Viewpoints Research Institute

Experimenting with Programming Languages

Alessandro Warth

VPRI Technical Report TR-2008-003

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

University of California

Los Angeles

Experimenting with Programming Languages

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Alessandro Warth

2009

The dissertation of Alessandro Warth is approved.

Rebecca Allen

Junghoo “John” Cho

Alan Kay

Jens Palsberg

Todd Millstein, Committee Chair

University of California, Los Angeles

2009

To Carolyn, who had no
idea what she was getting into when
she married a Ph.D. student, but
by now has completed the
requirements for the
degree of

>>– Doctor of Philosophy in
Unwavering Support–>>

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am forever indebted to my advisors Todd Millstein and Alan Kay for the tremendous opportunities, support, inspiration, insight, and—you guessed it—*advice* they have given me over the past few years.

Todd is, in more ways than one, the reason why I ended up at UCLA. Sure, he has taught me lots of really interesting "technical stuff", but more importantly, he taught me how to be a researcher. I am also thankful for all the freedom and encouragement he has given me to explore the ideas that excite me the most, no matter how risky they may be, publication-wise. As an advisor and as a research colleague, he could not have been more helpful.

I owe just as much thanks to Alan, who is not only a bottomless well of great ideas and inspiration—a veritable "epiphany generator"—but also couldn't have been more supportive. Nearly every time we talked about what I was working on over lunch, I ended up scrapping all of my source code and starting over from scratch... and I couldn't be happier.

I also owe a big "thanks" to the other members of my Ph.D. committee: Rebecca Allen and Junghoo Cho made me think about interesting applications of my work, and Jens Palsberg went above and beyond his duties when he not only suggested that I should add an operational semantics of OMeta to this dissertation, but also spent an entire afternoon with me, writing inference rules on a dry-erase board.

I would also like to thank Kim Rose and my colleagues at VPRI for the countless discussions that not only broadened my horizons, but also inspired my research. The same goes for Ben Titzer and my colleagues at the TERTL lab at UCLA.

My work on OMeta, presented in Chapter 2, started as a collaboration with Ian Piumarta. (Ian also introduced me to dynamic languages, and I can't think of anyone

who could have done that better!) I would like to thank Alan Kay for inspiring this project, Yoshiki Ohshima for helping me port OMeta to Squeak, Takashi Yamamiya for suggesting the JavaScript port and for inspiring me with his JavaScript Workspace, and Dan Amelang, Tom Bergan, Jamie Douglass, Paul Eggert, Bert Freudenberg, Mike Mammarella, Todd Millstein, Stephen Murrell, Ben Titzer, and Scott Wallace for their valuable feedback. I would also like to thank Chris Double, Ted Kaehler, Jeff Moser, Amarin Phaosawasdi, Allen Short, Chris Smoak, and the rest of OMeta's excellent user community. And a big thanks to Dan Ingalls for suggesting me as a keynote speaker to the organizers of Smalltalk Solutions 2008 (my talk was about OMeta).

Thanks to Jamie Douglass and Todd Millstein, who collaborated with me on the left recursion algorithm presented in Chapter 3, and also to Tom Bergan, Richard Cobbe, Dave Herman, Stephen Murrell, Yoshiki Ohshima, and the anonymous reviewers of PEPM '08 for their useful feedback on this work.

My work on worlds, presented in Chapter 4, has benefited greatly from discussions with Gilad Bracha, Jeff Fischer, Josh Gargus, Mark Miller and other members of the Caja group at Google, Eliot Miranda, Andreas Raab, David Reed, Hesam Samimi, and David Smith.

My time at UCLA would have been much less productive (and infinitely more stressful!) without the help of Verra Morgan, Korina Pacyniak, and Rachelle Reamkitkarn, who time and time again made cutting through massive amounts of red tape look easy.

A huge thanks to my wife Carolyn and the rest of my family for always being supportive and for understanding that the nervous, irritable, nasty guy that I turn into when a paper deadline approaches is not the real me. I love you guys.

Last but certainly not least, I would like to thank my wonderful friend and mentor Stephen Murrell for always inspiring me to be a better ~~programmer~~ thinker, for sparking my interests in programming languages as well as graduate school, and for being the best “secret weapon” ever.

During my time as a graduate student at UCLA, I was lucky enough to receive funding from a GAANN Fellowship (2004–2005), a Teaching Assistantship (2005–2006), a Research Assistantship (Fall 2006), and most recently, the Viewpoints Research Institute (2007–present).

Vita

1978 Born, Porto Alegre, Rio Grande do Sul, Brazil.

1988 Got an *Expert* (a Brazilian MSX-based home computer).

1998 Summer Intern, AT&T RAPID Development Group, Atlanta, GA.

1998—2000 Tutor / Grader, Electrical and Computer Engineering Department, University of Miami. Under the direction of Dr. Stephen Murrell.

1999 Participated in the Summer Undergraduate Program in Engineering Research at Berkeley (SUPERB).

2000 Dual B.S. in Computer Engineering and Computer Science, University of Miami.

2000—2003 Senior Software Engineer, Verid, Inc. (formerly iShopSecure).

2004—2005 Graduate Student Researcher, Computer Science Department, UCLA.

2005—2006 Teaching Assistant, Computer Science Department, UCLA. Taught sections of CS 131 (Programming Languages) under the direction of Dr. Todd Millstein and Dr. Paul Eggert.

2006 M.S. in Computer Science, UCLA.

2007—present Research Associate, Viewpoints Research Institute.

Publications and Presentations

Alessandro Warth, Milan Stanojevíc, and Todd Millstein. Statically Scoped Object Adaptation with Expanders. In *OOPSLA '06: Proceedings of the 21st ACM SIGPLAN International Conference on Object-Oriented Programming Languages and Applications.* Portland, OR. October 2006.

Alessandro Warth. LazyJ: Seamless Lazy Evaluation in Java. In *FOOL/WOOD '07: the International Workshop on Foundations and Developments of Object-Oriented Languages.* Nice, France. January 2007.

Alessandro Warth and Ian Piumarta. OMeta: an Object-Oriented Language for Pattern Matching. In *DLS '07: the Dynamic Languages Symposium.* Montreal, Canada. October 2007.

Alessandro Warth, James R. Douglass, and Todd Millstein. Packrat Parsers Can Support Left Recursion. In *PEPM '08: the Workshop on Partial Evaluation and Program Manipulation.* San Francisco, CA. January 2008.

Alessandro Warth, Takashi Yamamiya, Yoshiki Ohshima, and Scott Wallace. Toward a More Scalable End-User Scripting Language. In *C5 '08: Proceedings of the Sixth International Conference on Creating, Connecting, and Collaborating through Computing.* Poitiers, France. January 2008.

Ian Piumarta and Alessandro Warth. Open, Reusable Object Models. In *S3 '08: the*

Workshop on Self-Sustaining Systems. Potsdam, Germany. May 2008.

Alessandro Warth. Implementing Programming Languages for Fun and Profit with OMeta. Keynote, *Smalltalk Solutions '08.* Reno, NV. June 2008.

Takashi Yamamiya, Alessandro Warth, and Ted Kaehler. Active Essays on the Web. In *C5 '09: Proceedings of the Seventh International Conference on Creating, Connecting, and Collaborating through Computing.* Kyoto, Japan. January 2009. (to appear)

Todd Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Open, Expressive and Modular Predicate Dispatch for Java. In *Transactions on Programming Languages and Systems.* (to appear)

Abstract of the Dissertation

Experimenting with Programming Languages

by

Alessandro Warth

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2009

Professor Todd Millstein, Chair

Researchers are in the business of having good ideas. Of course, not every idea we ever have is a good idea, so we rely on *experiments* to find out how good an idea really is. Unfortunately, because experiments are expensive—they usually take considerable time and effort—we can only afford to experiment with our "more promising" ideas. This is a problem, because the only way to really know if an idea is promising or not is to experiment with it in the first place! It follows that new ideas and technologies that enable researchers to experiment more quickly and easily can have a huge impact on the rate of progress of any given scientific discipline.

This dissertation focuses on experimentation in computer science. In particular, I will show that new programming languages and constructs designed specifically to support experimentation can substantially simplify the jobs of researchers and programmers alike.

I present work that addresses two very different kinds of experimentation. The first aims to help programming language researchers experiment with their ideas, by mak-

ing it easier for them to prototype new programming languages and extensions to existing languages. The other investigates experimentation as a programming paradigm, by enabling programs themselves to experiment with different actions and possibilities—in other words, it is an attempt to provide language support for *what if...?* or *possible worlds* reasoning.

CHAPTER 1

Introduction

"Every great idea is on the verge of being stupid."

— Michel Gondry, Filmmaker

Researchers are in the business of having good ideas. Of course, not every idea we ever have is a good idea, so we rely on *experiments* to find out how good an idea really is. Unfortunately, because experiments are expensive—they usually take considerable time and effort—we can only afford to experiment with our "more promising" ideas. This is a problem, because the only way to really know if an idea is promising or not is to experiment with it in the first place! It follows that new ideas and technologies that enable researchers to experiment more quickly and easily can have a huge impact on the rate of progress of any given scientific discipline.

This dissertation focuses on experimentation in computer science. In particular, I will show that new programming languages and constructs designed specifically to support experimentation can substantially simplify the jobs of researchers and programmers alike.

The work presented here addresses two very different kinds of experimentation. The first aims to help programming language researchers experiment with their ideas, by making it easier for them to prototype new programming languages and extensions to existing languages. The other investigates experimentation as a programming paradigm, by enabling programs themselves to experiment with different actions and possibilities—in other words, it is an attempt to provide language support for *what if...?* or *possible worlds* reasoning.

In the remainder of this chapter, I describe the goals of each of these projects in more detail (Sections 1.1 and 1.2), and outline the rest of the dissertation (Section 1.3).

1.1 *OMeta*: Experimenting with Language Design

> All languages are equally powerful in the sense of being Turing equivalent, but that's not the sense of the word programmers care about. (No one wants to program a Turing machine.) The kind of power programmers care about may not be formally definable, but one way to explain it would be to say that it refers to features you could only get in the less powerful language by writing an interpreter for the more powerful language in it. If language A has an operator for removing spaces from strings and language B doesn't, that probably doesn't make A more powerful, because you can probably write a subroutine to do it in B. But if A supports, say, recursion, and B doesn't, that's not likely to be something you can fix by writing library functions.
>
> — Paul Graham, in "Beating the Averages" [Gra03]

Programming language researchers are always looking for ways to put more expressive power in the hands of programmers. They do this by inventing new forms of abstraction, e.g., encapsulation and extensibility mechanisms, control structures, and type systems. While, computationally speaking, none of these constructs actually makes a programming language more powerful, they allow programmers to abstract away from uninteresting details and concentrate on the problem at hand. This in turn makes *programmers themselves* more powerful, i.e., more productive and able to tackle more complex problems.

These abstractions, each with its own syntax and semantics, make up the user interface that a programmer uses to control the computer. Note that a good semantics is seldom effective on its own; in order to be truly helpful to programmers, it must be provided in a graceful form that is convenient to use. For example, consider a library that provides the semantics of classes and messaging to C programs. Using this abstraction to build object-oriented applications would be burdensome and error-prone, since its implementation details would be exposed (much like all the knobs and buttons of an over-complicated user interface). Also, as Paul Graham notes, there are times when it is not possible for the programmer to implement the desired semantics as a library. Programming language researchers usually avoid these problems by implementing new abstractions as extensions to existing programming languages.

But building language extensions is not an easy task. Traditionally, a programming language is implemented using a number of different tools, like *lex* [LS90], *yacc* [Joh79], and the *visitor design pattern* [GHJ95]. While each of these tools simplifies a single part of the implementation, their combination makes the task of extending the implementation as a whole difficult because each part must be extended in a different way. To make matters worse, some of these tools—e.g., *lex* and *yacc*—do not even provide extensibility mechanisms, which forces the implementor of a language extension to duplicate code from the base implementation and modify it in-place. This in turn results in code bloat and a versioning problem, since subsequent changes/improvements to the base language will not carry over to the extension. All of these complications make it difficult for researchers to experiment with new ideas, which slows down the rate of innovation in programming languages.

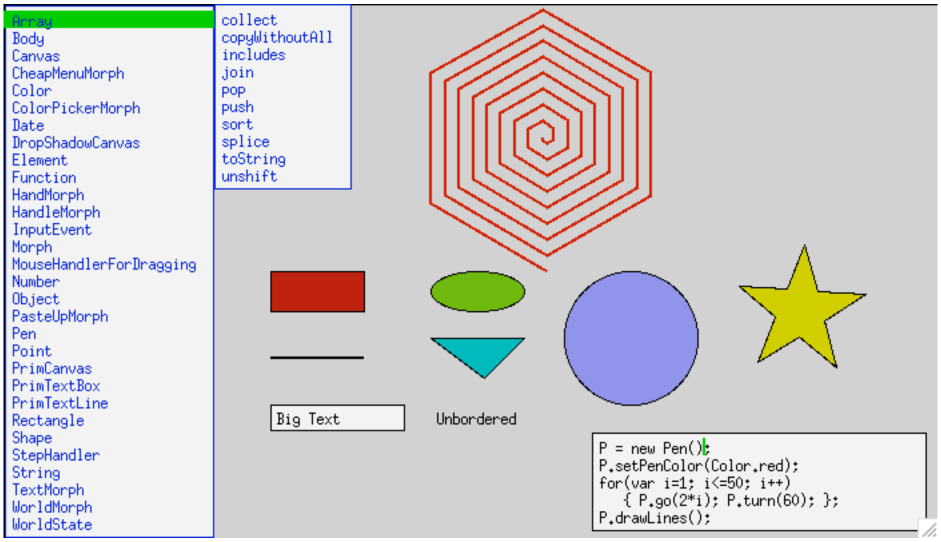
An equally serious problem is the inflexibility of traditional programming languages, which force programmers to cope with a fixed set of abstractions. Many applications would be substantially easier to write and to maintain if only programmers were able to extend a programming language with their own application-specific constructs.

My language *OMeta* was designed to make it easier for researchers and programmers alike to provide useful abstractions in forms that are convenient to use. OMeta's key insight is the realization that all of the passes in a traditional compiler are essentially pattern matching operations:

- a *lexical analyzer* finds patterns in a stream of characters to produce a stream of tokens;
- a *parser* matches a stream of tokens against a grammar (which itself is a collection of productions, or patterns) to produce abstract syntax trees (ASTs);
- a *typechecker* pattern-matches on ASTs to produce ASTs annotated with types;
- more generally, *visitors* pattern-match on ASTs to produce other ASTs;
- finally, a (naive) *code generator* pattern-matches on ASTs to produce code.

By extending Parsing Expression Grammars (PEGs) [For04] with support for pattern matching on arbitrary datatypes, left recursion, and parameterized and higher-order rules, OMeta gives programmers a natural and convenient way to implement parsers, visitors, and tree transformers, all of which can be extended in interesting ways using familiar object-oriented mechanisms. This makes OMeta particularly well-suited as a tool for building language implementations and extensions.

As an example, in just a few hundred lines of OMeta I was able to implement a JavaScript compiler for the COLA platform [Piu06a]. Figure 1.1 shows a screenshot of Sun Microsystems' Lively Kernel [Ing08] running on my JavaScript implementation.



```
P = new Pen();
P.setPenColor(Color.red);
for(var i=1; i<=50; i++)
   { P.go(2*i); P.turn(60); };
P.drawLines();
```

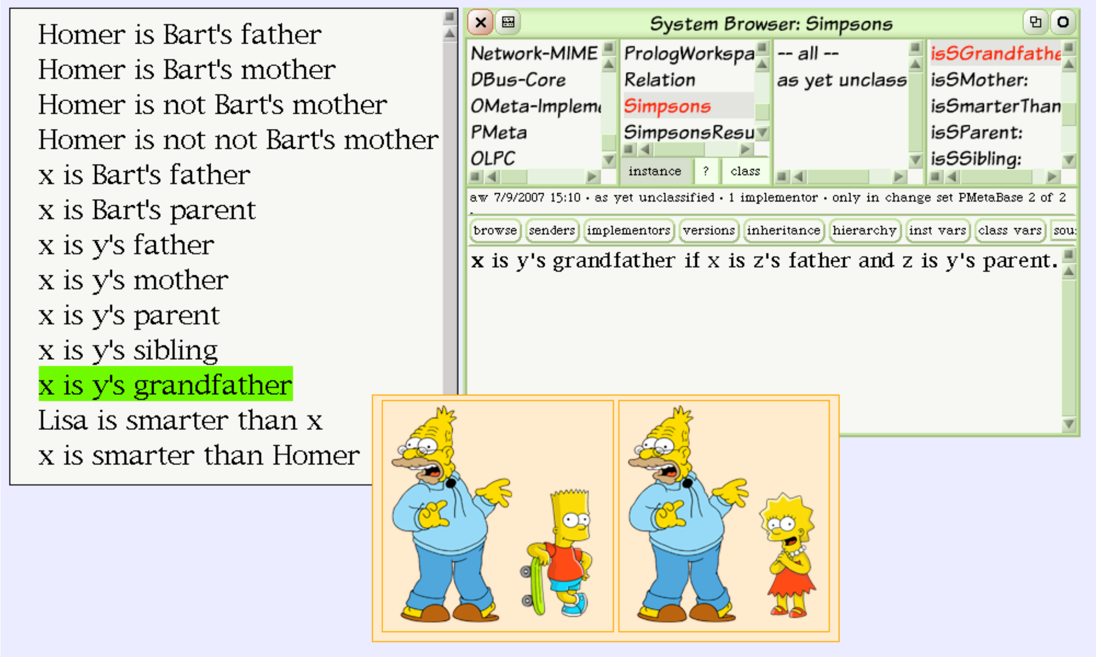
Figure 1.1: An early version of Sun's Lively Kernel running on a JavaScript implementation written in OMeta/COLA

Having a JavaScript compiler written in OMeta made it easy for me to experiment with a number of interesting language extensions, including the macro system that was used to implement TileScript, an end-user scripting language [WYO08].

Another example of OMeta's suitability as a rapid prototyping tool is Toylog, a natural language front-end to Prolog designed to get children and teenagers interested in programming, shown in Figure 1.2. I wrote the Toylog parser in less than one hundred lines of OMeta, as an afternoon project. Toylog was recently used to teach a group of approximately one hundred high school students in Australia—hands-on, in a laboratory setting—where I am told it generated a lot of enthusiasm.

1.2 *Worlds*: Experimenting with Possibilities

We can think of a program as an *agency* set up to accomplish or *reach* one or more goals. For some goals—such as adding up numbers—there is a simple straight road in most programming languages. Other goals—such as finding the square root of a



Homer is Bart's father
Homer is Bart's mother
Homer is not Bart's mother
Homer is not not Bart's mother
x is Bart's father
x is Bart's parent
x is y's father
x is y's mother
x is y's parent
x is y's sibling
x is y's grandfather
Lisa is smarter than x
x is smarter than Homer

aw 7/9/2007 15:10 · as yet unclassified · 1 implementor · only in change set PMetaBase 2 of 2

x is y's grandfather if x is z's father and z is y's parent.

