distinguish the two alternatives of S in this case is to perform actual reductions of E and F in parallel, successively matching a's with single b's and pairs of b's until one of the branches of exploration fails.

6.3.3 Other Noncanonical Algorithms

Szymanski and Williams [21] developed a general formal model for bottom-up parsing algorithms and used it to analyze various classes of noncanonical bottom-up parsable grammars including BCP. The broadest of these classes in terms of language recognition power, the LR(k, ∞) grammars, consists of all unambiguous CFGs in which every sentential form (i.e., every possible "state" a bottom-up parser could arrive at) has *some* reducible phrase that can be uniquely distinguished by examining all the symbols to the left of the phrase and the first k symbols (terminals or nonterminals) to the right. Even this class of languages, in which membership is undecidable, does not include the language described by the example grammar above. In any string of the language serving as the "initial state" for the bottom-up parser, the only reducible phrase is the innermost 'ab' or 'abb', and which of these reductions to start with can only be determined by counting all the symbols to the left and to the right of the phrase.

Therefore, even though the expressiveness of GTDPL and noncanonical bottom-up algorithms has not been compared formally, it appears that all algorithms that fit into the non-speculative reduction model share a common limitation that GTDPL grammars do not. It is not known whether there exist bottom-up parsable languages that are not recognizable with GTDPL.

6.4 General CFG Parsing Algorithms

A recursive descent parser in a functional language can be built to recognize any context-free language by making the parsing functions return a list of results and allowing unrestricted backtracking [26, 10, 7]. However, this approach risks exponential worst-case execution time in general and makes useful error detection much more difficult.

More efficient general parsing algorithms for CFGs exist, such as those of Earley [3] and Tomita [23], which can handle arbitrary context-free grammars including ambiguous ones. Earley's algorithm, the asymptotically fastest known, runs in $O(n^3)$ on arbitrary CFGs and in $O(n^2)$ on unambiguous CFGs. The tabular structures used in Earley's algorithm bear some resemblance to the tabular result matrix constructed by a packrat parser. Due to the greater difficulty of the general CFG parsing problem, however, Earley's algorithm must store *sets* of results or "items" in each "cell" in the table, rather than merely one result per cell as a packrat parser does, leading to the super-linear parse time of Earley's algorithm.

SDF, a practical parser generation system, uses generalized CFG parsing based on Tomita's Generalized LR (GLR) algorithm to support parsing with integrated lexical analysis [25, 24]. Due to the limited expressiveness of pure CFG notation, SDF provides extensions allowing useful disambiguation rules of various kinds to be declared. The parser uses these disambiguation rules to trim the set of alternative parse trees produced by the parser both during and after the parsing process. Among these forms of disambiguation rules are the adjacency restrictions and exclusion rules introduced by Salomon and Cormack in NSLR(1) specifically for lexical analysis. In addition, SDF supports rules for specifying the precedence and associativity of binary operators, and rules enabling the parser to choose a "default" among multiple alternative parse trees when a ambiguity occurs.

Although the disambiguation rules provided by SDF provide much of the same practical expressive power as TDPL, this expressive power is achieved at the cost of linear-time parsing. Furthermore, although these disambiguation rules are "declarative" in the sense of expressing high-level intentions concisely, these declarations seem more consistent with the TDPL paradigm than the CFG paradigm because they essentially represent direct instructions to the SDF parser. A disambiguation rule inherently describes how a language is *read*, rather than how a language is *written*: in general it is not at all straightforward to generate strings in a language automatically from a CFG annotated with disambiguation rules, as it is for a pure CFG. If we must cross partway into the TDPL paradigm anyway in order to gain the expressiveness we seek, it becomes questionable whether the CFG paradigm is the right starting point.

Chapter 7

Conclusion and Future Work

In this thesis we have explored *top-down parsing language* (TDPL), a powerful notation for the specification of language syntax, and *packrat parsing*, a practical algorithm for parsing any TDPL grammar in linear time.

7.1 TDPL

While TDPL was originally created as a formal model for top-down parsers with backtracking capability, this thesis has developed an extended TDPL notation that serves as a viable and compelling alternative to traditional context-free grammars (CFGs) for describing the syntax of programming languages and other machine-readable languages. The fundamental difference between TDPL and CFG notation is that while a CFG specifies how strings in a language are *written*, TDPL specifies how strings in a language are *read.*

Many common syntactic idioms that are impossible to represent concisely in a pure CFG are easily expressed in TDPL. For example, TDPL is naturally oriented toward longest-match disambiguation, which is pervasive at the lexical level of most programming languages and frequently used in higher-level syntax as well. More sophisticated decision rules including general *syntactic predicates* are also supported directly by TDPL. In contrast with CFG notation, it is possible to express precisely the complete syntax of many practical programming languages, including lexical syntax, as a unified TDPL grammar.