Figure 1.2: Toylog, a natural language interface to Prolog

number—will require some search, but can still be homed in on very effectively. For more complex goals—such as certain kinds of parsing, problem solving, and editing in a user interface—it may not be possible for the program to simply home in on the desired result; instead, it may need to experiment with multiple alternatives, sometimes making mistakes that require retraction in order to make fresh starts.

For some goals of this latter kind, it might be a good idea to use *backtracking*, as in Prolog (it would also be a good idea to automatically undo `asserts` that are backtracked over, which does not happen in Prolog). For problem solving on a larger scale, it might be a better ploy to use "possible worlds", perhaps even *parallel* possible worlds that can report progress to a coordination agent that is trying to make choices about best paths. If the problem terrain is really rough and tricky—a veritable "elephant to blind men"—then we might want to use the metaphor of dispatching "scouts" (or "scientists") with walkie-talkies who can simultaneously explore different parts and intercommunicate to gradually build up a better model of the obstacles and the possible routes around them.

Unfortunately, the nature of *state* in traditional programming languages makes this "experimental" style of programming impractical. For example, because the state of a program is scattered around the computer's memory in several kinds of data structures (e.g., arrays, objects, and activation records), it is both messy and difficult to *undo* the side effects of an action that has been performed by the program. Also, because there is only one "program state", a program cannot explore multiple alternatives concurrently.

My approach to solving this problem was to introduce a new language construct that reifies the notion of program state as a first-class object. I call this construct a *world.* All computation takes place inside a world, which captures all of the side effects—changes to global, local, and instance variables, arrays, etc.—that happen inside it. Worlds provide multiple views on the state of a program, and mechanisms for interacting among these views. They subsume the mechanisms of backtracking, tentative evaluation, possible worlds, undo, and many similar control and state regimes. We shall see that while it is often convenient for programmers to use worlds directly, there are also cases where worlds are better suited as a kind of "semantic building block" for higher-level language constructs.

1.3 Statement of the Thesis and Organization of the Dissertation

The thesis of this dissertation is as follows:

> *Programming languages and constructs designed specifically to support experimentation can substantially simplify the jobs of researchers and programmers alike.*

I support this thesis by developing programming language support for the two kinds of experimentation discussed earlier in this chapter.

The remainder of this dissertation is organized as follows:

- Chapter 2 describes OMeta, its general-purpose pattern matching facilities and extensibility mechanisms, and shows how these can be used by researchers and programmers to experiment with language design ideas,
- Chapter 3 focuses on the mechanism I devised in order to support left recusion in my OMeta implementations,
- Chapter 4 introduces worlds and identifies a number of interesting idioms that are made possible by this new construct, and
- Chapter 5 concludes.

CHAPTER 2

OMeta: An Object-Oriented Language for Pattern-Matching

This chapter introduces a notion of general-purpose pattern matching and its instantiation in OMeta, an object-oriented language designed to make it easier for programming language researchers to experiment with their ideas. OMeta is based on a variant of Parsing Expression Grammars (PEGs) [For04]—a recognition-based foundation for describing syntax—which I have extended to support pattern matching on arbitrary datatypes. I show that OMeta's general-purpose pattern matching facilities provide a natural and convenient way for programmers to implement tokenizers, parsers, visitors, and tree transformers, all of which can be extended in interesting ways using familiar object-oriented mechanisms. This makes OMeta particularly well-suited as a medium for experimenting with new designs for programming languages and extensions to existing languages.

2.1 Introduction

Many problems in computer science, especially in programming language implementation, involve some form of pattern matching. Lexical analysis, for example, consists of finding patterns in a stream of characters to produce a stream of tokens. Similarly, a parser matches a stream of tokens against a grammar—which itself is a collection of rules, or patterns—to produce parse trees. Several other tasks, such as constant folding and naive code generation, can be implemented by pattern matching on parse trees.

Despite the fact that these are all instances of the same problem, most compiler writers use a different tool or technique (e.g., *lex*, *yacc*, and the *visitor design pattern* [GHJ95]) to implement each compilation phase. As a result, the skill of programming language implementation has a steep learning curve (because one must learn how to use a number of different tools) and is not widely understood.

Several popular programming languages—ML [MTM97], for instance—include support for pattern matching. But while ML-style pattern matching is suitable for processing structured data, it is not expressive enough on its own to support more complex pattern matching tasks such as lexical analysis and parsing.

Perhaps by providing programming language support for a more general form of pattern matching, many useful techniques such as parsing—a skill more or less exclusive to "programming languages people"—might become part of the skill-set of a much wider audience of programmers. (Consider how many Unix applications could be improved if suddenly their implementers had the ability to process more interesting configuration files!) This is not to say that general-purpose pattern matching is likely to subsume specialized tools such as parser generators; that would be difficult to do, especially in terms of performance. But as I will show with various examples, general-purpose pattern matching provides a natural and convenient way to implement tokenizers, parsers, visitors, and tree transformers, which makes it an unrivaled tool for rapid prototyping.

This work builds on Parsing Expression Grammars (PEGs) [For04], a recognition-based foundation for describing syntax, as a basis for general-purpose pattern matching, and makes the following technical contributions:

1. a generalization of PEGs that can operate on arbitrary datatypes—not just

streams of characters—and supports parameterized and higher-order rules (Section 2.2),

2. a simple yet powerful extensibility mechanism for PEGs (Section 2.3),

3. the design and implementation of *OMeta*, a programming language with convenient BNF-like syntax that embodies (1) and (2),

4. a series of examples that demonstrate how my general-purpose pattern matching facilities may be used in the domain of programming language implementation, and finally,

5. a formalism that describes the operational semantics of Core-OMeta, an extension to PEGs that supports OMeta-style pattern matching on arbitrary datatypes (Section 2.4).

The rest of this chapter explores my notion of general-purpose pattern matching in the context of OMeta.

2.2 OMeta: an extended PEG

An OMeta program is a Parsing Expression Grammar (PEG) [For04] that can make use of a number of extensions in order to operate on arbitrary datatypes. (PEGs are limited to processing streams of characters.) This section begins by introducing the features that OMeta and PEGs have in common, and then describes some of OMeta's extensions to PEGs.

expression	meaning
$e_1\ e_2$	sequencing
$e_1 \mid e_2$	*prioritized* choice
e*	zero or more repetitions
e+	one or more repetitions (not essential)
(e)	grouping
~e	negation
&e	look-ahead (not essential)
r	rule application
'x'	matches the character x

Table 2.1: Inductive definition of the language of *parsing expressions* (e, e_1, and e_2 are parsing expressions, and r is a non-terminal)

```
ometa ExpRecognizer {
  dig = '0' | ... | '9',
  num = dig+,
  fac = fac '*' num
      | fac '/' num
      | num,
  exp = exp '+' fac
      | exp '-' fac
      | fac
}
```

Figure 2.1: A PEG that recognizes simple arithmetic expressions

2.2.1 PEGs, OMeta Style

PEGs are a recognition-based foundation for describing syntax. A PEG is a collection of rules of the form *non-terminal* → *parsing-expression*; the language of parsing expressions is shown in Table 2.1.

Figure 2.1 shows a PEG, written in OMeta syntax, that recognizes simple arithmetic expressions. In order to go beyond simply accepting or rejecting input—e.g., to turn our recognizer into a parser—the programmer must write *semantic actions.* These are specified using the -> operator and written in OMeta's *host language*, which is usually the language in which the OMeta implementation was written. Most of the examples in this chapter are written in OMeta/JS, my JavaScript-based implementation.[1]

Using semantic actions, we can modify the definition of our recognizer's `exp` rule to create parse tree nodes:

```
exp = exp:x '+' fac:y -> ['add', x, y]
    | exp:x '-' fac:y -> ['sub', x, y]
    | fac
```

In this example, we represent parse tree nodes as JavaScript arrays whose first element is a string that identifies a kind of node, e.g., `'add'`, and whose other elements are sub-trees, e.g., the two operands of an *add* expression. Note that the results of `exp` and `fac` are bound to identifiers using the `:` operator so that they can be referenced in the semantic actions. Also note that the last choice in this rule, `fac`, does not specify a semantic action. In the absence of a semantic action, the value returned by a rule upon a successful match is the result of the last expression evaluated. Hence

```
fac
```

is equivalent to

```
fac:x -> x
```

A complete parser for our language of arithmetic expressions, in which the `fac` and `num` rules are also modified with semantic actions, is shown in Figure 2.2. In the `num` rule, the identifier `ds` is bound to an array of digits, which is converted to a string using `Array`'s `join` method, which in turn is converted to a number using the `parseInt` function. (`join` and `parseInt` are part of JavaScript's standard library.)

OMeta has a single built-in rule from which every other rule is derived. The name of this rule is `anything`, and it consumes exactly one object from the input stream.

[1] I have two other implementations: OMeta/Squeak, written in Squeak Smalltalk [IKM97], and OMeta/COLA, written in COLA [Piu06b]. All three of my OMeta implementations are available at `http://www.tinlizzie.org/ometa/`. There are also several third-party implementations available, based on a number of other languages including C#, Python, Scheme, Lisp, and Factor.

```
ometa ExpParser {
  dig = '0' | ... | '9',
  num = dig+:ds         -> ['num', parseInt(ds.join(''))],
  fac = fac:x '*' num:y -> ['mul', x, y]
      | fac:x '/' num:y -> ['div', x, y]
      | num,
  exp = exp:x '+' fac:y -> ['add', x, y]
      | exp:x '-' fac:y -> ['sub', x, y]
      | fac
}
```

Figure 2.2: A parser for simple arithmetic expressions

Even the `end` rule, which detects the end of the input stream, is implemented in terms of `anything`:

```
end = ~anything
```

In other words, the parser has reached the end of the input stream if it is not able to consume another object. As noted by Ford, the negation operator (~) can also be used to provide unlimited look-ahead capability [For04]. For example, `~~exp` ensures that an `exp` follows, but does not consume any input. This is in fact how OMeta's look-ahead operator (&) is implemented.

Like several other PEG implementations (e.g., *Pappy* [For02c]), OMeta also supports *semantic predicates* [PQ94], i.e., host language (boolean) expressions that are evaluated while pattern matching. In OMeta, semantic predicates are written using the ? operator. For example, the following rule matches a digit:

```
digit = char:d ?(d >= '0' && d <= '9') -> d
```

2.2.2 PEG Extensions

2.2.2.1 Matching Objects

PEGs operate on streams of characters. OMeta, on the other hand, operates on streams of *arbitrary host-language objects* (e.g., JavaScript objects). For this reason, it provides special syntax for matching common kinds of objects:

- strings, e.g., `'hello'`
- numbers, e.g., `42`
- lists, e.g., `['hello' 42 []]`

Note that the patterns `'x' 'y' 'z'` and `'xyz'` are not equivalent: the former matches three string objects, whereas the latter matches a single string object.[2] On the other hand, the patterns `['x' 'y' 'z']` and `'xyz'` will both match the string `'xyz'`, because a string can always be viewed as a list of characters.

List patterns enable OMeta grammars to pattern-match on arbitrarily-structured data. A list pattern may contain a nested pattern, which itself may be any valid parsing expression (e.g., a sequence of patterns, another list pattern, etc.). In order for a list pattern p to match a value v, two conditions must be met:

1. v must be a list, or list-like entity (e.g., a string), and
2. p's nested pattern must match *all* of the contents of v.

The pattern `[anything*]`, for example, matches any list, whereas `[]` only matches empty lists.

[2] In JavaScript, there is no such thing as a character datatype: the nth element of a string is simply a string of length 1.

```
ometa ExpEvaluator {
  eval = [’num’ anything:x]     -> x
       | [’add’ eval:x eval:y]  -> (x + y)
       | [’sub’ eval:x eval:y]  -> (x - y)
       | [’mul’ eval:x eval:y]  -> (x * y)
       | [’div’ eval:x eval:y]  -> (x / y)
}
```

Figure 2.3: Evaluating expressions

```
ometa ExpTranslator {
  trans = [’num’ anything:x]     -> x.toString()
        | [’add’ eval:x eval:y]  -> (’(’ + x + ’+’ + y + ’)’)
        | [’sub’ eval:x eval:y]  -> (’(’ + x + ’-’ + y + ’)’)
        | [’mul’ eval:x eval:y]  -> (’(’ + x + ’*’ + y + ’)’)
        | [’div’ eval:x eval:y]  -> (’(’ + x + ’/’ + y + ’)’)
}
```

Figure 2.4: “Compiling” expressions to JavaScript

Figure 2.3 shows a simple OMeta grammar that uses list patterns to evaluate the parse trees generated by the `ExpParser` (shown in Figure 2.2). Feeding the parse tree

```
[’mul’, [’num’, 6],
        [’add’, [’num’, 4], [’num’, 3]]]
```

to our grammar’s `eval` rule produces the result `42`.

Similarly, we can write an OMeta grammar that *translates* our parse trees to JavaScript code that, when evaluated, will yield the desired result (see Figure 2.4). This technique, known as *source-to-source compilation*, is used in most OMeta-based language implementations.

This extension to PEGs, which enables OMeta grammars to operate on structured as well as unstructured data, is formalized in Section 2.4.

2.2.2.2 Left Recursion

To avoid ambiguities that arise from using a non-deterministic choice operator (the kind of choice found in context-free grammars), PEGs only support *prioritized choice*. In other words, choices are always evaluated in order. As a result, there is no such thing as an ambiguous PEG, and their behavior is easy to understand.

Unfortunately, this semantics precludes the use of left recursion in PEGs. As an example, consider the following rule, taken from the expression parser shown in Figure 2.2:

```
fac = fac:x '*' num:y -> ['mul', x, y]
    | fac:x '/' num:y -> ['div', x, y]
    | num,
```

Note that the first choice in `fac` begins with an application of `fac` itself. Because the choice operator in PEGs tries each alternative in order, this recursion will never terminate: an application of `fac` will result in another application of `fac` without consuming any input, which in turn will result in yet another application of `fac`, and so on. The base case (`num`) will never be used.

We could re-order the alternatives so that `num` comes first, but to no avail. Since all valid `fac`tors begin with a `num`ber, the left-recursive alternatives would never be used, and `fac` would only parse a single number.

A slightly better alternative would be to rewrite `fac` to be right-recursive. However, this would result in right-associative parse trees, which is a problem because the multiplication operator is supposed to be left-associative.

Our last resort is to rewrite `fac` using the repetition operator (`*`) instead of left recursion. But in order to generate properly left-associative parse trees, we need somewhat complicated semantic actions,

```
mulOp = '*'                                          -> 'mul'
      | '/'                                          -> 'div',
fac   = num:x (mulOp:op num:y -> (x = [op, x, y]))* -> x
```

which makes this significantly less readable than the original left-recursive version. (The "decidedly imperative" version of the fac rule shown above uses OMeta's iteration operator (*) as a kind of loop construct, and a semantic action to update the x variable, which holds the parse tree, each time a new part of the expression is recognized.)

OMeta avoids these issues by extending PEGs with support for left recursion. While this does not make OMeta more powerful than PEGs, it does make it easier for programmers to express certain rules. This in turn increases OMeta's usefulness as a rapid prototyping tool. Details on how left recursion is supported in my OMeta implementations can be found in Chapter 3.

2.2.2.3 Parameterized Rules

OMeta's rules, unlike those in PEGs, may take any number of arguments. This feature can be used to implement a lot of functionality that would otherwise have to be built into the language. As an example, consider regular-expression-style *character classes*, which traditional PEG implementations support in order to spare programmers from the tedious and error-prone job of writing rules such as

```
lowerCase = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
          | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r
          | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
```

and instead allow them to write the more convenient

```
lowerCase = [a-z]
```

Using *parameterized rules*, i.e., rules with arguments, OMeta programmers can

```
eq       = ’=’        -> {kind: ’=’,   value: ’=’},
num      = digit+:ds  -> {kind: ’num’, value: parseInt(ds.join(’’))},
id       = letter+:ls -> {kind: ’id’,  value: ls.join(’’)},

scanner  = space* (eq | num | id),
token :k = scanner:t ?(t.kind == k) -> t.value,

assign   = token(’id’) token(’=’) token(’num’)
```

Figure 2.5: Combining scannerless and “scannerful” parsing with parameterized rules

write

```
  charRange :x :y = char:c ?(x <= c && c <= y) -> c
```

which is almost as convenient to use as character classes, e.g.,

```
  lowerCase = charRange(’a’, ’z’)
```

but much more flexible because it is defined by the programmer.

Hybrid “Scannerful” and Scannerless Parsing

The combination of parameterized rules and semantic predicates can be used to support a hybrid of (traditional) “scannerful” and *scannerless parsing* [SC89, Vis97], as shown in Figure 2.5:

- The `scanner` rule is essentially a lexical analyzer, i.e., it consumes the next token from the input stream. It represents a token as an object that contains two properties: its `kind` (e.g., `'num'`, if it is a number), and its `value` (e.g., `123`).
- The `token` rule, which takes a kind of token as an argument, retrieves the next token (using the `scanner` rule), and, if it is the right kind of token, returns its associated value.
- The `assign` rule defines the syntax of an assignment as an identifier token followed by an equal sign token and a number token.

I have found this idiom to be less error-prone than scannerless parsing (the only kind supported by PEGs), and yet just as expressive since each rule may access the character stream directly if desired.

To make this idiom more convenient to use, OMeta supports a syntactic sugar for invoking a user-defined rule called `token`, i.e., `token('=')` can be written as `"="`. This is effectively a kind of Meta-Object Protocol (MOP) [KRB91] that enables programmers to define the semantics of `"..."`. Using this syntactic sugar can result in remarkably readable rules. For example, with an implementation of `token` similar to the one in Figure 2.5, we could write:

```
condStmt = "if" "(" expr:c ")" stmt:tb "else" stmt:fb -> ...
```

The `Parser` grammar, which is part OMeta's "standard library", provides a default implementation of `token` that skips any number of spaces and then tries to match the characters that make up the string that it receives as an argument.

Pattern Matching on Rule Arguments

OMeta's parameterized rules can also pattern-match on their arguments. In fact,

```
charRange :x :y = ...
```

is actually shorthand for

```
charRange anything:x anything:y = ...
```

which means that `x` and `y` can be any kind of object.

This mechanism can be used to validate the arguments that are passed to a rule. For example, the following version of `charRange` ensures that both of its arguments are characters:

```
charRange char:x char:y = ...
```

Also, because any pattern (not just rule applications) can be used on the left-hand side of a rule, OMeta naturally supports the inductive style used to define functions in languages like Haskell and ML:

```
fact 0  =                 -> 1,
fact :n = fact(n - 1):m -> (n * m)
```

(When a rule has multiple definitions, as in the example above, each definition is tried in order until the first one succeeds.)

2.2.2.4 Higher-Order Rules

OMeta also provides a mechanism for implementing higher-order rules, i.e., rules that take other rules as arguments. This is supported by the `apply` rule, which takes as an argument the name of a rule, and invokes it. In other words, `apply('expr')` is equivalent to `expr`.

As an example, the rule

```
listOf :p = apply(p) ("," apply(p))*
```

can be used to recognize both comma-delimited lists of expressions

```
listOf('expr')
```

and lists of names

```
listOf('name')
```

OMeta's Parameterized and higher-order rules bring the expressive power of parser combinator libraries [HM98, LM01] to the world of PEGs.

2.2.3 A Note on Memoization

Packrat parsers are parsers for PEGs that are able to guarantee linear parse times while supporting backtracking and unlimited look-ahead "by saving all intermediate parsing results as they are computed and ensuring that no result is evaluated more than once." [For02a] While OMeta is based on PEGs, it does not necessarily have to be implemented using packrat-style memoization.

My implementations do in fact memoize the results of rules without arguments, but in order to keep their memory footprints small, I chose not to memoize the results of rules with arguments.

While the linear time guarantee that comes with memoization is certainly desirable, some of my experiments with PEGs indicate that the overhead of memoization may outweigh its benefits for the common case, where backtracking is limited. These trade-offs are discussed in detail in a paper by Becket and Somogyi [BS08], and are orthogonal to the ideas discussed in this chapter.

2.3 O is for Object-Oriented

Programming in OMeta would be very frustrating if all rules were defined in the same namespace: two grammars might unknowingly use the same name for two rules that have different purposes, and one of them would certainly stop working! (Picture one sword-wielding grammar decapitating another, *Highlander*-style: "There can be only one!")

A *class* is a special kind of namespace that comes with a huge bonus: a familiar and well-understood extensibility mechanism. By making OMeta an object-oriented language (i.e., making grammars analogous to classes and rules analogous to methods), several interesting things become possible.

```
ometa EJSParser <: JSParser {
  isKeyword :x = ?(x == ’say’)
               | ^isKeyword(x),
  stmt         = "say" expr:x sc -> [’call’, [’get’, ’alert’], x]
               | ^stmt
}
```

Figure 2.6: Extending a JavaScript parser with the `say` statement (`say ’hello’;` is equivalent to `alert(’hello’);`)

2.3.1 Quick and Easy Language Extensions

Programming language researchers often implement extensions to existing languages in order to experiment with new ideas in a real-world setting. Consider the task of adding a new kind of statement to JavaScript, for example; Figure 2.6 shows how this might be done in OMeta by creating a new parser that inherits from an existing JavaScript parser and overrides the rules for parsing statements, and detecting keywords. Note that the rule application `^stmt` behaves exactly like a super-send in traditional OO languages.

2.3.2 Extensible Pattern Matching

The OMeta parser (the front-end of my implementation) is written in OMeta itself. It translates the code for a rule, which is a stream of characters, into a parse tree in which sequences are represented as `And` nodes, choices as `Or` nodes, rule applications as `App` nodes, and so on. As an example, the parse tree generated for the body of the rule

```
  foo = bar baz
```

is

```
  [’Or’, [’And’, [’App’, ’foo’], [’App’, ’bar’]]]
```

which is later transformed by the OMeta compiler into the code that implements that rule.

My simple-minded parser always produces an Or node for the body of a rule, even when there is only one alternative (as in the example above). This is wasteful, and can degrade the performance of OMeta programs. After all, the ordered choice operation must store the current position of the parser's input stream so that when a choice fails, it can backtrack before trying the next choice.

Expressions like

```
['Or', ['Or', ['App', 'x'], ['App', 'y']], ['App', 'z']]
```

are also needlessly inefficient. Because Ors are associative, the expression above can be flattened to the more efficient

```
['Or', ['App', 'x'], ['App', 'y'], ['App', 'z']]
```

My implementation makes use of several such transformations in order to improve the performance of OMeta programs. Each of these is implemented in OMeta itself, using an idiom similar to the visitor design pattern. Figure 2.7 shows (i) the NullOptimization grammar, which visits each node in the parse tree of a rule, and (ii) the OROptimization grammar, which inherits the traversal code from NullOptimization and overrides the opt rule in order to implement the two optimizations for Or nodes discussed in this section. (In JavaScript, Array's concat method concatenates two or more arrays.)

I have implemented several other transformations, including left factoring and a *jumptable*-based optimization that allows choices such as

```
['Or', ['App', 'char', 97],
       ['App', 'char', 98],
       ['App', 'char', 99]]
```

```
ometa NullOptimization {
  opt = [’And’   opt*:xs]          -> [’And’].concat(xs)
      | [’Or’    opt*:xs]          -> [’Or’].concat(xs)
      | [’Form’  opt*:xs]          -> [’Form’].concat(xs)
      | [’Not’   opt:x]            -> [’Not’,   x]
      | [’Many’  opt:x]            -> [’Many’,  x]
      | [’Many1’ opt:x]            -> [’Many1’, x]
      | [’Set’   anything:n opt:v] -> [’Set’,   n, v]
      | anything
}

ometa OROptimization <: NullOptimization {
  opt    = [’Or’ opt:x]               -> x
         | [’Or’ inside:xs]           -> [’Or’].concat(xs)
         | ^opt,
  inside = [’Or’ inside:xs] inside:ys -> xs.concat(ys)
         | opt:x            inside:xs -> [x].concat(xs)
         | empty                      -> []
}
```

Figure 2.7: Extensible pattern matching in OMeta

to be evaluated in constant time.[3]

2.3.3 Stateful Pattern Matching

OMeta’s grammars may have any number of instance variables. These variables are initialized by the `initialize` method, which is invoked automatically when a new instance of the grammar is created. (A grammar object must be instantiated before it can be used to match a value with a start symbol, i.e., a rule. This is done by sending the grammar the `match` message, e.g., `G.match([1, 2, 3], ’myList’)`.)

Using an earlier version of OMeta, I implemented a parser for a significant subset of Python [Ros95] that used an instance variable to hold a stack of indentation levels. This stack was used for implementing Python’s *offside rule*, which enables programs

[3] These transformations are part of my OMeta/COLA implementation.

```
ometa Calc <: Parser {
  var      = letter:x                -> x,
  num      = num:n digit:d           -> (n * 10 + d.digitValue())
           | digit:d                 -> d.digitValue(),
  priExpr  = spaces var:x            -> self.vars[x]
           | spaces num:n            -> n
           | "(" expr:r ")"          -> r,
  mulExpr  = mulExpr:x "*" priExpr:y -> (x * y)
           | mulExpr:x "/" priExpr:y -> (x / y)
           | priExpr,
  addExpr  = addExpr:x "+" mulExpr:y -> (x + y)
           | addExpr:x "-" mulExpr:y -> (x - y)
           | mulExpr,
  expr     = var:x     "=" expr:r    -> (self.vars[x] = r)
           | addExpr,
  doit     = (expr:r)* spaces end    -> r
}

Calc.initialize = function() { this.vars = {}; }
```

Figure 2.8: A calculator

to use indentation instead of brackets for delimiting lexical scopes.

Another example of OMeta's stateful grammars is Calc, the calculator grammar shown in Figure 2.8. This grammar is not just a parser; it is a complete *interpreter* for arithmetic expressions with variables (the interpreting is done by the rules' semantic actions). Calc's instance variable vars holds a symbol table that maps variable names to their current values, and is modified by the semantic action of the expr rule (self.vars[x] = r). The following transcript shows my calculator in action:

```
> 3+4*5
23
> x = y = 2
2
> x = x * 7
14
> y
2
```

Note that OMeta does not attempt to undo the side effects of semantic actions while backtracking; for some semantic actions, like printing characters to the console, this would be impossible. Programmers implementing stateful pattern matchers must therefore write their semantic actions carefully. (The case study of Chapter 4 of this dissertation provides a solution to this problem.)

2.3.4 Foreign Rule Invocation

Consider the task of implementing `OMetaJSParser`, a parser for a language that is the union of OMeta and JavaScript. Suppose that we already have parsers for both of these languages; they are called `OMetaParser` and `JSParser`, respectively.

Using OMeta's single inheritance mechanism, we could either

1. declare `OMetaJSParser` as a sub-grammar of `OMetaParser` and duplicate (i.e., re-implement) the rules of `JSParser`, or

2. declare `OMetaJSParser` as a sub-grammar of `JSParser` and duplicate the rules of `OMetaParser`,

but neither of these choices is satisfactory, since it results in code bloat and creates a versioning problem, e.g., subsequent changes to `JSParser` will not automatically carry over to the the `OMetaJSParser` parser resulting from (1). Making

OMetaJSParser inherit from both OMetaParser and JSParser would also be a bad idea, since name clashes would most likely result in incorrect behavior.[4]

A much better solution to this problem is OMeta's *foreign rule invocation* mechanism, which allows a grammar to "lend" its input stream to another grammar in order to make use of a foreign rule. This mechanism is accessed via the foreign rule, which takes as arguments the foreign parser and rule name, as shown below:

```
ometa OMetaJSParser {
  topLevel = foreign(OMetaParser, 'grammar')
           | foreign(JSParser,    'srcElem')
}
```

Foreign rule invocation enables programmers to combine grammars in interesting ways without having to worry about name clashes.

2.4 Core-OMeta: an Operational Semantics for OMeta-Style Pattern Matching

This section provides an operational semantics for Core-OMeta, an extension to PEGs that supports OMeta-style pattern matching on arbitrary datatypes.

2.4.1 Definitions

A Core-OMeta grammar G is a finite set of rules; each rule $r \in G$ has the form $A \leftarrow e$, i.e., it is a pair consisting of a nonterminal (the name of the rule) and its associated parsing expression.

Core-OMeta's language of parsing expressions, shown in Figure 2.9, extends Bryan Ford's original formalism for PEGs [For04] with support for:

[4] OMeta does not support multiple inheritance.

e	::=	ε	(empty)
	|	a	(an atom)
	|	A	(a non-terminal)
	|	$e_1\ e_2$	(sequencing)
	|	e_1 / e_2	(alternation)
	|	e^*	(iteration)
	|	$!e$	(negation)
	|	e`:x`	(binding)
	|	$\rightarrow t$	(semantic action)
	|	`[`e`]`	(list pattern)

Figure 2.9: The language of parsing expressions for Core-OMeta

v	::=	a	(an atomic value, e.g., a character)
	|	`[` v^* `]`	(a list of values)
	|	**none**	(no value)

Figure 2.10: The language of values, used as input and output in Core-OMeta

- bindings and semantic actions, and
- pattern matching on structured data (i.e., lists).

Together, these extensions make Core-OMeta a suitable language for implementing transformations on unstructured data (e.g., parsers) as well as structured data (e.g., visitors).

It is important to note that Core-OMeta's bindings, like those in OMeta, are rule-local (i.e., their scope is the entire rule in which they are defined), and mutable (e.g., binding `x` twice results in changing the value of `x`).

Figures 2.10 and 2.11 define the language of values (used as input and output to Core-OMeta grammars) and the language of terms (used for writing semantic actions), respectively.

$$
\begin{array}{rcll}
t & ::= & a & \\
  & | & \texttt{[}\, t^{*}\, \texttt{]} & \\
  & | & \textbf{none} & \\
  & | & \texttt{x} & \text{(a variable)}
\end{array}
$$

Figure 2.11: The language of terms, used for writing semantic actions

2.4.2 Semantics

2.4.2.1 Notation

- Let $x, y, z \in v^*$ (i.e., sequences of zero or more values).
- Let $|x|$ be the length of a sequence x.
- Let a and b be atomic values (e.g., characters).
- Let μ be a finite set of bindings from variables to values (a store); each binding has the form $\texttt{x} \rightarrow v$.
- $[\texttt{x} \rightarrow v]\mu$ means "the store that maps $\texttt{x}$ to v and all other variables to the same values as μ".
- $(e, xy, \mu) \Rightarrow (v, y, \mu')$ means that the parsing expression e successfully matches (consumes) the sequence of values x, leaving y on the input stream, and producing the value v as a result. μ and μ' represent the store (i.e., the values of the bound variables) before and after the pattern is applied to the input, respectively.
- $(e, x, \mu) \Rightarrow (\textbf{FAIL}, \mu')$ means that the parsing expression e does not match the input. (Note that the bindings may have changed as a result of the failed match.)

2.4.2.2 Basic PEG Functionality

The following evaluation rules describe the semantics of PEGs. They are equivalent to Ford's original formalism.

Empty

The empty pattern ε always succeeds without consuming any input (i.e., it leaves the input stream untouched), and yields the value **none**.

$$\frac{}{(\varepsilon, x, \mu) \Rightarrow (\mathbf{none}, x, \mu)} \qquad \text{(Empty)}$$

Atomic Values (e.g., characters, numbers, booleans)

The atomic value pattern a succeeds only if the input stream is not empty and its first element is equal to the atomic value in the pattern. A successful match yields the value that was consumed.

$$\frac{}{(a, ax, \mu) \Rightarrow (a, x, \mu)} \qquad \text{(Atom-Success)}$$

$$\frac{a \neq b}{(a, bx, \mu) \Rightarrow \ (\mathbf{FAIL}, \mu)} \qquad \text{(Atom-Failure-1)}$$

$$\frac{}{(a, [x]y, \mu) \Rightarrow \ (\mathbf{FAIL}, \mu)} \qquad \text{(Atom-Failure-2)}$$

$$\frac{}{(a, \mathbf{none}x, \mu) \Rightarrow \ (\mathbf{FAIL}, \mu)} \qquad \text{(Atom-Failure-3)}$$

$$\frac{|x| = 0}{(a, x, \mu) \Rightarrow \ (\mathbf{FAIL}, \mu)} \qquad \text{(Atom-Failure-4)}$$

Note that the only kind of atomic value supported by PEGs is the character. Since this formalism models OMeta's pattern matching (not just PEGs), it makes sense to have a broader notion of atomic value that includes numbers, booleans, etc.

Nonterminal

The nonterminal (or rule application) A succeeds if its associated pattern succeeds when applied to the input. Note that the "body of the rule" is applied to the input with a fresh store ($\emptyset$) to ensure that bindings are (statically) scoped to the rules in which they appear.

$$\frac{\begin{array}{c} A \leftarrow e \in G \\ (e, xy, \emptyset) \Rightarrow (v, y, \mu_2') \end{array}}{(A, xy, \mu_1) \Rightarrow (v, y, \mu_1)} \qquad \text{(Nonterminal-Success)}$$

$$\frac{\begin{array}{c} A \leftarrow e \in G \\ (e, xy, \emptyset) \Rightarrow (\mathbf{FAIL}, \mu_2') \end{array}}{(A, xy, \mu_1) \Rightarrow (\mathbf{FAIL}, \mu_1)} \qquad \text{(Nonterminal-Failure)}$$

Sequence

The sequence pattern $e_1\ e_2$ succeeds only if e_1 matches a prefix of the input, and e_2 matches a prefix of whatever is left on the input stream after e_1 is applied. A successful match yields the same value as e_2.

$$\frac{\begin{array}{c} (e_1, xy, \mu) \Rightarrow (v_1, y, \mu') \\ (e_2, y, \mu') \Rightarrow ans \end{array}}{(e_1\ e_2, xy, \mu) \Rightarrow ans} \qquad \text{(Sequence-Next)}$$

$$\frac{(e_1, x, \mu) \Rightarrow\ (\mathbf{FAIL}, \mu')}{(e_1\ e_2, x, \mu) \Rightarrow\ (\mathbf{FAIL}, \mu')} \qquad \text{(Sequence-Failure)}$$

Alternation

The alternation pattern e_1 / e_2 succeeds if e_1 matches the input (in which case e_2 is skipped); otherwise (i.e., if e_1 fails), the result of the alternation pattern is the same as the result of e_2.

$$\frac{(e_1, xy, \mu) \Rightarrow (v_1, y, \mu')}{(e_1 \ / \ e_2, xy, \mu) \Rightarrow (v_1, y, \mu')} \qquad \text{(Alternation-Success)}$$

$$\frac{\begin{array}{c}(e_1, x, \mu) \Rightarrow \ (\textbf{FAIL}, \mu') \\ (e_2, x, \mu') \Rightarrow ans\end{array}}{(e_1 \ / \ e_2, x, \mu) \Rightarrow ans} \qquad \text{(Alternation-Next)}$$

Note that e_1's side effects—i.e., whatever changes it may have made to the store—are not rolled back before e_2 is evaluated.

Iteration

The iteration pattern e^* successively applies e to the input, only stopping when that results in a match failure. It yields a (possibly empty) list containing the results of each successful application.

$$\frac{\begin{array}{c}(e, xyz, \mu) \Rightarrow (v, yz, \mu') \\ (e^*, yz, \mu') \Rightarrow (\overline{v}, z, \mu'') \\ v_{ans} = append(\texttt{[}v\texttt{]}, \overline{v})\end{array}}{(e^*, xyz, \mu) \Rightarrow (v_{ans}, z, \mu'')} \qquad \text{(Iteration-Repetition)}$$

$$\frac{(e, x, \mu) \Rightarrow \ (\textbf{FAIL}, \mu')}{(e^*, x, \mu) \Rightarrow (\texttt{[]}, x, \mu')} \qquad \text{(Iteration-Termination)}$$

Negation (Syntactic Predicate)

The negation pattern $!e$ succeeds if the application of e to the input results in a match failure, in which case it consumes no input and yields the value **none**. It fails if the application of e succeeds.

$$\frac{(e, x, \mu) \Rightarrow (\mathbf{FAIL}, \mu')}{(!e, x, \mu) \Rightarrow (\mathbf{none}, x, \mu')} \qquad \text{(Negation-Success)}$$

$$\frac{(e, xy, \mu) \Rightarrow (v, y, \mu')}{(!e, xy, \mu) \Rightarrow (\mathbf{FAIL}, \mu')} \qquad \text{(Negation-Failure)}$$

2.4.2.3 Bindings and Semantic Actions

Next, I extend this formalism with support for bindings to variables and semantic actions, which were not included in Ford's original formalism.