7.2 Packrat Parsing

Packrat parsing is an adaptation of a classic tabular parsing algorithm whose existence has been known since the development of TDPL in the 1970s, but which apparently has never been put into practice until now. A packrat parser can recognize any string defined by a TDPL grammar in linear time, providing the power and flexibility of a backtracking recursive descent parser without the attendant risk of exponential parse time. A packrat parser can recognize any LL(k) or LR(k) language, as well as many languages requiring unlimited lookahead that cannot be parsed by shift/reduce parsers. Packrat parsing also provides better composition properties than LL/LR parsing, making it more suitable for dynamic or extensible languages. The primary limitation of the algorithm is its considerable, though asymptotically linear, storage cost.

This thesis has explored the implementation of packrat parsers both manually and automatically with a parser generator. The monadic combinators and lazy evaluation capabilities of modern functional programming languages such as Haskell make the manual implementation of packrat parsers particularly simple and elegant. Packrat parsing is nearly as easy to implement in a non-strict language as recursive descent parsing with backtracking, and even simpler than predictive parsing because no special lookahead mechanism is necessary. Though graceful error handling in a packrat parser is slightly more involved than it is in a conventional deterministic parser, the use of monadic combinators to implement a packrat parser makes implementation of sophisticated error handling straightforward and painless.

7.3 Pappy

Finally, this thesis has presented Pappy, an automatic packrat parser generator that further simplifies parser implementation by creating packrat parsers from declarative specifications. Pappy parser specifications support the full generality of TDPL for the purpose of specifying syntax. In addition, Pappy extends TDPL notation with features that allow the generated parser to compute semantic values for recognized constructs using arbitrary fragments of Haskell code. Finally, Pappy complements TDPL's capability of expressing syntactic predicates with an analogous capability of representing *semantic predicates*, allowing parsing decisions to be made based on semantic values. In the example parsers presented here, semantic predicates are used to implement character classes, and to allow keywords and operators in a language to be handled uniformly at the lexical level while remaining distinguishable by high-level syntactic constructs.

7.4 Practical Experience

Both hand-implemented and automatically-generated packrat parsers for the Java programming language were developed in this thesis in order to evaluate the packrat parsing algorithm. The example parsers demonstrate that packrat parsing is a viable method of parsing real programming languages with complex grammars. Packrat parsers provide reliable linear-time performance, particularly when when repetition constructs in the grammar are rewritten to avoid hidden recursion. The packrat parsers consume 300–650 bytes of heap storage on average per byte of input text, which is reasonable for typical problem sizes on modern modern machines.

The example packrat parsers demonstrate practical linear-time "scannerless" parsing, in which the lexical analyzer is fully integrated with the parser. Packrat parsing is the first practical parsing algorithm to support scannerless parsing in linear time, without imposing complex constraints on the grammar used to specify the language.

7.5 Future Work

While the results presented here demonstrate the power and practicality of packrat parsing, more experimentation is needed to evaluate its flexibility, performance, and space consumption on a wider variety of languages. For example, languages that rely extensively on parser state, such as C and C++, as well as layout-sensitive languages such as ML and Haskell, may prove more difficult for a packrat parser to handle efficiently.

On the other hand, the syntax of a practical language is usually designed with a particular parsing technology in mind. For this reason, an equally compelling question is what new syntax design possibilities are created by the "free" unlimited lookahead and unrestricted grammar composition capabilities of packrat parsing. Section 3.2.2 suggested a few simple extensions that depend on integrated lexical analysis, but packrat parsing may be even more useful in languages with extensible syntax [6] where grammar composition flexibility is important.

One practical area in which packrat parsing may have difficulty and warrants further study is in parsing interactive streams. For example, the "read-eval-print" loops in language interpreters often expect the parser to detect at the end of each line whether or not more input is needed to finish the current statement, and this requirement violates the packrat algorithm's assumption that the entire input stream is available up-front. A similar open question is under what conditions packrat parsing may be suitable for parsing infinite streams.