Binding

The binding pattern $e\texttt{:x}$ succeeds only if e succeeds when applied to the input. It yields the same value as e, but also modifies the store in order to bind the variable `x` to that value.

$$\frac{(e, xy, \mu) \Rightarrow (v, y, \mu')}{(e\texttt{:x}, xy, \mu) \Rightarrow (v, y, [\texttt{x} \rightarrow v]\mu')} \qquad \text{(Binding-Success)}$$

$$\frac{(e, x, \mu) \Rightarrow (\mathbf{FAIL}, \mu')}{(e\texttt{:x}, x, \mu) \Rightarrow (\mathbf{FAIL}, \mu')} \qquad \text{(Binding-Failure)}$$

Semantic Action

The semantic action $\rightarrow t$ always succeeds without consuming any input. It yields the value obtained from evaluating the term t in the context of the current store. The semantics of term evaluation is given by the relation $eval(t, \mu) = v$, shown below:

$$\frac{}{eval(a, \mu) = a} \quad \text{(Eval-Atom)}$$

$$\frac{}{eval(\mathbf{none}, \mu) = \mathbf{none}} \quad \text{(Eval-None)}$$

$$\frac{\mathtt{x} \to v \in \mu}{eval(\mathtt{x}, \mu) = v} \quad \text{(Eval-Var)}$$

$$\frac{\begin{array}{c} eval(t_1, \mu, v_1) \\ \vdots \\ eval(t_n, \mu, v_n) \end{array}}{eval(\mathtt{[}t_1 \ldots t_n\mathtt{]}, \mu) = \mathtt{[}v_1 \ldots v_n\mathtt{]}} \quad \text{(Eval-List)}$$

And now the evaluation rule for semantic actions is straightforward:

$$\frac{eval(t, \mu) = v}{(\to t, x, \mu) \Rightarrow (v, x, \mu)} \quad \text{(Semantic-Action)}$$

2.4.3 List (Tree) Matching

Last but not least, I extend my formalism with support for pattern matching on lists (trees). A list pattern succeeds only if the input stream is not empty and its first element is a list all of whose elements are matched by e.

$$\frac{\begin{array}{c} (e, xy, \mu) \Rightarrow (v, y, \mu') \\ |y| = 0 \end{array}}{(\mathtt{[e]}, \mathtt{[}xy\mathtt{]}z, \mu) \Rightarrow (\mathtt{[}x\mathtt{]}, z, \mu')} \quad \text{(List-Success)}$$

$$\frac{\begin{array}{c}(e, xy, \mu) \Rightarrow (v, y, \mu') \\ |y| > 0\end{array}}{(\texttt{[e]}, \texttt{[}xy\texttt{]}z, \mu) \Rightarrow (\mathbf{FAIL}, \mu')} \tag{List-Failure-1}$$

$$\frac{(e, x, \mu) \Rightarrow (\mathbf{FAIL}, \mu')}{(\texttt{[e]}, \texttt{[}x\texttt{]}y, \mu) \Rightarrow (\mathbf{FAIL}, \mu')} \tag{List-Failure-2}$$

$$\frac{}{(\texttt{[e]}, ax, \mu) \Rightarrow (\mathbf{FAIL}, \mu)} \tag{List-Failure-3}$$

$$\frac{}{(\texttt{[e]}, \mathbf{none}x, \mu) \Rightarrow (\mathbf{FAIL}, \mu)} \tag{List-Failure-4}$$

$$\frac{|x| = 0}{(\texttt{[e]}, x, \mu) \Rightarrow (\mathbf{FAIL}, \mu)} \tag{List-Failure-5}$$

Note that the result of a successful match on a list object (given by List-Success, above) is not the result of the pattern that matched its contents, but rather the list object itself.

2.5 Related Work

The kind of pattern matching found in functional languages [Jon87]—"ML-style pattern matching"—can be used to define functions in an inductive style and also to deconstruct values of algebraic/structured types, both of which are useful for writing programs that manipulate tree-structured data (e.g., AST transformations). *Regular expression pattern matching* [HP01] extends ML-sytle pattern matching with support for regular expression operators such as repetition and alternation, which makes it an even more expressive mechanism for manipulating tree-structured data (XML, in particular). Both of these forms of pattern matching interact nicely with static typing, and support static checks for exhaustiveness and redundancy. OMeta's ability to manipulate unstructured data and its support for semantic predicates make it even more expressive than regular expression pattern matching, but also preclude these useful static checks.

There are also many orthogonal research directions in pattern matching, some of which may interact with OMeta in useful ways. Wadler's *views* [Wad87], for example, enable programmers to provide a "virtual representation" of their data that can be pattern-matched against without exposing any implementation details. This idea could be used to make OMeta's list patterns even more general, since they might match objects that are not actually lists, but instead expose a list-like representation of themselves (perhaps by implementing a small number of basic list operations). Another example is Chase's technique for improving the performance of bottom-up tree pattern matching [Cha87], which may be useful for implementing OMeta more efficiently (although, because OMeta supports *top-down* pattern matching, this may not be the case).

Some existing programming language implementation frameworks are based in part on pattern matching. Stratego/XT [Vis01, BKV08], for example, has as its central component a language (Stratego) that supports a form of metaprogramming with a combination of rewrite rules, which are specified using pattern matching, and strategies that are used to control the application of those rules. Because Stratego's rewrite rules can only manipulate ASTs, the framework includes an additional language, SDF, for implementing parsers. Another related language implementation framework is the ANTLR parser generator [PQ95], which comes bundled with a *tree parser generator* [Par94] that can be used to implement simple transformations on ANTLR-generated ASTs. OMeta's notion of general-purpose pattern matching was designed to make it possible for the same language to be used to implement all of the phases of a compiler, thus avoiding the complexities and steep learning curve that result from using multiple tools in language implementation.

The separation of rewrite rules and strategies in Stratego is an interesting design decision; it enables a single set of rewrite rules to be used in multiple transformations. In OMeta, rules are responsible for rewriting terms as well as driving traversals through ASTs (the latter is achieved with rules that recursively invoke other rules, which is not supported in Stratego). This design results in a more minimalist foundation for metaprogramming and supports a style that is more familiar to programmers, albeit at the cost of some reusability.

Polyglot [NCM03] and JastAdd [EH07] are extensible compiler frameworks that have been used by programming language researchers to build a large number of Java extensions. (I myself have used Polyglot to implement compilers for LazyJ [War07] and eJava [WSM06]). Unfortunately, the large size and complexity of these frameworks makes them uninviting to potential users who are not programming language experts (e.g., an expert in another area who wishes to implement a domain-specific language), and too "heavy-handed" for rapidly prototyping small extensions. Moreover, Polyglot is written in Java using various conventions for extensibility; these are easy to violate accidentally, since the implementer of an extension is required to adhere to them manually, i.e., without any support from the language.

JastAdd, which is based on Rewritable Reference Attributed Grammars [EH04], provides conditional rewrite rules as a means to transform ASTs. In this model, the application of a rewrite rule may cause the conditions of one or more other rules to be satisfied, thereby triggering those rules. This enables programmers to implement a complex transformation as a series of simpler transformations, although special care must be taken in order to ensure termination. While programmers may also express this kind of sequential composition in OMeta, it is impossible to do without making the order of the constituent rules explicit.

My work on OMeta began when I implemented my own version of Val Schorre's META-II [Sch64]: a simple yet practical recognition-based compiler-writing language that could be implemented in itself in roughly a page of code. META-II was a wonderful tool, but it had significant limitations:

- it did not support backtracking, which made it necessary for the programmer to do a large amount of left-factoring in rules, and
- its analog of semantic actions were `PRINT` commands, which meant that compilers had to generate code *while* recognizing programs. The resulting programs were usually interpreted by a special-purpose virtual machine which had been implemented specially for the language being compiled.

Once I added backtracking and semantic actions to my language, it became equivalent in power to Bryan Ford's Parsing Expression Grammars (PEGs) [For04].

OMeta's PEG foundation makes it a close relative of packrat parser generators [For02a]. Rats! [Gri06] in particular, supports a notion of "modular syntax" with a form of inheritance, which (like in OMeta) can be used to create "subgrammars" that may override the rules of their "supergrammars". But OMeta is not a parser generator; it is a programming language whose control structure is based on PEGs. And unlike previous PEG-based tools, which operate only on streams of characters, OMeta extends PEGs with support for arbitrary datatypes.

OMeta's ability to pattern match over arbitrary datatypes and the notion of parameterized rules were both inspired by LISP70 [TES73], which used pattern matching for pattern-directed computation (as in ML) as well as extending its own syntax. But unlike OMeta, LISP70 relied on an external lexical analyzer that could not be modified by user programs, did not support object-oriented extensibility mechanisms, and was never fully implemented.

Parameterized and higher-order rules can also be found in parser combinator libraries [HM98, LM01]. But OMeta is a language, not a library, and its specialized syntax and object-oriented features make OMeta grammars more readable and more extensible than those written using parser combinator libraries.

2.6 Conclusions and Future Work

I have shown that OMeta's general-purpose pattern matching enables programmers to easily implement lexical analyzers, parsers, visitors, etc. This makes OMeta particularly well-suited as a medium for experimenting with new designs for programming languages and extensions to existing languages.

In future work, I would like to improve the performance of my OMeta implementations; it should be possible for them to be competitive with state-of-the-art packrat parser implementations such as Robert Grimm's *Rats!* [Gri06].

CHAPTER 3

Left Recursion Support for Packrat Parsers

Packrat parsing [For02a] offers several advantages over other parsing techniques, such as the guarantee of linear parse times while supporting backtracking and unlimited look-ahead. Unfortunately, the limited support for left recursion in packrat parser implementations makes them difficult to use for a large class of grammars (Java's, for example). This chapter presents a modification to the memoization mechanism used by packrat parser implementations that makes it possible for them to support (even indirectly or mutually) left-recursive rules. While it is possible for a packrat parser with my modification to yield super-linear parse times for some left-recursive grammars, my experiments show that this is not the case for typical uses of left recursion.

3.1 Introduction

Packrat parsers [For02a] are an attractive choice for programming language implementers because:

- They provide "the power and flexibility of backtracking and unlimited look-ahead, but nevertheless [guarantee] linear parse times." [For02a]
- They support syntactic and semantic predicates.
- They are easy to understand: because packrat parsers only support *ordered choice*—as opposed to *unordered choice*, as found in Context-Free Grammars

(CFGs)—there are no ambiguities and no shift-reduce/reduce-reduce conflicts, which can be difficult to resolve.

- They impose no separation between lexical analysis and parsing. This feature, sometimes referred to as *scannerless parsing* [SC89, Vis97], eliminates the need for *moded lexers* [VS07] when combining grammars (e.g., in Domain-Specific Embedded Language (DSEL) implementations).

Unfortunately, "like other recursive descent parsers, packrat parsers cannot support left-recursion" [Gri06], which is typically used to express the syntax of left-associative operators. To better understand this limitation, consider the following rule for parsing expressions:

```
expr = expr "-" num | num
```

Note that the first alternative in `expr` begins with `expr` itself. Because the choice operator in packrat parsers (denoted here by "`|`") tries each alternative in order, this recursion will never terminate: an application of `expr` will result in another application of `expr` without consuming any input, which in turn will result in yet another application of `expr`, and so on. The second choice—the non-left-recursive case—will never be used.

We could change the order of the choices in `expr`,

```
expr = num | expr "-" num
```

but to no avail. Since all valid expressions begin with a number, the second choice—the left-recursive case—would never be used. For example, applying the `expr` rule to the input "`1-2`" would succeed after consuming only the "`1`", and leave the rest of the input, "`-2`", unprocessed.

Some packrat parser implementations, including *Pappy* [For02b] and *Rats!* [Gri06], circumvent this limitation by automatically transforming *directly left-recursive* rules into equivalent non-left-recursive rules. This technique is called *left recursion elimination.* As an example, the left-recursive rule above can be transformed to

```
expr = num ("-" num)*
```

which is not left-recursive and therefore can be handled correctly by a packrat parser. Note that the transformation shown here is overly simplistic; a suitable transformation must preserve the left-associativity of the parse trees generated by the resulting non-left-recursive rule, as well as the meaning of the original rule's *semantic actions.*

Now consider the following minor modification to the original grammar, which has no effect on the language accepted by `expr`:

```
x    = expr
expr = x "-" num | num
```

When given this grammar, the *Pappy* packrat parser generator reports the following error message:

```
Illegal left recursion: x -> expr -> x
```

This happens because `expr` is now *indirectly* left-recursive, and *Pappy* does not support indirect left recursion (also referred to as mutual left recursion). In fact, to the best of my knowledge, none of the currently-available packrat parser implementations supports indirectly left-recursive rules.

Although this example is certainly contrived, indirect left recursion does in fact arise in real-world grammars. For instance, Roman Redziejowski discusses the difficulty of implementing a packrat parser for Java [Gos05], whose *Primary* rule (for expressions) is indirectly left-recursive with five other rules [Red08]. While programmers can always refactor grammars manually in order to eliminate indirect left recursion, doing so is tedious and error-prone, and in the end it is generally difficult to be convinced that the resulting grammar is equivalent to the original.

This chapter presents a modification to the memoization mechanism used by packrat parser implementations that enables them to support both direct and indirect left recursion directly (i.e., without first having to transform rules). While it is possible for a packrat parser with my modification to yield super-linear parse times for some left-recursive grammars, my experiments (Section 3.5) show that this is not the case for typical uses of left recursion.

The rest of this chapter is structured as follows. Section 3.2 gives a brief overview of packrat parsing. Section 3.3 describes my modification to the memoization mechanism, first showing how direct left recursion can be supported, and then extending the approach to support indirect left recursion. Section 3.4 validates this work by showing that it enables packrat parsers to support a grammar that closely mirrors Java's heavily left-recursive *Primary* rule. Section 3.5 discusses the effects of my modification on parse times. Section 3.6 discusses related work, and Section 3.7 concludes.

3.2 An Overview of Packrat Parsing

Packrat parsers are able to guarantee linear parse times while supporting backtracking and unlimited look-ahead "by saving all intermediate parsing results as they are computed and ensuring that no result is evaluated more than once" [For02a]. For example, consider what happens when the rule

```
expr = num "+" num
     | num "-" num
```

(where `num` matches a sequence of digits) is applied to the input "`1234-5`".

Since choices are always evaluated in order, the parser begins by trying to match the input with the pattern

```
num "+" num
```

The first term in this pattern, `num`, successfully matches the first four characters of the input stream ("`1234`"). Next, the parser attempts to match the next character on the input stream, "`-`", with the next term in the pattern, `"+"`. This match fails, and thus we backtrack to the position at which the previous choice started (0) and try the second alternative:

```
num "-" num
```

At this point, a conventional top-down backtracking parser would have to apply `num` to the input, just like we did while evaluating the first alternative. However, because packrat parsers *memoize* all intermediate results, no work is required this time around: the parser already knows that `num` succeeds at position 0, consuming the first four characters. It suffices to update the current position to 4 and carry on evaluating the remaining terms. The next pattern, `"-"`, successfully matches the next character, and thus the current position is incremented to 5. Finally, `num` matches and consumes the "`5`", and the parse succeeds.

Intermediate parsing results are stored in the parser's memo table, which we shall model as a function

$$\textsc{Memo} : (\textsc{Rule}, \textsc{Pos}) \rightarrow \textsc{MemoEntry}$$

where

$$\textsc{MemoEntry} : (ans : \text{AST}, pos : \textsc{Pos})$$

In other words, Memo maps a rule-position pair (R, P) to a tuple consisting of

- the AST (or the special value FAIL[1]) resulting from applying R at position P, and

[1] Failures are also memoized in order to avoid doing unnecessary work when backtracking occurs.

```
APPLY-RULE($R, P$)
    let $m$ = MEMO($R, P$)
    if $m$ = NIL
        then let $ans$ = EVAL($R.body$)
                $m \leftarrow$ new MEMOENTRY($ans, Pos$)
                MEMO($R, P$) $\leftarrow m$
                return $ans$
        else $Pos \leftarrow m.pos$
                return $m.ans$
```

Figure 3.1: The original APPLY-RULE procedure

- the position of the next character on the input stream.

or NIL, if there is no entry in the memo table for the given rule-position pair.

The APPLY-RULE procedure (see Figure 3.1), used in every rule application, ensures that no rule is ever evaluated more than once at a given position. When rule *R* is applied at position *P*, APPLY-RULE consults the memo table. If the memo table indicates that *R* was previously applied at *P*, the appropriate parse tree node is returned, and the parser's current position is updated accordingly. Otherwise, APPLY-RULE evaluates the rule, stores the result in the memo table, and returns the corresponding parse tree node.

By using the memo table as shown in this section, packrat parsers are able to support backtracking and unlimited look-ahead while guaranteeing linear parse times. In the next section, I present modifications to the memo table and the APPLY-RULE procedure that make it possible for packrat parsers to support left recursion.

3.3 Adding Support for Left Recursion

In Section 3.1, I showed informally that the original version of the `expr` rule,

```
expr = expr "-" num | num
```

causes packrat parsers to go into infinite recursion. We now revisit the same example, this time from Section 3.2's more detailed point of view.

Consider what happens when `expr` is applied to the input "`1-2-3`". Since the parser's current position is initially 0, this application is encoded as APPLY-RULE(`expr`, 0). APPLY-RULE, shown in Figure 3.1, begins by searching the parser's memo table for the result of `expr` at position 0. The memo table is initially empty, and thus MEMO(`expr`, 0) evaluates to NIL, indicating that `expr` has not yet been used at position 0. This leads APPLY-RULE to evaluate the body of the `expr` rule, which is made up of two choices. The first choice begins with `expr`, which, since the parser's current position is still 0, is encoded as the familiar APPLY-RULE(`expr`, 0). At this point, the memo table remains unchanged and thus we are back exactly where we started! The parser is doomed to repeat the same steps forever, or more precisely, until the computer eventually runs out of stack space.

The rest of this section presents a solution to this problem. First, I modify the algorithm to make left-recursive applications fail, in order to avoid infinite loops. I then build on this extension to properly support direct left recursion. Extending this idea to support indirect left recursion is conceptually straightforward; I present the intuition for this in Section 3.3. Finally, Section 3.4 focuses on the operational details of this extension.

3.3.1 Avoiding Infinite Recursion in Left-Recursive Rules

A simple way to avoid infinite recursion is for APPLY-RULE to store a result of FAIL in the memo table *before* it evaluates the body of a rule, as shown in Figure 3.2. This has the effect of making all left-recursive applications (both direct and indirect) fail.

```
Apply-Rule($R, P$)
  let $m$ = Memo($R, P$)
  if $m$ = nil
    then $m \leftarrow$ new MemoEntry(Fail, $P$)          *
         Memo($R, P$) $\leftarrow m$                       *
         let $ans$ = Eval($R.body$)
         $m.ans \leftarrow ans$                            *
         $m.pos \leftarrow Pos$                            *
         return $ans$
    else $Pos \leftarrow m.pos$
         return $m.ans$
```

Figure 3.2: Avoiding non-termination by making left-recursive applications fail (lines marked with * are either new or have changed since the previous version)

Consider what happens when `expr` is applied to the input "`1-2-3`" using the new version of Apply-Rule. Once again this application is encoded as Apply-Rule(`expr`, 0). Apply-Rule first updates the memo table with a result of fail for `expr` at position 0, then goes on to evaluate the rule's body, starting with its first choice. The first choice begins with an application of `expr`, which, because the current position is still 0, is also encoded as Apply-Rule(`expr`, 0). This time, however, Apply-Rule *will* find a result in the memo table, and thus will not evaluate the body of the rule. And because that result is fail, the current choice will be aborted. The parser will then move on to the second choice, `num`, which will succeed after consuming the "`1`", and leave the rest of the input, "`-2-3`", unprocessed.

3.3.2 Supporting Direct Left Recursion

While this is clearly not `expr`'s intended behavior, the modified Apply-Rule procedure shown in Figure 3.2 was a step in the right direction. Consider the side-effects of the application of *expr* at position 0:

1. The parser's current position was updated to 1, and
2. The parser's memo table was updated with a mapping from (`expr`, 0) to (`expr` → `num` → `1`, 1).

The parse shown above avoided all left-recursive terms; I call it the *seed parse*.

Now, suppose we backtrack to position 0 and evaluate `expr`'s body one more time. Note that unlike evaluating APPLY-RULE(`expr`, 0), which would simply retrieve the previous result stored in the memo table, evaluating the *body* of this rule (which we denote in pseudo-code as EVAL(*expr.body*)) will sidestep one level of memoization and begin to evaluate each of its choices. The first choice,

```
expr "-" num
```

begins with a left-recursive application, just like before. This time, however, that application succeeds because the memo table now contains the seed parse. Next, the terms `"-"` and `num` successfully match and consume the "`-`" and "`2`" on the input, respectively. If we update the memo table with the new answer and repeat these steps one more time, we will have parsed "`1-2-3`", the entire input stream!

I refer to this iterative process as *growing the seed*; Figure 3.3 shows GROW-LR, which implements the seed-growing algorithm. GROW-LR tries to grow the parse of rule *R* at position *P*, given the seed parse in the MEMOENTRY *M*.[2] Note that each time the rule's body is evaluated, the parser must backtrack to *P*; this is accomplished with the statement "$Pos \leftarrow P$". At the start of each iteration, *M* contains the last successful result of the left recursion. The loop's terminating condition, "ans = FAIL or $Pos \leq M.pos$", detects that no progress was made as a result of evaluating the rule's body. Once this condition is satisfied, the parser's current position is updated to the one associated with the last successful result.

[2] Lines A, B, and C, and the argument *H* can be ignored at this point.

```
GROW-LR($R, P, M, H$)
    ...                                                    ▷ line A
    while TRUE
        do
            $Pos \leftarrow P$
            ...                                            ▷ line B
            let $ans$ = EVAL($R.body$)
            if $ans$ = FAIL or $Pos \leq M.pos$
               then break
            $M.ans \leftarrow ans$
            $M.pos \leftarrow Pos$
    ...                                                    ▷ line C
    $Pos \leftarrow M.pos$
    return $M.ans$
```

Figure 3.3: GROW-LR: support for direct left recursion

```
APPLY-RULE($R, P$)
    let $m$ = MEMO($R, P$)
    if $m$ = NIL
       then let $lr$ = new LR(FALSE)                              *
            $m \leftarrow$ new MEMOENTRY($lr, P$)                  *
            MEMO($R, P$) $\leftarrow m$
            let $ans$ = EVAL($R.body$)
            $m.ans \leftarrow ans$
            $m.pos \leftarrow Pos$
            if $lr.detected$ and $ans \neq$ FAIL                   *
               then return GROW-LR($R, P, m$, NIL)               *
               else  return $ans$                                *
       else  $Pos \leftarrow m.pos$
            if $m.ans$ is LR                                     *
               then $m.ans.detected \leftarrow$ TRUE              *
                     return FAIL                                 *
               else  return $m.ans$                              *
```

Figure 3.4: Detecting left recursion and growing the seed with GROW-LR (lines marked with * are either new or have changed since the previous version)

GROW-LR can be used to compute the result of a left-recursive application. Before we can use it, however, we must be able to detect when a left recursion has occurred. I do this by introducing a new datatype, LR, and modifying MEMOENTRY so that LRs may be stored in *ans*,

LR : (*detected* : BOOLEAN)

MEMOENTRY : (*ans* : AST or LR, *pos* : POSITION)

and modifying APPLY-RULE as shown in Figure 3.4.

To detect left-recursive applications, APPLY-RULE memoizes an LR with *detected* = FALSE before evaluating the body of the rule. A left-recursive application of the same rule will cause its associated LR's *detected* field to be set to TRUE, and yield a result of FAIL. When an application is found to be left-recursive and it has a successful seed parse, GROW-LR is invoked in order to grow the seed into the rule's final result.

The modifications shown in Figures 3.3 and 3.4 enable packrat parsers to support direct left recursion without the need for left recursion elimination transformations. This includes nested direct left recursion, such as

```
term = term "+" fact
     | term "-" fact
     | fact
fact = fact "*" num
     | fact "/" num
     | num
```

In the remainder of this section, I will present additional modifications that will enable the parser to also support indirect left recursion.

3.3.3 Getting Ready For Indirect Left Recursion

Recall the following grammar, taken from the introduction,

```
x    = expr
expr = x "-" num | num
```

and consider what happens when x is applied to the input “4-3” using the new version of Apply-Rule given in the previous section. First, the x rule is detected to be left-recursive with a seed parse of x→expr→num→4. Grow-LR then evaluates x’s body once again to try to grow the seed. At this point, the memo table already has an answer for expr, namely expr→num→4; this causes the second evaluation of x to yield a parse identical to the seed parse. Because the last evaluation of x consumed no more input than the seed parse, the loop in Grow-LR terminates and the seed parse becomes the final result. This is clearly not the behavior we wanted.

The example above shows that the modifications for supporting left recursion described in the previous section are overly simplistic. Grow-LR repeatedly evaluates *a single rule* in order to grow the seed parse, which is not sufficient when more than one rule is involved in a left recursion.

I shall now introduce a few concepts that will play a key role in the next and final set of modifications to the parser. The first of these concepts is that of a **rule invocation stack**. Before a rule is evaluated, it is pushed onto the parser’s rule invocation stack, only to be popped off the stack once it has finished computing a result. In Figure 3.5, (A) depicts the rule invocation stack just after the x rule invokes expr.

An invocation of rule *R* is left-recursive if *R* is already on the rule invocation stack, and the parser’s position has not changed since that first invocation. In the example above, the invocation of x by expr, shown in (B), is left-recursive. Left-recursive invocations form a loop in the rule invocation stack. I refer to the rule that started that loop as the **head rule** of the left recursion, and to the other rules in the loop as being **involved** in that left recursion. In (C), x’s thicker border denotes that it is the head of a left recursion, and {expr}, inside the x node, represents the set of rules involved in

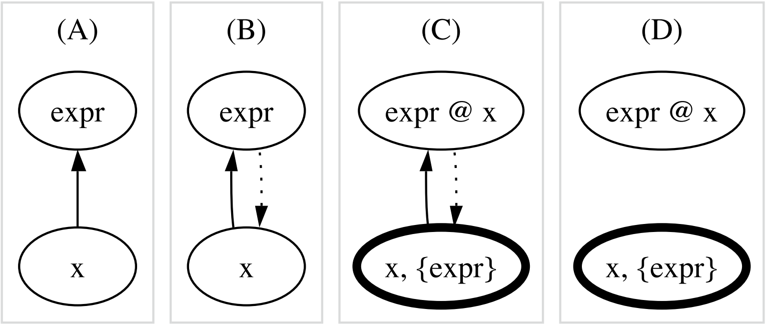


Figure 3.5: The rule invocation stack, shown at various stages during a left-recursive application

that left recursion. The `expr` node, now labeled "`expr @ x`", indicates that `expr` is involved in a left recursion whose head rule is `x`.

We can use this information to handle the invocation of `expr` specially while growing `x`'s seed parse. More specifically, we can force `expr`'s body to be re-evaluated, ignoring the rule's previously memoized result. In general, when growing a left recursion result, we should bypass the memo table and re-evaluate the body of any rule involved in the left recursion. This is the intuition for the final set of modifications, which are presented in the next section.

3.3.4 Adding Support for Indirect Left Recursion

The final version of the APPLY-RULE procedure is shown in Figure 3.6. It has been modified in order to maintain a rule invocation stack as described above. The stack is represented by the LR datatype, which I have modified as follows:

LR : $(seed : \text{AST}, rule : \text{RULE}, head : \text{HEAD}, next : \text{LR})$

The rule invocation stack is kept in the global variable *LRStack*, of type LR, and is represented as a linked list, using LR's *next* field.

LR's *seed* field holds the initial parse found for the associated rule, which is stored in the *rule* field. In place of LR's old *detected* field, we now have the *head* field which, for a left-recursive invocation, holds information pertinent to the left recursion (*head* is set to NIL for non-left-recursive invocations). The HEAD datatype,

$\textsc{Head} : (\mathit{rule} : \textsc{Rule}, \mathit{involvedSet}, \mathit{evalSet} : \textsc{Set}\ \text{of}\ \textsc{Rule})$

contains the head rule of the left recursion (*rule*), and the following two sets of rules:

- *involvedSet*, for the rules involved in the left recursion, and
- *evalSet*, which holds the subset of the involved rules that may still be evaluated during the current growth cycle.

These data structures are used to represent the information depicted in Figure 3.5.

The parser must be able to determine whether left recursion growth is in progress, and if so, which head rule is being grown. Because only one left recursion can be grown at a time for any given position, a global variable, HEADS, is used to map a position to the HEAD of the left recursion which is currently being grown:

$\textsc{Heads} : \textsc{Position} \rightarrow \textsc{Head}$

HEADS is NIL at any position where left recursion growth is not underway.

The task of examining the rule invocation stack to find the head rule and its involved rule set is performed by the SETUP-LR procedure, shown in Figure 3.7.[3] In our example, SETUP-LR is invoked when the stack is at stage (B), in Figure 5, and leaves the stack as shown in stage (C).

GROW-LR, shown in Figure 3.3, must be modified to use the head rule and its involved rule set. Left recursion growth starts by changing Line A to

[3] There are more efficient ways to implement SETUP-LR which avoid walking the stack; I chose to include this one because it is easier to understand.

```
HEADS($P$) ← $H$
```

which indicates that left recursion growth is in progress. For each cycle of growth, the involved rules are given a fresh opportunity for evaluation. This is implemented by changing Line B to

```
$H.evalSet$ ← COPY($H.involvedSet$)
```

In our example, this will cause `expr` to be included in the set of rules which will be re-evaluated if reached. Finally, when left recursion growth is completed, the head at the left recursion position must be removed. To accomplish this, Line C is changed to

```
HEADS($P$) ← NIL
```

When a rule is applied, APPLY-RULE now invokes the RECALL procedure, shown in Figure 3.9, in order to retrieve previous parse results. In addition to fetching memoized results from the memo table, RECALL ensures that involved rules are evaluated during the growth phase. RECALL also prevents rules that were not previously evaluated as part of the left recursion seed construction from being parsed during the growth phase. This preserves the behavior that is expected of a Parsing Expression Grammar (PEG), namely that the first successful parse becomes the result of a rule.

Note that APPLY-RULE now invokes LR-ANSWER (see Figure 3.8), not GROW-LR, when it detects left recursion. If the current rule is the head of the left recursion, LR-ANSWER invokes GROW-LR just as APPLY-RULE did before. Otherwise, the current rule is involved in the left recursion and must defer to the head rule to grow any left-recursive parse, and pass its current parse to participate in the construction of a seed parse.

With these modifications, the parser supports both direct and indirect left recursion.

```
Apply-Rule($R, P$)
    let $m$ = Recall($R, P$)                                          *
    if $m$ = nil
       then ▷ Create a new LR and push it onto the rule
            ▷ invocation stack.
            let $lr$ = new LR(fail, $R$, nil, $LRStack$)              *
            $LRStack \leftarrow lr$                                   *
            ▷ Memoize $lr$, then evaluate $R$.
            $m \leftarrow$ new MemoEntry($lr, P$)
            Memo($R, P$) $\leftarrow m$
            let $ans$ = Eval($R.body$)
            ▷ Pop $lr$ off the rule invocation stack.
            $LRStack \leftarrow LRStack.next$                         *
            $m.pos \leftarrow Pos$
            if $lr.head \neq$ nil                                     *
               then $lr.seed \leftarrow ans$                          *
                    return LR-Answer($R, P, m$)                       *
               else $m.ans \leftarrow ans$                            *
                    return $ans$
       else $Pos \leftarrow m.pos$
            if $m.ans$ is LR
               then Setup-LR($R, m.ans$)                              *
                    return $m.ans.seed$                               *
               else return $m.ans$
```

Figure 3.6: The final version of Apply-Rule (lines marked with * are either new or have changed since the previous version)

```
SETUP-LR$(R, L)$
    if $L.head = \text{NIL}$
        then $L.head \leftarrow$ new HEAD$(R, \{\}, \{\})$
    let $s = LRStack$
    while $s.head \neq L.head$
        do $s.head \leftarrow L.head$
            $L.head.involvedSet \leftarrow L.head.involvedSet \cup \{s.rule\}$
            $s \leftarrow s.next$
```

Figure 3.7: The SETUP-LR procedure

```
LR-ANSWER$(R, P, M)$
    let $h = M.ans.head$
    if $h.rule \neq R$
        then return $M.ans.seed$
        else $M.ans \leftarrow M.ans.seed$
            if $M.ans = \text{FAIL}$
                then return FAIL
                else return GROW-LR$(R, P, M, h)$
```

Figure 3.8: The LR-ANSWER procedure

```
RECALL($R, P$)
    let $m$ = MEMO($R, P$)
    let $h$ = HEADS($P$)
    ▷ If not growing a seed parse, just return what is stored
    ▷ in the memo table.
    if $h$ = NIL
        then return $m$
    ▷ Do not evaluate any rule that is not involved in this
    ▷ left recursion.
    if $m$ = NIL and $R \notin \{h.head\} \cup h.involvedSet$
        then return new MEMOENTRY(FAIL, $P$)
    ▷ Allow involved rules to be evaluated, but only once,
    ▷ during a seed-growing iteration.
    if $R \in h.evalSet$
        then $h.evalSet \leftarrow h.evalSet \setminus \{R\}$
              let $ans$ = EVAL($R.body$)
              $m.ans \leftarrow ans$
              $m.pos \leftarrow Pos$
    return $m$
```

Figure 3.9: The RECALL procedure

3.4 Case Study: Parsing Java's *Primary* Expressions

To validate this mechanism for supporting left recursion, I modified the back-end of *OMeta*, the parsing and pattern matching language presented in Chapter 2, to use the new Apply-Rule procedure described in the previous section. My colleague Jamie Douglass also modified the implementation of his *Context-Free Attributed Transformations* (*CAT*) parser generator accordingly.

I also constructed a grammar that closely mirrors Java's *Primary* rule, as found in chapter 15 of the Java Language Specification [Gos05]. Because of its heavily mutually left-recursive nature, *Primary* cannot be supported directly by conventional packrat parsers [Red08].

My process for composing this grammar, shown in Figure 3.10, started with a careful examination of the grammar of Java expressions. I then identified all other rules that are mutually left-recursive with *Primary*. All such rules, namely

- *Primary*,
- *PrimaryNoNewArray*,
- *ClassInstanceCreationExpression*,
- *MethodInvocation*,
- *FieldAccess*, and
- *ArrayAccess*

are included in my grammar, as are "stubs" for the other rules they reference (*ClassName*, *InterfaceTypeName*, *Identifier*, *MethodName*, *ExpressionName*, and *Expression*).

Input String	*Parse Tree (in s-expression form)*
"`this`"	`this`
"`this.x`"	`(field-access this x)`
"`this.x.y`"	`(field-access (field-access this x) y)`
"`this.x.m()`"	`(method-invocation (field-access this x) m)`
"`x[i][j].y`"	`(field-access (array-access (array-access x i) j) y)`

Table 3.1: Some Java *Primary* expressions and their corresponding parse trees, as generated by a packrat parser modified as proposed in Section 3.3 (the head of an s-expression denotes the type of the AST node)

Next, I removed some of the uninteresting (i.e., non-left-recursive) choices from these rules, and ordered the remaining choices so that the rules would behave correctly when used in a packrat parser. For example, since the method invocation expression "`this.m()`" has a prefix of "`this.m`", which is also a valid field access expression, the *PrimaryNoNewArray* rule must try *MethodInvocation* before trying *FieldAccess*.

I then encoded the resulting grammar in the syntax accepted by the *Pappy* packrat parser generator [For02b]. Just as I expected, *Pappy* was unable to compile this grammar and displayed the error message "`Illegal left recursion: Primary -> PrimaryNoNewArray -> ClassInstanceCreationExpression -> Primary`".

Lastly, I encoded the same grammar in the syntax accepted by *CAT* and *OMeta*. The resulting parsers exhibit the correct behavior, as shown in Table 3.1.

3.5 Performance

A packrat parser's guarantee of linear parse times is based on its ability to compute the result of any single rule application in constant time. My iterative process for growing the seed parse of a left-recursive application violates this assumption, thus making it possible for some left-recursive grammars to yield super-linear parse times. As an example, the grammar

```
Primary                           =  <PrimaryNoNewArray>
PrimaryNoNewArray                 =  <ClassInstanceCreationExpression>
                                   | <MethodInvocation>
                                   | <FieldAccess>
                                   | <ArrayAccess>
                                   | this
ClassInstanceCreationExpression   =  new <ClassOrInterfaceType> ( )
                                   | <Primary> . new <Identifier> ( )
MethodInvocation                  =  <Primary> . <Identifier> ( )
                                   | <MethodName> ( )
FieldAccess                       =  <Primary> . <Identifier>
                                   | super . <Identifier>
ArrayAccess                       =  <Primary> [ <Expression> ]
                                   | <ExpressionName> [ <Expression> ]
ClassOrInterfaceType              =  <ClassName> | <InterfaceTypeName>
ClassName                         =  C | D
InterfaceTypeName                 =  I | J
Identifier                        =  x | y | <ClassOrInterfaceType>
MethodName                        =  m | n
ExpressionName                    =  <Identifier>
Expression                        =  i | j
```

Figure 3.10: Java’s *Primary* expressions

```
start = ones "2" | "1" start | ϵ
ones  = ones "1" | "1"
```

accepts strings of zero or more “1”s in $O(n^2)$ time. The same inefficiency will arise for any grammar that causes the parser to backtrack to the middle of a previously computed left-recursive parse and then re-apply the same left-recursive rule. Fortunately, this problem—which is analogous to that of Ford’s iterative combinators—does not manifest itself in practical grammars [For02b].

In order to gauge the expected performance of packrat parsers modified as described in this chapter, I constructed the following two rules:

```
rr = "1" rr | "1"
lr = lr "1" | "1"
```

The first rule, `rr`, is right-recursive, and the second, `lr`, is left-recursive. Both recognize the same language, i.e., a string of one or more “1”s, and while the parse trees generated by these rules have different associativities, they have the same size.