There is much opportunity for additional formal development of the TDPL paradigm. It is still unproven whether or not gTS/GTDPL is in fact more powerful than TS/TDPL, or even whether context-free languages exist that cannot be recognized by a TDPL grammar. It would be highly useful to have algorithms available that could "bridge the gap" between CFGs and parsing grammars, for example by testing for language-equivalence or converting useful subclasses of grammars from one form to the other. Methods of "unparsing," or writing a string according to a TDPL grammar given structured high-level information such as an abstract syntax tree, would also be useful.

Appendix A

Example Packrat Parsers

This appendix contains complete listings of two of the example packrat parsers developed in Chapter 3. The next section contains the full version of the basic packrat parser described in Section 3.1.4, and Section A.2 contains an equivalent parser constructed using the monadic combinators developed in Section 3.2.3.

A.1 Basic Expression Parser

```
-- Packrat parser for trivial arithmetic language.
module ArithPackrat where


data Result v =
          Parsed v Derivs
        | NoParse

data Derivs = Derivs {
                dvAdditive      :: Result Int,
                dvMultitive     :: Result Int,
                dvPrimary       :: Result Int,
                dvDecimal       :: Result Int,
                dvChar          :: Result Char
        }


-- Evaluate an expression and return the unpackaged result,
-- ignoring any unparsed remainder.
eval s = case dvAdditive (parse s) of
                Parsed v rem -> v
                _ -> error "Parse error"
```

```
-- Construct a (lazy) parse result structure for an input string,
-- in which any result can be computed in linear time
-- with respect to the length of the input.
parse :: String -> Derivs
parse s = d where
    d    = Derivs add mult prim dec chr
    add  = pAdditive d
    mult = pMultitive d
    prim = pPrimary d
    dec  = pDecimal d
    chr  = case s of
             (c:s') -> Parsed c (parse s')
             [] -> NoParse


-- Parse an additive-precedence expression
pAdditive :: Derivs -> Result Int
pAdditive d = alt1 where

    -- Additive <- Multitive '+' Additive
    alt1 = case dvMultitive d of
             Parsed vleft d' ->
               case dvChar d' of
                 Parsed '+' d'' ->
                   case dvAdditive d'' of
                     Parsed vright d''' ->
                       Parsed (vleft + vright) d'''
                     _ -> alt2
                 _ -> alt2
             _ -> alt2

    -- Additive <- Multitive
    alt2 = case dvMultitive d of
             Parsed v d' -> Parsed v d'
             NoParse -> NoParse

-- Parse a multiplicative-precedence expression
pMultitive :: Derivs -> Result Int
pMultitive d = alt1 where

    -- Multitive <- Primary '*' Multitive
    alt1 = case dvPrimary d of
             Parsed vleft d' ->
               case dvChar d' of
                 Parsed '*' d'' ->
                   case dvMultitive d'' of
                     Parsed vright d''' ->
                       Parsed (vleft * vright) d'''
                     _ -> alt2
                 _ -> alt2
             _ -> alt2

    -- Multitive <- Primary
    alt2 = case dvPrimary d of
             Parsed v d’ -> Parsed v d’
             NoParse -> NoParse

-- Parse a primary expression
pPrimary :: Derivs -> Result Int
pPrimary d = alt1 where

    -- Primary <- ’(’ Additive ’)’
    alt1 = case dvChar d of
             Parsed ’(’ d’ ->
               case dvAdditive d’ of
                 Parsed v d’’ ->
                   case dvChar d’’ of
                     Parsed ’)’ d’’’ -> Parsed v d’’’
                     _ -> alt2
                 _ -> alt2
             _ -> alt2

    -- Primary <- Decimal
    alt2 = case dvDecimal d of
             Parsed v d’ -> Parsed v d’
             NoParse -> NoParse

-- Parse a decimal digit
pDecimal :: Derivs -> Result Int
pDecimal d = case dvChar d of
                Parsed ’0’ d’ -> Parsed 0 d’
                Parsed ’1’ d’ -> Parsed 1 d’
                Parsed ’2’ d’ -> Parsed 2 d’
                Parsed ’3’ d’ -> Parsed 3 d’
                Parsed ’4’ d’ -> Parsed 4 d’
                Parsed ’5’ d’ -> Parsed 5 d’
                Parsed ’6’ d’ -> Parsed 6 d’
                Parsed ’7’ d’ -> Parsed 7 d’
                Parsed ’8’ d’ -> Parsed 8 d’
                Parsed ’9’ d’ -> Parsed 9 d’
                _ -> NoParse
```

A.2 Monadic Expression Parser

```
-- Packrat parser for trivial arithmetic language.
module ArithPackrat where

import Pos
import Parse


data ArithDerivs = ArithDerivs {
                dvAdditive      :: Result ArithDerivs Int,
                dvMultitive     :: Result ArithDerivs Int,
                dvPrimary       :: Result ArithDerivs Int,
                dvDecimal       :: Result ArithDerivs Int,
                advChar         :: Result ArithDerivs Char,
                advPos          :: Pos
        }

instance Derivs ArithDerivs where
        dvChar d = advChar d
        dvPos d = advPos d


-- Evaluate an expression and return the unpackaged result,
-- ignoring any unparsed remainder.
eval s = case dvAdditive (parse (Pos "<input>" 1 1) s) of
                Parsed v d' e' -> v
                _ -> error "Parse error"


-- Construct a (lazy) parse result structure for an input string,
-- in which any result can be computed in linear time
-- with respect to the length of the input.
parse :: Pos -> String -> ArithDerivs
parse pos s = d where
    d    = ArithDerivs add mult prim dec chr pos
    add  = pAdditive d
    mult = pMultitive d
    prim = pPrimary d
    dec  = pDecimal d
    chr  = case s of
             (c:s') -> Parsed c (parse (nextPos pos c) s') (nullError d)
             [] -> NoParse (eofError d)
```

```
-- Parse an additive-precedence expression
pAdditive :: ArithDerivs -> Result ArithDerivs Int
Parser pAdditive =
            (do vleft <- Parser dvMultitive
                char '+'
                vright <- Parser dvAdditive
                return (vleft + vright))
        </> (do Parser dvMultitive)

-- Parse a multiplicative-precedence expression
pMultitive :: ArithDerivs -> Result ArithDerivs Int
Parser pMultitive =
            (do vleft <- Parser dvPrimary
                char '*'
                vright <- Parser dvMultitive
                return (vleft * vright))
        </> (do Parser dvPrimary)

-- Parse a primary expression
pPrimary :: ArithDerivs -> Result ArithDerivs Int
Parser pPrimary =
            (do char '('
                vexp <- Parser dvAdditive
                char ')'
                return vexp)
        </> (do Parser dvDecimal)

-- Parse a decimal digit
pDecimal :: ArithDerivs -> Result ArithDerivs Int
Parser pDecimal =
            (do c <- digit
                return (digitToInt c))
```

Appendix B

TDPL Grammar for the Pappy Parser Specification Language

```
Start          ←  Spacing Grammar EOF

Grammar        ←  Header RawCode? TopDecl Definition* RawCode?

Header         ←  PARSER Identifier COLON
TopDecl        ←  TOP Nonterminal (COMMA Nonterminal)*

Nonterminal    ←  Identifier

Definition     ←  Nonterminal DOUBLECOLON HaskellType EQUALS Rule

HaskellType    ←  Identifier
               /  RawCode

PrimRule       ←  Nonterminal
               /  CharLiteral
               /  StringLiteral
               /  OPEN Rule CLOSE
UnaryRule      ←  PrimRule QUESTION
               /  PrimRule STAR
               /  PrimRule PLUS
               /  PrimRule
SeqRule        ←  Sequence
               /  UnaryRule
AltRule        ←  SeqRule (SLASH SeqRule)*
Rule           ←  AltRule
```

```
Sequence                  ←  SeqMatcher* ARROW SeqResult
SeqMatcher                ←  Identifier COLON UnaryRule
                          /  RawCode COLON UnaryRule
                          /  CharLiteral COLON UnaryRule
                          /  StringLiteral COLON UnaryRule
                          /  AND UnaryRule
                          /  NOT UnaryRule
                          /  AND RawCode
                          /  UnaryRule
SeqResult                 ←  Identifier
                          /  RawCode

Identifier                ←  IdentStart IdentCont* Spacing
IdentStart                ←  Letter / ‘_’
IdentCont                 ←  IdentStart / Digit / ‘'’

CharLit                   ←  ‘'’ (!(‘'’) QuotedChar) ‘'’ Spacing
StringLit                 ←  ‘"’ (!(‘"’) QuotedChar)* ‘"’ Spacing
QuotedChar                ←  ‘\n’ / ‘\r’ / ‘\t’ / ‘\\’ / ‘\'’ / ‘\"’
                          /  !(‘\’) Char

RawCode                   ←  HaskellBlock Spacing

HaskellBlock              ←  ‘{’ HaskellToken* ‘}’
HaskellToken              ←  HaskellBlock
                          /  HaskellIdentifier
                          /  HaskellCharLiteral
                          /  HaskellStringLiteral
                          /  !(‘{’ / ‘}’ / ‘'’ / ‘"’) Char

HaskellIdentifier         ←  IdentStart IdentCont*

HaskellCharLiteral        ←  ‘'’ HaskellSingleQuoteChar* ‘'’
HaskellSingleQuoteChar    ←  ‘\’ Char
                          /  !(‘'’ / CR / LF) Char

HaskellStringLiteral      ←  ‘"’ HaskellDoubleQuoteChar* ‘"’
HaskellDoubleQuoteChar    ←  ‘\’ Char
                          /  !(‘"’ / CR / LF) Char

PARSER                    ←  ‘parser’ Spacing
TOP                       ←  ‘top’ Spacing
DOUBLECOLON               ←  ‘::’ Spacing
EQUALS                    ←  ‘=’ Spacing
COLON                     ←  ‘:’ Spacing
COMMA                     ←  ‘,’ Spacing
OPEN                      ←  ‘(’ Spacing
CLOSE                     ←  ‘)’ Spacing
```

```
QUESTION          ←  ‘?’ Spacing
STAR              ←  ‘*’ Spacing
PLUS              ←  ‘+’ Spacing
SLASH             ←  ‘/’ Spacing
ARROW             ←  ‘->’ Spacing
AND               ←  ‘&’ Spacing
NOT               ←  ‘!’ Spacing

Spacing           ←  (SpaceChar / LineComment)*
LineComment       ←  ‘--’ (!(LineTerminator) Char)* LineTerminator

SpaceChar         ←  ‘ ’ / TAB / CR / LF
LineTerminator    ←  CR LF / CR / LF

CR                ←  ‘\r’
LF                ←  ‘\n’
TAB               ←  ‘\t’

EOF               ←  !(Char)
```