I used these rules to recognize strings with lengths ranging from 1,000 to 10,000, in increments of 1,000. The results of this experiment are shown in Figure 3.11. The `rr` rule was first timed using a “vanilla” packrat parser implementation (*RR-ORIG*), and then using a version of the same implementation that was modified by one of my colleagues as described in Section 3.3 (*RR-MOD*). The `lr` rule was only timed using the modified implementation (*LR-MOD*), since left recursion is not supported in our “vanilla” implementation.

The following conclusions can be drawn from this experiment:

- **My modifications to support left recursion do not introduce significant overhead for non-left-recursive rules.** Although the recognizing times for *RR-MOD* were consistently slower than those for *RR-ORIG*, the difference was

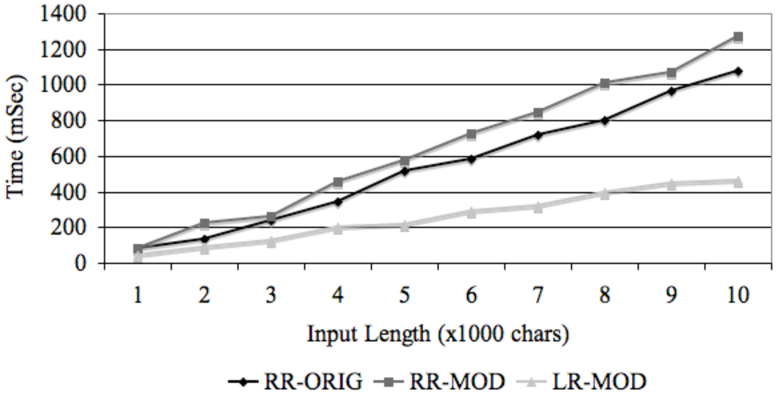


Figure 3.11: *RR-ORIG* shows the performance characteristics of `rr` in a traditional packrat parser implementation; *RR-MOD* and *LR-MOD* show the performance characteristics of `rr` and `lr`, respectively, in an implementation that was modified as described in Section 3.3

rather small. Furthermore, *RR-MOD* and *RR-ORIG* appear to have the same slope.

- **The modified packrat parser implementation supports typical uses of left recursion in linear time**, as shown by *LR-MOD*.

- **Using left recursion can actually improve parse times.** The results of *LR-MOD* were consistently better than those of *RR-MOD* and *RR-ORIG*. More importantly, *LR-MOD*'s gentler slope tells us that it will scale much better than the others on larger input strings. This difference in performance is likely due to the fact that left recursion uses only a constant amount of stack space, while right recursion causes the stack to grow linearly with the size of the input.

In order to measure the effect of indirect left recursion on parse times, I constructed three more versions of the `lr` rule. The first,

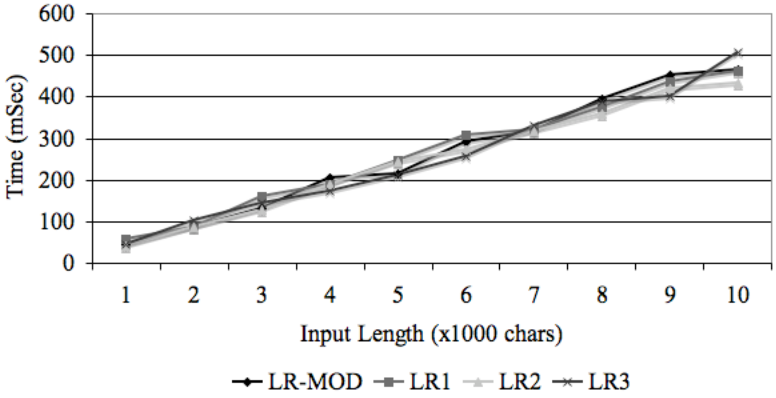


Figure 3.12: The effect of indirect left recursion on parse times

```
lr1 = x "1" | "1"
x   = lr1
```

is indirectly left-recursive "one rule deep" (`lr`, which is directly left-recursive, may be considered to be indirectly left-recursive zero rules deep). The two other rules, `lr2` and `lr3` (not shown), are indirectly left-recursive two and three rules deep, respectively.

Figure 3.12 shows the timing results for the new rules, in addition to the original `lr` rule. These results indicate that adding simple indirection to left-recursive rules does not have a significant effect on parse times.

3.6 Related Work

As discussed in Section 3.1, a number of packrat parser implementations, including *Pappy* [For02b] and *Rats!* [Gri06], support *directly left-recursive* rules by transforming them into equivalent non-left-recursive rules. Unfortunately, because neither of these implementations supports *mutually left-recursive* rules, implementing parsers for grammars containing such rules (such as Java's) using these systems is not trivial.

The programmer must carefully analyze the grammar and rewrite certain rules, which can be tricky. And because the resulting parser is no longer obviously equivalent to the grammar for which it was written, it is difficult to be certain that it does indeed parse the language for which it was intended.

While it may be possible to adapt the "transformational approach" of *Pappy* and *Rats!* to support mutual left recursion by performing a global analysis on the grammar,

- the presence of syntactic and semantic predicate terms may complicate this task significantly, and
- such a global analysis does not mix well with modular parsing frameworks such as *Rats!*, in which rules may be overridden in "sub-parsers".

The approach described here works seamlessly with syntactic and semantic predicates, as well as modular parsing, which makes it a good fit for OMeta.

Frost and Hafiz have proposed a technique for supporting left recursion in top-down parsers that involves limiting the depth of the (otherwise) infinite recursion that arises from left-recursive rules to the length of the remaining input plus 1 [FH06]. While this technique is applicable to any kind of top-down parser (including packrat parsers), it cannot be used when the length of the input stream is unknown, as is the case with interactive input (e.g., read-eval-print loops and network sockets). The approach presented here does not have this limitation and is significantly more efficient, although its need to interact with the memo table makes it applicable only to packrat parsers.

Johnson has proposed a technique based on memoization and Continuation-Passing Style (CPS) for implementing top-down parsers that support left recursion and polynomial parse times [Joh95]. This technique was developed for CFGs and relies heavily on the non-determinism of the CFG choice operator; for this reason, I believe that it would be difficult (if at all possible) to adapt it for use in packrat parser implementations, where the ordering of the choices is significant.

Jamie Douglass, who—along with Todd Millstein—collaborated with me on this project, had previously developed a memoization-based technique for supporting left recursion in an earlier version of *CAT* which only supported CFG rules. That technique's memoization mechanism was restricted to only the head and involved rules, and used only while growing a seed parse.

3.7 Conclusions and Future Work

I have described a modification to the memoization mechanism used by packrat parser implementations that enables them to support both direct and indirect (or mutual) left recursion. This modification obviates the need for left recursion elimination transformations, and supports typical uses of left recursion without sacrificing linear parse times.

Applying this modification to the packrat parser implementations of *OMeta* and *CAT* enabled both of these systems to support the heavily left-recursive portion of the Java grammar discussed in Section 3.4.

One of the anonymous reviewers of PEPM 2008 (the workshop in which this work was published) noted that the compelling simplicity of packrat parsing is "to a large extent lost in the effort to support indirect left recursion." I, like this reviewer, believe that the final version of the algorithm presented in Section 3.3 can be simplified, and hope to do so in future work.

Packrat parsing was originally developed by Bryan Ford to support PEGs [For04]. By extending packrat parsers with support for left recursion, we have also extended the class of grammars they are able to parse (which is now a superset of PEGs). Therefore, it may be interesting to develop a formalism for this new class of grammars that can serve as the theoretical foundation for this new style of packrat parsing.

CHAPTER 4

Worlds: Controlling the Scope of Side Effects

The state of an imperative program—e.g., the values stored in global and local variables, objects' instance variables, and arrays—changes as its statements are executed. These changes, or side effects, are visible globally: when one part of the program modifies an object, every other part that holds a reference to the same object (either directly or indirectly) is also affected. This chapter introduces *worlds*, a language construct that reifies the notion of program state, and enables programmers to control the scope of side effects. I investigate this idea as an extension of JavaScript, and provide examples that illustrate some of the interesting idioms that it makes possible.

4.1 Introduction

Suppose that, while browsing the web, you get to a page that has multiple links and it is not clear which one (if any) will lead to the information you're looking for. Maybe the desired information is just one or two clicks away, in which case it makes sense to click on a link, and if you don't find what you're looking for, click the back button and try the next link. If the information is more than a few clicks away, it might be better to open the link in a new *tab* in which you can explore it to arbitrary depths. That way, if you eventually decide that wasn't the way to go, you can close the tab, and easily try a different path. Another option is to open each link in its own tab, and explore all of them "concurrently".

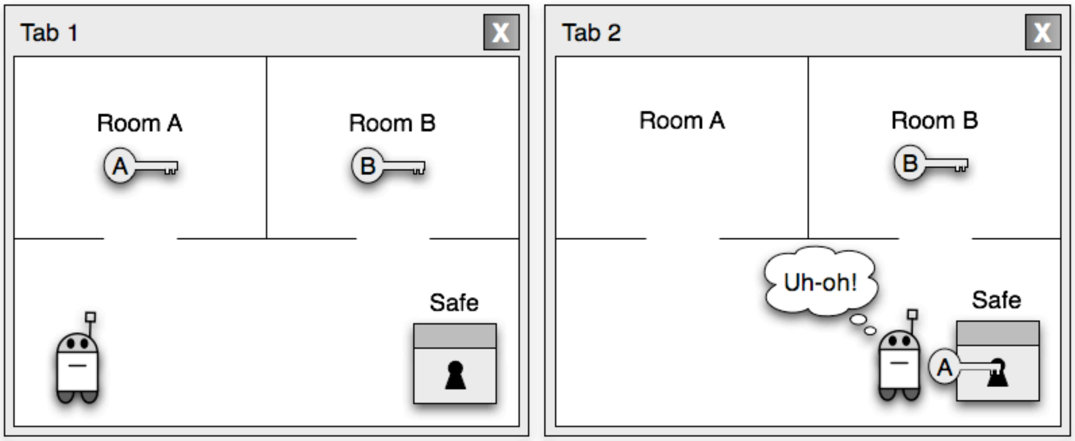
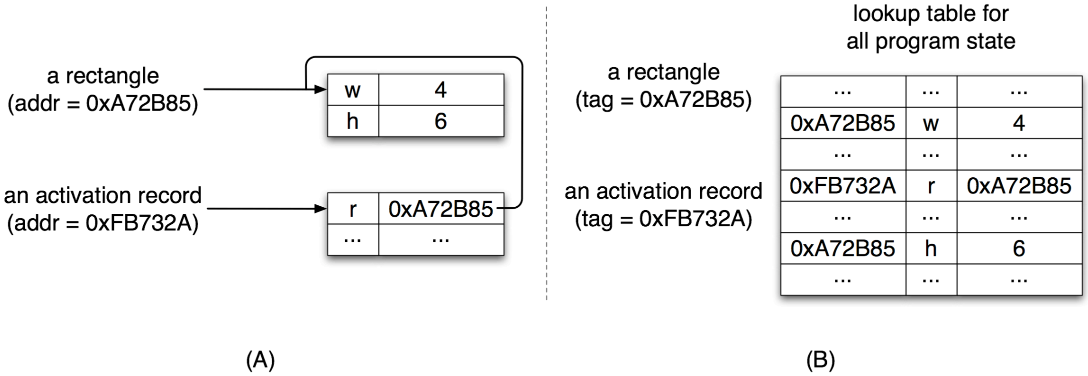


Figure 4.1: “Tabs” 1 and 2 show the state of the world initially, and when the robot discovers that key A does not unlock the safe, respectively.

Something like the tabs of a web browser would be even more useful in a programming language, where undoing actions is a lot trickier than clicking a back button. As an example, consider the task of programming a robot to open a locked safe, as shown in Figure 4.1. There are two keys, A and B (each in its own room), but only one of them unlocks the safe. Using a conventional programming language, we might tell the robot to grab key A from room A, then go to the safe and try to unlock it. At this point, if we find that key A does not open the safe, we probably want to have the robot clean up after himself before trying the next alternative (nobody likes a messy robot). So we must tell the robot to take key A back to room A, and then return to its initial position.

In a programming language that supports “tabs”, these clean-up actions would not be required: we could simply open a new tab, and inside it try to open the safe with key A. If A turns out to be the wrong key, we can simply close this new tab to return to the initial conditions.

This chapter explores the idea of “tabs for programming languages”, which I call *worlds*.



...	...	...
0xA72B85	w	4
...	...	...
0xFB732A	r	0xA72B85
...	...	...
0xA72B85	h	6
...	...	...

Figure 4.2: Two ways to represent program state. In (A), an object is uniquely identified by the address of the block of memory in which its state is stored. In (B), objects are just tags and their state is stored externally in a single lookup table.

4.2 Approach

The state of a program is scattered around the computer's memory in several kinds of data structures: arrays, objects, activation records, etc. We normally think of these data structures as little "bundles of state", and they are often implemented as such (see Figure 4.2 (A)). Alternatively, we can think of program state *itself* as a data structure—a kind of lookup table or associative array—that is used to represent all other data structures in the system. In this model, each object or data structure is uniquely identified by a *tag*, and the program state maps (tag, property name) pairs to their values (see Figure 4.2 (B)).

Reifying the notion of program state raises some interesting questions:

- Is it useful for the program state to be a first-class value/object?
- Does it make sense for multiple "program states" to co-exist in the same program?
- If so, should a program state be able to inherit from (or delegate to) another program state?

I answer all of these questions with a resounding “yes”.

4.2.1 Worlds

The *world* is a new language construct that reifies the notion of program state. All computation takes place inside a world, which captures all of the side effects—changes to global, local, and instance variables, arrays, etc.—that happen inside it.

A new world can be “sprouted” from an existing world at will. The state of a *child world* is derived from the state of its parent, but the side effects that happen inside the child do not affect the parent. (This is analogous to the semantics of delegation in prototype-based languages with copy-on-write slots.) At any time, the side effects captured in the child world can be propagated to its parent via a *commit* operation.

4.2.2 Worlds/JS

A programming language that supports worlds must provide some way for programmers to:

- refer to the current world,
- sprout a new world from an existing world,
- commit a world’s changes to its parent world, and
- execute code in a particular world.

I now describe the particular way in which these operations are supported in Worlds/JS, an extension of JavaScript [ECM99] I have prototyped in order to experiment with the ideas discussed in this chapter.[1]

[1] My prototype implementation of Worlds/JS is available at `http://www.tinlizzie.org/ometa-js/#Worlds_Paper`. No installation is necessary; you can experiment with the language di-

Worlds/JS extends JavaScript with the following additional syntax:

- `thisWorld` is an expression that evaluates to the current world, and
- `in` *expr block* is a statement that executes *block* inside the world obtained by evaluating *expr*.

Worlds are first-class values: they can be stored in variables, passed as arguments to functions, etc. They can even be garbage-collected just like any other object. All worlds delegate to the world prototype, whose `sprout` and `commit` methods can be used to create a new world that is a child of the receiver, and propagate the side effects captured in the receiver to its parent, respectively.

In the following example, we modify the height of the same instance of `Rectangle` in two different ways, each in its own world, and then commit one of them to the original world. This serves the dual purpose of illustrating the syntax of Worlds/JS as well as the semantics of `sprout` and `commit`.

```
A = thisWorld;
r = new Rectangle(4, 6);

B = A.sprout();
in B { r.h = 3; }

C = A.sprout();
in C { r.h = 7; }

C.commit();
```

Figures 4.3 and 4.4 show the state of all worlds involved before and after the commit operation, respectively.

rectly in your web browser.

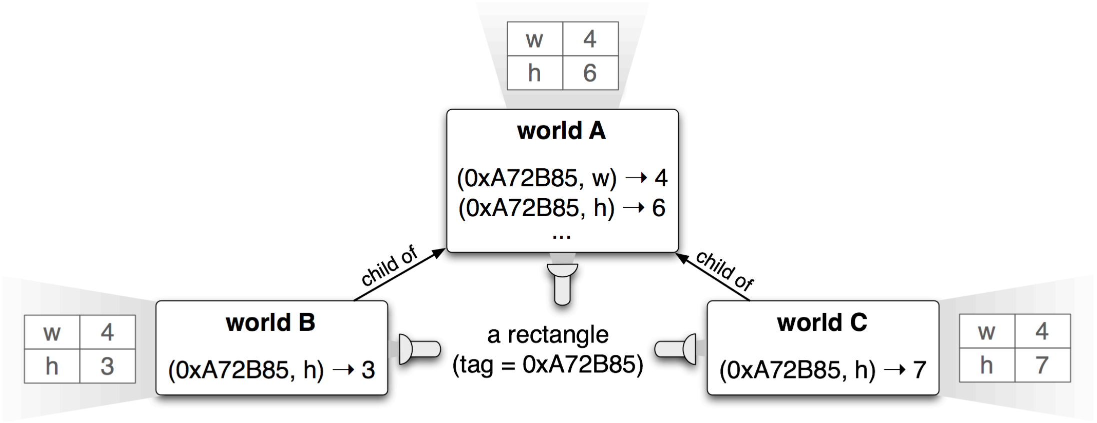


Figure 4.3: Projections/views of the same object in three different worlds

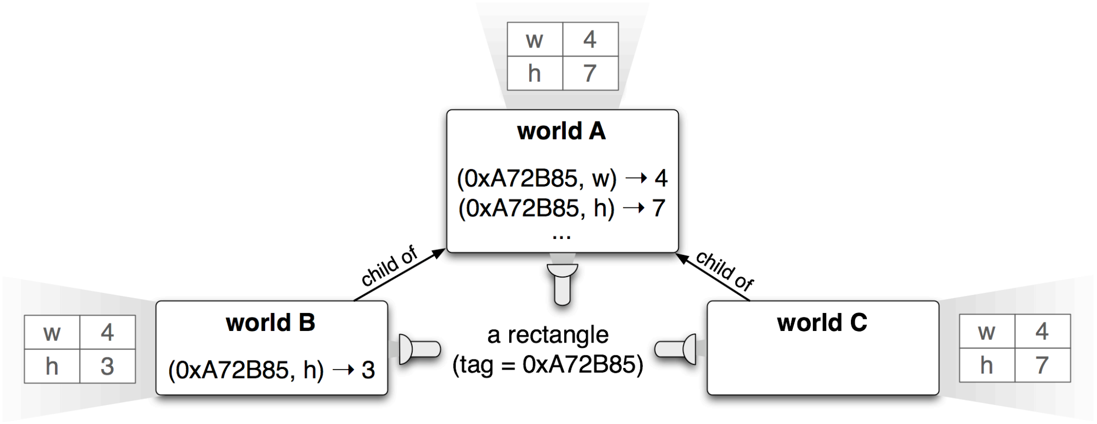


Figure 4.4: The state of the “universe” shown in Figure 4.3 after a *commit* on world C

4.3 Property (or Field) Lookup in Worlds/JS

This section formally describes the semantics of property (or field) lookup in Worlds/JS, which is a natural generalization of property lookup in JavaScript.

4.3.1 Property Lookup in JavaScript

JavaScript's object model is based on single delegation, which means that every object inherits (and may override) the properties of another object. The only exception to this rule is `Object.prototype` (the ancestor of all objects), which is the root of JavaScript's delegation hierarchy and therefore does not delegate to any other object.

The semantics of property lookup in JavaScript can be formalized using the following two primitive operations:

(i) $getOwnProperty(x, p)$, which looks up property p in object x *without looking up the delegation chain*. More specifically, the value of $getOwnProperty(x, p)$ is

- v, if x has property p *that is not inherited from another object*, and whose value is v, and
- the special value **none**, otherwise;

(ii) $parent(x)$, which evaluates to

- y, the object to which x delegates, or
- the special value **none**, if x does not delegate to any other object.

and the following set of inference rules:

$$\frac{\begin{array}{c} getOwnProperty(x, p) = v \\ v \neq \textbf{none} \end{array}}{lookup(x, p) = v} \qquad \text{(JS-Lookup-Own)}$$

$$\frac{\begin{array}{c} getOwnProperty(x,p) = \textbf{none} \\ parent(x) = \textbf{none} \end{array}}{lookup(x,p) = \textbf{none}} \qquad \text{(JS-Lookup-Root)}$$

$$\frac{\begin{array}{c} getOwnProperty(x,p) = \textbf{none} \\ parent(x) = y \\ y \neq \textbf{none} \\ lookup(y,p) = v \end{array}}{lookup(x,p) = v} \qquad \text{(JS-Lookup-Child)}$$

4.3.2 Property Lookup in Worlds/JS

In Worlds/JS, property lookup is always done in the context of a world. And since it may be that an object x has a property p in some world w but not in another, the primitive operation $getOwnProperty(x, p)$ must be replaced by a new primitive operation, $getOwnPropertyInWorld(x, p, w)$.

Another primitive operation we will need in order to formalize the semantics of property lookup in Worlds/JS is $parentWorld(w)$, which yields w's parent, or the special value **none**, if w is the top-level world.

Using these two new primitive operations, we can define a new operation, $getOwnProperty(x, p, w)$, which yields the value of x's p property in world w, or (if $x.p$ is not defined in w) in w's closest ancestor:

$$\frac{\begin{array}{c} getOwnPropertyInWorld(x,p,w) = v \\ v \neq \textbf{none} \end{array}}{getOwnProperty(x,p,w) = v} \quad \text{(WJS-GetOwnProperty-Own)}$$

$$\frac{\begin{array}{c} getOwnPropertyInWorld(x, p, w) = \textbf{none} \\ parentWorld(w) = \textbf{none} \end{array}}{getOwnProperty(x, p, w) = \textbf{none}} \text{(WJS-GetOwnProperty-Root)}$$

$$\frac{\begin{array}{c} getOwnPropertyInWorld(x, p, w_1) = \textbf{none} \\ parentWorld(w_1) = w_2 \\ w_2 \neq \textbf{none} \\ getOwnProperty(x, p, w_2) = v \end{array}}{getOwnProperty(x, p, w_1) = v} \text{(WJS-GetOwnProperty-Child)}$$

And finally, using the worlds-friendly variant of *getOwnProperty* defined above, the inference rules that formalize the semantics of lookup in Worlds/JS can be written as follows:

$$\frac{\begin{array}{c} getOwnProperty(x, p, w) = v \\ v \neq \textbf{none} \end{array}}{lookup(x, p, w) = v} \qquad \text{(WJS-Lookup-Own)}$$

$$\frac{\begin{array}{c} getOwnProperty(x, p, w) = \textbf{none} \\ parent(x) = \textbf{none} \end{array}}{lookup(x, p, w) = \textbf{none}} \qquad \text{(WJS-Lookup-Root)}$$

$$\frac{\begin{array}{c} getOwnProperty(x, p, w) = \textbf{none} \\ parent(x) = y \\ y \neq \textbf{none} \\ lookup(y, p, w) = v \end{array}}{lookup(x, p, w) = v} \qquad \text{(WJS-Lookup-Child)}$$

Note that these rules closely mirror those that describe the semantics of lookup in

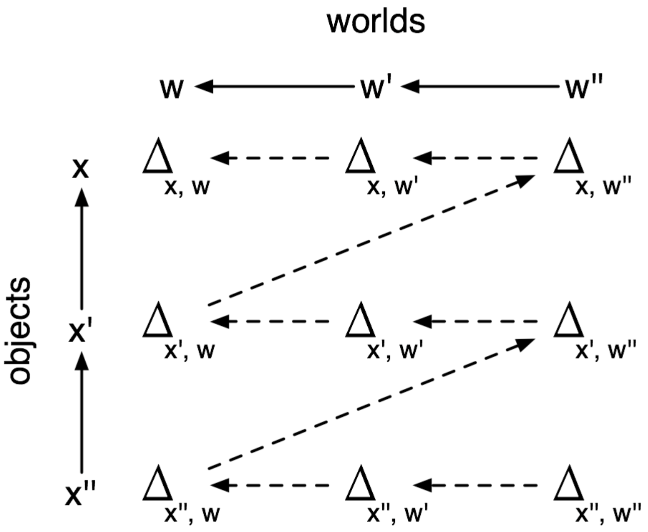


Figure 4.5: The property lookup order used when evaluating $x''.p$ in world w'' (the notation $\Delta_{x,w}$ represents the properties of x that were modified in w)

JavaScript—the only difference is that *getOwnProperty* and *lookup* now both take a world as an additional argument.

Figure 4.5 illustrates the property lookup order that results from the algorithm described above. The solid vertical lines in the diagram indicate *delegates-to* relationships (e.g., object x' delegates to x), whereas the solid horizontal lines indicate *is-child-of* relationships (e.g., world w' is a child of w). Note that the chain of worlds gets precedence over the object delegation chain; in other words, any relevant "version" of an object may override the properties of the object to which it delegates. This lookup order preserves JavaScript's copy-on-write delegation semantics, i.e., if a delegates to b, and then we assign into a's p property, subsequent changes to b's p property will not affect a. So no matter what world a statement is executed in—whether it is the top-level world, or a world that sprouted from another world—it will behave in exactly the same way as it would in "vanilla" JavaScript.

In programming languages that do not support delegation (e.g., Java), the semantics of property lookup in the presence of worlds can be considered as a special case of the semantics described in this section in which all prototype chains have length 1, i.e., no object delegates to any other object.

4.4 Examples

The following examples illustrate some of the applications of worlds. Other obvious applications (not discussed here) include heuristic search and sand-boxing.

4.4.1 Better Support for Exceptions

In languages that support exception-handling mechanisms (e.g., the `try/catch` statement), a piece of code is said to be *exception-safe* if it guarantees not to leave the program in an inconsistent state when an exception is thrown. Writing exception-safe code is a tall order, as I illustrate with the following example:

```
try {
  for (var idx = 0; idx < xs.length; idx++)
    xs[idx].update();
} catch (e) {
  // ...
}
```

Our intent is to update every element of `xs`, an array. The problem is that if one of the calls to `update` throws an exception, some (but not all) of `xs`’ elements will have been updated. So in the `catch` block, the program should restore `xs` to its previous consistent state, in which none of its elements was updated.

One way to do this might be to make a copy of every element of the array before entering the loop, and in the `catch` block, restore the successfully-updated elements to their previous state. In general, however, this is not sufficient since `update` may also have modified global variables and other objects on the heap. Writing truly exception-safe code is difficult and error-prone.

Versioning exceptions [NJ06] offer a solution to this problem by giving try/catch statements a transaction-like semantics: if an exception is thrown, all of the side effects resulting from the incomplete execution of the try block are automatically rolled back before the catch block is executed . In a programming language that supports worlds and a traditional (non-versioning) try/catch statement, the semantics of versioning exceptions can be implemented as a design pattern. I illustrate this pattern with a rewrite of the previous example:

```
w = thisWorld.sprout();
try {
  in w {
    for (var idx = 0; idx < xs.length; idx++)
      xs[idx].update();
  }
} catch (e) {
  w = null;
  // ...
} finally {
  if (w != null)
    w.commit();
}
```

First, we create a new world w in which to capture the side effects of the try block. If an exception is thrown, we simply discard w. Otherwise—if the try block completes successfully—we propagate (commit) the side effects to the "real world". This is done inside a finally block to ensure that the side effects will be propagated even if the try block returns.

4.4.2 Undo for Applications

We can think of the "automatic clean-up" supported by versioning exceptions as a kind of one-level *undo*. In the last example, we implemented this by capturing the side effects of the `try` block—the operation we may need to undo—in a new world. The same idea can be used as the basis of a framework that makes it easy for programmers to implement applications that support multi-level undo.

Applications built using this framework are objects that support two operations: `perform` and `undo`. Clients use the `perform` operation to issue commands to the application, and the `undo` operation to restore the application to its previous state (i.e., the state it was in before the last command was performed). The example below illustrates how a client might interact with a counter application that supports the commands `inc`, `dec`, and `getCount`, for incrementing, decrementing, and retrieving the counter's value, respectively. (The counter's value is initially zero.)

```
counter.perform('inc');
counter.perform('inc');
counter.perform('dec');
counter.undo();                     // undo the 'dec' command
print(counter.perform('getCount')); // outputs the no. '2'
```

The interesting thing about our framework is that it allows programmers to implement applications that support multi-level undo *for free*, i.e., without having to do anything special such as using the *command* design pattern [GHJ95]. The implementation of the counter application—or rather, a factory of counters—is shown below:

```
Application = function() { };
Application.prototype = {
  worlds: [thisWorld],
  perform: function(command) {
    var w = this.worlds.last().sprout();
    this.worlds.push(w);
    in w { return this[command](); }
  },
  undo: function() {
    if (this.worlds.length > 0)
      this.worlds.pop();
  },
  flattenHistory: function() {
    while (this.worlds.length > 1) {
      var w = this.worlds.pop();
      w.commit();
    }
  }
};
```

Figure 4.6: A framework for building applications that support multi-level undo

```
makeCounter = function() {
  var app      = new Application();
  var count    = 0;
  app.inc      = function() { count++;      };
  app.dec      = function() { count--;      };
  app.getCount = function() { return count; };
  return app;
};
```

Note that the counter application is an instance of the `Application` class. `Application` *is* our framework; in other words, it is where all of the undo functionality is implemented. Its source code is shown in Figure 4.6.

The state of the application is always accessed in a world that "belongs" to the application. When the application is instantiated, it has only one world. Each time a client issues a command to the application via its `perform` operation, the method that corresponds to that command (the one with the same name as the command) is invoked in a new world. This new world is sprouted from the world that holds the previous version of the application's state (i.e., the one in which the last command was executed). The `undo` operation simply discards the world in which the last command was executed, effectively returning the application to its previous state. Lastly, the (optional) `flattenHistory` operation coalesces the state of the application into a single world, which prevents clients from `undo`ing past the current state of the application.

Note that the application's public interface (the `perform` and `undo` methods) essentially models the way in which web browsers interact with online applications, so this technique could be used in a web application framework like Seaside [DLR07].

4.4.3 Extension Methods in JavaScript

In JavaScript, functions and methods are "declared" by assigning into properties. For example,

```
Number.prototype.fact = function() {
  if (this == 0)
    return 1;
  else
    return this * (this - 1).fact();
};
```

adds the `fact`orial method to the `Number` prototype. Similarly,

```
inc = function(x) { return x + 1 };
```

declares a function called `inc`. (The left-hand side of the assignment above is actually shorthand for `window.inc`, where `window` is bound to JavaScript's *global* object.)

JavaScript does not support modules, which makes it difficult, sometimes even impossible for programmers to control the scope of declarations. But JavaScript's declarations are really side effects, and worlds enable programmers to control the scope of side effects. I believe that worlds could serve as the basis of a powerful module system for JavaScript, and have already begun experimenting with this idea.

Take extension methods, for example. In dynamic languages such as JavaScript, Smalltalk, and Ruby, it is common for programmers to extend existing objects/classes (e.g., the `Number` prototype) with new methods that support the needs of their particular application. This practice is informally known as *monkey-patching* [Bra08].

Monkey-patching is generally frowned upon because, in addition to polluting the interfaces of the objects involved, it makes programs vulnerable to name clashes that are impossible to avoid. Certain module systems, including those of MultiJava [Cli06] and eJava [WSM06], eliminate these problems by allowing programmers to declare *lexically-scoped* extension methods. These must be explicitly imported by the parts of an application that wish to use them, and are invisible to the rest of the application.

The following example shows that worlds can be used to support this form of modularity:

```
ourModule = thisWorld.sprout();
in ourModule {
  Number.prototype.fact = function() { ... };
}
```

The `fact`orial method defined above can only be used inside `ourModule`,

```
in ourModule {
  print((5).fact());
}
```

and therefore does not interfere with other parts of the program.

This idiom can also be used to support *local rebinding*, a feature found in some module systems [BDW03, BDN05, DGL07] that enables programmers to locally replace the definitions of existing methods. As an example, we can change the behavior of Number's toString method only when used inside ourModule:

```
in ourModule {
  numberToEnglish = function(n) { ... };
  Number.prototype.toString = function() {
    return numberToEnglish(this);
  };
}
```

and now the output generated by

```
arr = [1, 2, 3];
print(arr.toString());
in ourModule {
  print(arr.toString());
}
```

is

```
[1, 2, 3]
[one, two, three]
```

4.4.4 Scoping Methods, not Side Effects

Although the previous set of examples suggests that worlds can be used to implement a module system for JavaScript, it elided an important problem. Consider the following "module", which extends the Person prototype (not shown) with a new method:

```
ourOtherModule = thisWorld.sprout();
in ourOtherModule {
  Person.prototype.makeOlder = function() {
    this.age = this.age + 1;
  };
}
```

Note that, unlike the methods in the other examples, makeOlder has side effects—namely, it updates the age property of the receiver. This is problematic because makeOlder can only be invoked from ourOtherModule, and since a world captures all of the side effects that happen inside it, the updated age will not be visible in the top-level world.

Invoking the commit method on ourOtherModule might seem like a good idea, since it would propagate the updated age to the top-level world. Unfortunately, it would also propagate the makeOlder method, which would make the module useless.

A more viable solution to this problem is to add another method to Person—a setter for the age property—that is somehow statically bound to the top-level world. In other words, invocations of setAge should always be evaluated in the top-level world, unlike other methods which are evaluated in whatever world from which they are invoked. We can do this by capturing the top-level world in the method's closure:

```
Person.prototype.setAge = (function() {
  var w = thisWorld;
  return function(age) {
    in w { this.age = age; }
  };
})();
```

And now we can rewrite makeOlder in ourOtherModule to use the setter method:

```
in ourOtherModule {
  Person.prototype.makeOlder = function() {
    this.setAge(this.age + 1);
  };
}
```

While using "world-bound" setter methods like setAge is somewhat clunky, it does enable modules to contain functions with side effects. One problem remains, however: these side-effectful functions do not interact well with worlds. For example, the side effects of an invocation of makeOlder will not be rolled back as expected if it is used with the versioning exceptions idiom presented in Section 4.4.1. This is because the assignments performed by our setter method always take place in the top-level world, rather than in the world from which the module is being used.

A better idiom for dealing with module-local functions that have side effects is shown in Figure 4.7. This new idiom uses a pair of generic getter and setter methods that can be used to access and update the properties of any object, and *are always evaluated in the world from which the module is being used* (the “calling world”). The module’s functionality is no longer accessed directly, using an `in` statement, but rather through the `useForEvaluating` method, which is defined on worlds (modules), i.e.,

```
betterModule.useForEvaluating(function() {
  joe.makeOlder();
});
```

means “execute the statement `joe.makeOlder();` inside `betterModule`”. The `useForEvaluating` method saves the calling world in a global variable (inside the module that is its receiver) so that it can be accessed by the generic getter and setter methods. (This global variable is declared implicitly in the `useForEvaluating` method by the assignment `callingWorld = cw`.) Since the invocation `joe.makeOlder()` takes place inside `betterModule`, where `callingWorld` is set to the world from which `useForEvaluating` was called, `makeOlder`’s calls to `get` and `set` will access/update the receiver (`joe`) in the calling world. As a result, `makeOlder` now interacts nicely with worlds, e.g., `makeOlder`’s side effects will be properly rolled back if it is used in a program that uses the versioning exceptions idiom.

4.5 Case Study: Using Worlds to Improve OMeta

Consider the semantics of OMeta’s ordered choice operator (`|`). If a match fails while its first operand is being evaluated, it causes the parser, or more generally, the *matcher*

```
Object.prototype.get = function(p) {
  in callingWorld { return this[p]; }
};
Object.prototype.set = function(p, v) {
  in callingWorld { this[p] = v; }
};

World.prototype.useForEvaluating = function(f) {
  var cw = thisWorld;
  in this {
    callingWorld = cw;
    return f();
  }
};

betterModule = thisWorld.sprout();
in betterModule {
  Person.prototype.makeOlder = function() {
    this.set("age", this.get("age") + 1);
  };
}
```

Figure 4.7: A better way to deal with side effects in modules

to automatically backtrack to the appropriate position on the input stream before trying the second operand. We can think of this backtracking as a limited kind of undo that is only concerned with changes to the matcher's position on the input stream. Other kinds of side effects that can be performed by semantic actions—e.g., destructive updates such as assigning into one of the fields of the matcher object or a global variable—are not undone automatically, which means that the programmer must be specially careful when writing rules with side effects.

To show that worlds can greatly simplify the management of state in backtracking programming languages like OMeta and Prolog, I have implemented a variant of OMeta/JS in which the choice operator automatically discards the side effects of failed alternatives, and similarly, the repetition operator (`*`) automatically discards the side effects of the last (unsuccessful) iteration. This was surprisingly straightforward: since Worlds/JS is a superset of JavaScript (the language in which OMeta/JS was originally implemented), all I had to do was redefine the methods that implement the semantics of these two operators.

Figures 4.8 (A) and (B) show the original and modified implementations of the ordered choice operator, respectively. Note that the modified implementation sprouts a new world in which to evaluate each alternative, so that the side effects of failed alternatives can easily be discarded. These side effects include the changes to the matcher's input stream position, and therefore the code that implemented backtracking in the original version (`this.input = origInput`) is no longer required. Finally, the side effects of the first successful alternative are committed to the parent world (in the `finally` block).