Bibliography

[1] Stephen Robert Adams. *Modular Grammars for Programming Language Prototyping.* PhD thesis, University of Southampton, 1991.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling - Vol. I: Parsing.* Prentice Hall, Englewood Cliffs, N.J., 1972.

[4] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34, Aug 1973.

[5] Carlos Camarão and Lucília Figueiredo. A monadic combinator compiler compiler. In *5th Brazilian Symposium on Programming Languages*, Curitiba – PR – Brazil, May 2001. Universidade Federal do Paraná.

[6] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Systems Research Center, 1994.

[7] Jeroen Fokker. Functional parsers. In *Advanced Functional Programming*, pages 1–23, 1995.

[8] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming*, Oct 2002.

[9] Andy Gill and Simon Marlow. Happy: The parser generator for Haskell. `http://www.haskell.org/happy`.

[10] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, Jul 1992.

[11] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, Jul 1998.

[12] Simon Peyton Jones and John Hughes (editors). *Haskell 98 Report*, 1998. `http://www.haskell.org`.

[13] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM*, 2002. To appear.

[14] Daan Leijen. Parsec, a fast combinator parser. `http://www.cs.uu.nl/~daan`.

[15] R. M. McClure. TMG—a syntax directed compiler. In *Proceedings of the 20th ACM National Conference*, pages 262–274, 1965.

[16] Terence J. Parr and Russell W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *Computational Complexity*, pages 263–277, 1994.

[17] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

[18] Peter Pepper. LR parsing = grammar transformation + LL parsing: Making LR parsing more understandable and more efficient. Technical Report 99-5, TU Berlin, Apr 1999.

[19] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, pages 170–178, Jul 1989.

[20] D. V. Schorre. META II, a syntax-oriented compiler writing language. In *Proceedings of the 19th ACM National Conference*, pages D1.3-1–D1.3-11, 1964.

[21] Thomas G. Szymanski and John H. Williams. Noncanonical extensions of bottom-up parsing techniques. *SIAM Journal on Computing*, 5(2):231–250, Jun 1976.

[22] Kuo-Chung Tai. Noncanonical SLR(1) grammars. *ACM Transactions on Programming Languages and Systems*, 1(2):295–320, Oct 1979.

[23] Masaru Tomita. *Efficient parsing for natural language.* Kluwer Academic Publishers, 1985.

[24] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction*, 2002.

[25] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[26] Philip Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, pages 113–128, 1985.

[27] John H. Williams. Bounded context parsable grammars. Technical Report 72-127, Department of Computer Science, Cornell University, April 1972.