Similarly, the alternative implementation of the repetition operator (omitted for brevity) sprouts a new world in which to try each iteration, so that the effects of the last (unsuccessful) iteration can be discarded.

```
// (A) Original implementation of the ordered choice operator
OMeta._or = function() {
  var origInput = this.input;
  for (var idx = 0; idx < arguments.length; idx++)
    try {
      this.input = origInput;
      return arguments[idx]();
    }
    catch (f) {
      if (f != fail)
        throw f;
    }
  throw fail;
};

// (B) Modified implementation of the ordered choice operator
OMeta._or = function() {
  for (var idx = 0; idx < arguments.length; idx++) {
    var ok = true;
    in thisWorld.sprout() {
      try { return arguments[idx](); }
      catch (f) {
        ok = false;
        if (f != fail)
          throw f;
      }
      finally {
        if (ok)
          thisWorld.commit();
      }
    }
  }
  throw fail;
};
```

Figure 4.8: Implementations of two different semantics for OMeta's ordered choice operator

4.6 Related Work

The idea of treating the program store as a first-class value and more importantly, enabling programmers to take snapshots of the store which could be restored at a later time, first appeared in Johnson and Duggan's GL programming language [JD88]. This model was later extended by Morrisett to allow the store to be partitioned into a number of disjoint "sub-stores" (each with its own set of variables) that could be saved and restored separately [Mor93].

The main difference between previous formulations of first-class stores and worlds lies in the programming model: whereas first-class stores have until now been presented as a mechanism for manipulating *a single store* through a save-and-restore interface, worlds enable multiple versions of the store—several "parallel universes"—to co-exist in a single program. This makes worlds a better fit for the experimental programming style that is the topic of this dissertation, and also makes it possible for multiple "experiments" to be carried out in parallel, which I intend to investigate in future work.

In languages that support Software Transactional Memory (STM) [ST95, HMP05], every transaction that is being executed at a given time has access to its own view of the program store that can be modified in isolation, without affecting other transactions. Therefore, like worlds, STM enables multiple versions of the store to co-exist. But while the primary motivation of STM is to provide a simple model for writing multithreaded programs, the goal of my work on worlds is to provide language support experimental programming. This is the source of a number of differences between the two approaches. Transactions are meant to be short-lived, are implicitly committed to the "real world" (i.e., the persistent store), and look like atomic operations when viewed from the outside. Worlds, on the other hand, are not tied to any particular control structure. A world may exist indefinitely, if it is never committed to its parent, and can be examined from the outside using the `in` statement (this is useful for implementing heuristic searches). Furthermore, the commit operation for worlds currently does not detect conflicts: when multiple sibling worlds propagate their changes with the commit operation, it is possible that the parent world will be left in an inconsistent state.

Tanter has shown that (implicitly) *contextual values*, i.e., values that vary depending on the context in which they are accessed or modified, can be used to implement a *scoped assignment* construct that enables programmers to control the scope of side effects [Tan08]. Although Tanter's construct does not support the equivalent of the commit operation on worlds, it is more general than worlds in the sense that it allows any value to be used as a context. However, it is not clear whether this additional generality justifies the complexity that it brings to the programming model. For example, while it is straightforward to modify a group of variables in the context of the current thread (e.g., thread id 382), or the current user (e.g., awarth), it is difficult to reason about the state of the program when both contexts are active, since they need not be mutually exclusive. (This is similar to the semantic ambiguities that are caused by multiple inheritance in object-oriented languages.)

Lastly, a number of mechanisms for synchronizing distributed and decentralized systems (e.g., TeaTime [Ree78, Ree05] and Virtual Time / Time Warp [Jef85]) and optimistic methods for concurrency control [KR81] rely on the availability of a rollback (or undo) operation. As shown in Section 4.4.2, a programming language that supports worlds greatly simplifies the implementation of rollbacks, and therefore could be the ideal platform for building these mechanisms.

4.7 Future Work

I believe that worlds have the potential to provide a tractable programming model for multi-core architectures. As part of the STEPS project [KIO06, KPR07], I intend to investigate the feasibility of an efficient, hardware-based implementation of worlds that will enable the kinds of experiments that might validate this claim. For example, there are many problems in computer science for which there are several known algorithms, each with its own set of performance tradeoffs. In general, it is difficult to tell when one algorithm (or optimization) should be used over another. My hardware-based implementation should make it practical for a program to choose among optimizations simply by sprouting multiple "sibling worlds"—one for each algorithm—and running all of them in parallel. The first one to complete its task would be allowed to propagate its results, and the others would be discarded.

When multiple sibling worlds propagate their side effects using the *commit* operation, it is possible that the parent world will be left in an inconsistent state. The notion of serializability, in transaction-processing systems, could be used for detecting "collisions" that arise when multiple sibling worlds commit their changes. This in turn could be used to provide a variant of the *commit* operation that is only carried out when there are no collisions, which would have a number of interesting applications. For example, in a language that supports concurrency, it would enable a transactional memory abstraction to be implemented as a library.

One limitation of worlds is that they only capture the *in-memory* side effects that happen inside them. Programmers must therefore be careful when executing code that includes other kinds of side effects, e.g., sending packets on the network and obtaining input from the user. It would be interesting to investigate whether some of the techniques used in reversible debuggers such as EXDAMS [Bal69] and IGOR [FB89] can be used to ensure that, for example, when two sibling worlds read a character from the console, they get the same result.

The (abstract) lookup table shown in Figure 4.2 (B) is indexed by two keys: object tag and property name. While this is sufficient to support worlds in a conventional programming language like JavaScript, certain advanced features may require more keys. In order to *directly* support module-specific state, for example, we might add a third key to our lookup table that associates each piece of state with a particular module. Similarly, in order to support context-oriented programming (COP) [HCN08], we may want to add a third key to our lookup table that identifies a context. If we want to support both modules and COP, we will need four keys. Therefore it may be interesting to look into supporting an arbitrary number of keys, although this will have to be done carefully in order to avoid over-complicating the programming model.

CHAPTER 5

Conclusions

This dissertation argues that programming languages and constructs designed specifically to support experimentation can substantially simplify the jobs of researchers and programmers alike. I have supported this thesis by targeting two very different kinds of experimentation, namely (i) experimenting with new ideas in the domain of programming language design, and (ii) using experimentation as a programming paradigm.

Of course, a new programming language is not actually helpful unless programmers are compelled to use it, or in the very least, *experiment* with it enough to truly understand the powerful ideas behind it. I end this dissertation with a few pieces of "marketing advice" for language designers who wish to encourage programmers to do this third kind of experimentation, and seamlessly harness the interest generated by their experiments to build a user community. This advice is based on my experience with the *OMeta/JS Workspace Wiki*[1], a framework that enables programmers to experiment with OMeta (or any other language implemented with OMeta, like Worlds/JS) in the convenience of their web browser, without having to install any additional software.

- **Give them a prototype that runs inside the web browser.**

  Conference papers are a good way to introduce a new programming language to a large audience of programmers and researchers. But let's face it: no matter

[1] Available at `http://www.tinlizzie.org/ometa-js/`

how novel, useful, and well-designed your language may be, it is not going to captivate everyone. And only a fraction of those who are interested in your language will be interested enough to go through the hassle of downloading and installing your prototype implementation so that they can experiment with it.

By implementing a prototype that runs inside the web browser—one that does not have to be installed—you make it easy for whomever is interested in your language to try it out, which is likely to translate into a larger user community. We can think of the web browser-based prototype as a kind of "gateway drug": once the programmer is hooked, he will likely graduate to a more conventional (and possibly more robust) implementation.

There are several ways in which a programming language may be prototyped to run inside the web browser. My OMeta/JS implementation, for example, translates OMeta programs to JavaScript code that can be executed directly with the `eval` function. Other alternatives include Java applets, ActionScript / Flash, etc.

- **Eliminate the cumbersome edit-compile-run cycle with a workspace.**

  The traditional edit-compile-run cycle is not conducive to experimentation. For example, it can be frustrating to have to create a test file just to verify an assumption about how the language works. A Smalltalk-style *workspace*—i.e., a text editor that allows code fragments to be evaluated—makes programming a much more interactive (and fun!) activity, and therefore provides a better environment in which to introduce programmers to your language.

- **Give them a place to share their code online.**

  The availability of a medium for users to share interesting examples can be instrumental in building a user community. A great way to do this is to evolve the workspace discussed above into a wiki. In other words, make it possible for users to save the contents of their workspace sessions, and share them with other users. (In fact, I made my Worlds/JS prototype available as a OMeta/JS Workspace Wiki page.) This kind of wiki also makes it easy for users to create off-shoots of other users' examples.

The idea of using the workspace as a user interface for exploratory programming inside the web browser, as well as the idea of turning it into a wiki, originated in my colleague Takashi Yamamiya's *JavaScript Workspace* [Yam].

- **Keep your prototype implementation simple, and make the source code freely available.**

  A prototype implementation that is simple and easy to understand...

  - will be read by programmers who wish to attain a deeper understanding of the semantics of your language.
  - will be modified by those who want to experiment with variations of the syntax and/or semantics of your language. This can be a valuable source of ideas for future work, and you can encourage it by making the source code to your prototype available using the same workspace-wiki discussed previously.
  - will encourage members of the user community to port your language to different platforms. This really works: OMeta has been ported to a number of other languages including C#, Python, Scheme, Lisp, and Factor.

  A good way to reach this goal of simplicity, apart from using OMeta, is to embrace the differences between "prototype" and "industrial-quality implementation". Your prototype does not have to be particularly efficient or robust, it just needs to be good enough to allow programmers to write interesting examples.

References

[Bal69] Robert M. Balzer. "EXDAMS—EXtendable Debugging and Monitoring System." In *AFIPS Spring Joint Computer Conference*, volume 34, pp. 567–580, 1969.

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. "Classbox/J: Controlling the scope of change in Java." In *OOPSLA'05: Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 177–189, New York, NY, USA, 2005. ACM Press.

[BDW03] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. "Classboxes: A minimal module model supporting local rebinding." In *In Proceedings of JMLC 2003 (Joint Modular Languages Conference), volume 2789 of LNCS*, pp. 122–131. Springer-Verlag, 2003.

[BKV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. "Stratego/XT 0.17. A language and toolset for program transformation." *Science of Computer Programming*, **72**(1-2):52–70, 2008.

[Bra08] Gilad Bracha. "Monkey patching (blog post)." `http://gbracha.blogspot.com/2008/03/monkey-patching.html`, 2008.

[BS08] Ralph Becket and Zoltan Somogyi. "DCGs + Memoing = Packrat Parsing, but Is It Worth It?" In Paul Hudak and David Scott Warren, editors, *PADL '08: Proceedings of the 10th International Symposium on the Practical Aspects of Declarative Languages*, volume 4902 of *Lecture Notes in Computer Science*, pp. 182–196. Springer, 2008.

[Cha87] David R. Chase. "An improvement to bottom-up tree pattern matching." In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 168–177, New York, NY, USA, 1987. ACM.

[Cli06] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. "MultiJava: Design Rationale, Compiler Implementation, and Applications." *ACM Transactions on Programming Languages and Systems*, **28**(3), May 2006.

[DGL07] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. "Encapsulating and exploiting change with changeboxes." In *ICDL '07: Proceedings of the 2007 International Conference on Dynamic Languages*, pp. 25–49, New York, NY, USA, 2007. ACM.

[DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. "Seaside: A Flexible Environment for Building Dynamic Web Applications." *IEEE Software*, **24**(5):56–63, 2007.

[ECM99] ECMA International. *ECMA-262: ECMAScript Language Specification*. European Association for Standardizing Information and Communication Systems, Geneva, Switzerland, third edition, December 1999.

[EH04] Torbjörn Ekman and Görel Hedin. "Rewritable Reference Attributed Grammars." In Martin Odersky, editor, *ECOOP '04: Proceedings of the 18th European Conference on Object-Oriented Programming*, pp. 144–169, 2004.

[EH07] Torbjörn Ekman and Görel Hedin. "The JastAdd extensible Java compiler." In *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 773–774, New York, NY, USA, 2007. ACM.

[FB89] Stuart I. Feldman and Channing B. Brown. "IGOR: a system for program debugging via reversible execution." *ACM SIGPLAN Notices*, **24**(1):112–123, 1989.

[FH06] Richard A. Frost and Rahmatullah Hafiz. "A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time." *ACM SIGPLAN Notices*, **41**(5):46–54, 2006.

[For02a] Bryan Ford. "Packrat Parsing: simple, powerful, lazy, linear time, functional pearl." In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pp. 36–47, New York, NY, USA, 2002. ACM Press.

[For02b] Bryan Ford. "*Packrat Parsing: a practical linear-time algorithm with backtracking*." Master's thesis, Massachusetts Institute of Technology, September 2002.

[For02c] Bryan Ford. "Pappy: a parser generator for Haskell." `http://pdos.csail.mit.edu/~baford/packrat/thesis/`, 2002.

[For04] Bryan Ford. “Parsing Expression Grammars: a recognition-based syntactic foundation.” In *POPL ’04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 111–122, New York, NY, USA, 2004. ACM Press.

[GHJ95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[Gos05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[Gra03] Paul Graham. “Beating the Averages.” `http://paulgraham.com/avg.html`, 2003.

[Gri06] Robert Grimm. “Better extensibility through modular syntax.” In *PLDI ’06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 38–51, New York, NY, USA, 2006. ACM Press.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. “Context-oriented Programming.” *Journal of Object Technology (JOT)*, **7**(3):125–151, March-April 2008.

[HM98] Graham Hutton and Erik Meijer. “Monadic parsing in Haskell.” *Journal of Functional Programming*, **8**(4):437–444, 1998.

[HMP05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. “Composable memory transactions.” In *PPoPP ’05: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 48–60, New York, NY, USA, 2005. ACM.

[HP01] Haruo Hosoya and Benjamin Pierce. “Regular expression pattern matching for XML.” In *POPL ’01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 67–80, New York, NY, USA, 2001. ACM.

[IKM97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. “Back to the future: the story of Squeak, a practical Smalltalk written in itself.” In *OOPSLA ’97: Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 318–326, New York, NY, USA, 1997. ACM.

[Ing08] Dan Ingalls. “The Lively Kernel: just for fun, let’s take JavaScript seriously.” In *DLS ’08: Proceedings of the 2008 Dynamic Languages Symposium*, pp. 1–1, New York, NY, USA, 2008. ACM.

[JD88] Gregory F. Johnson and Dominic Duggan. “Stores and partial continuations as first-class objects in a language and its environment.” In *POPL ’88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 158–168, New York, NY, USA, 1988. ACM.

[Jef85] David R. Jefferson. “Virtual time.” *ACM Transactions on Programming Languages and Systems*, **7**(3):404–425, 1985.

[Joh79] Steven C. Johnson. “YACC: Yet Another Compiler Compiler.” In *UNIX Programmer’s Manual*, volume 2, pp. 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[Joh95] Mark Johnson. “Memoization in Top-Down Parsing.” *Computational Linguistics*, **21**(3):405–417, 1995.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[KIO06] Alan Kay, Dan Ingalls, Yoshiki Ohshima, Ian Piumarta, and Andreas Raab. “Proposal to NSF, granted on August 31st, 2006.” `http://www.vpri.org/pdf/NSF_prop_RN-2006-002.pdf`, 2006.

[KPR07] Alan Kay, Ian Piumarta, Kim Rose, Dan Ingalls, Daniel Amelang, Ted Kaehler, Yoshiki Ohshima, Chuck Thacker, Scott Wallace, Alessandro Warth, and Takashi Yamamiya. “Steps Toward the Reinvention of Programming (First Year Progress Report).” `http://www.vpri.org/pdf/steps_TR-2007-008.pdf`, 2007.

[KR81] H. T. Kung and John T. Robinson. “On optimistic methods for concurrency control.” *ACM Transactions on Database Systems*, **6**(2):213–226, 1981.

[KRB91] Gregor Kiczales, Jim D. Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[LM01] Daan Leijen and Erik Meijer. “Parsec: Direct Style Monadic Parser Combinators for the Real World.” Technical Report UU-CS-2001-35, Universiteit Utrecht, 2001.

[LS90] Michael E. Lesk and Eric Schmidt. “Lex – A lexical analyzer generator.” In *UNIX Vol. II: Research System (10th ed.)*, pp. 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[Mor93] J. Gregory Morrisett. "Generalizing first-class stores." In *SIPL '93: Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, pp. 73–87, 1993.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. "Polyglot: An extensible compiler framework for Java." In *CC '03: Proceedings of the 12th International Conference on Compiler Construction*, 2003.

[NJ06] V. Krishna Nandivada and Suresh Jagannathan. "Dynamic state restoration using versioning exceptions." *Higher-Order and Symbolic Computation*, **19**(1):101–124, 2006.

[Par94] Terence John Parr. "An Overview of SORCERER: A Simple Tree-Parser Generator." Technical report, University of San Francisco, 1994.

[Piu06a] Ian Piumarta. "Accessible Language-Based Environments of Recursive Theories (a white paper advocating widespread unreasonable behaviour)." Technical report, Viewpoints Research Institute, 2006.

[Piu06b] Ian Piumarta. "Open, extensible programming systems." Keynote, *Dynamic Languages Symposium*, 2006.

[PQ94] Terence J. Parr and Russell W. Quong. "Adding Semantic and Syntactic Predicates To LL(k): pred-LL(k)." In *Computational Complexity*, pp. 263–277, 1994.

[PQ95] T. Parr and R. Quong. "ANTLR: A predicatedLL (k) parser generator.", 1995.

[Red08] Roman R. Redziejowski. "Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking." *Fundamenta Informaticae*, **79**(3-4):513–524, 2008.

[Ree78] David P. Reed. "Naming and synchronization in a decentralized computer system (Ph.D. dissertation)." Technical Report TR-205, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[Ree05] David P. Reed. "Designing croquet's TeaTime: a real-time, temporal environment for active object cooperation." In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 7–7, New York, NY, USA, 2005. ACM.

[Ros95] Guido van Rossum. "Python reference manual." Technical Report CS-R9525, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1995.

[SC89] D. J. Salomon and G. V. Cormack. "Scannerless NSLR(1) parsing of programming languages." In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 170–178, New York, NY, USA, 1989. ACM Press.

[Sch64] D. V. Schorre. "META-II: a syntax-oriented compiler writing language." In *Proceedings of the 1964 19th ACM National Conference*, pp. 41.301–41.3011, New York, NY, USA, 1964. ACM Press.

[ST95] N. Shavit and D. Touitou. "Software Transactional Memory." In *PODC '95: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pp. 204–213, 1995.

[Tan08] Éric Tanter. "Contextual values." In *DLS '08: Proceedings of the 2008 Dynamic Languages Symposium*, pp. 1–10, New York, NY, USA, 2008. ACM.

[TES73] L. G. Tesler, H. J. Enea, and D. C. Smith. "The LISP70 Pattern Matching System." In *IJCAI '73: Proceedings of the 3rd International Joint Conferences on Artificial Intelligence*, pp. 671–676, Stanford, MA, 1973.

[Vis97] Eelco Visser. "Scannerless Generalized-LR Parsing." Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[Vis01] Eelco Visser. "Stratego: A Language for Program Transformation Based on Rewriting Strategies." In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pp. 357–362, London, UK, 2001. Springer-Verlag.

[VS07] Eric Van Wyk and August Schwerdfeger. "Context-Aware Scanning for Parsing Extensible Languages." In *GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*. ACM Press, October 2007.

[Wad87] Philip Wadler. "Views: a way for pattern matching to cohabit with data abstraction." In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 307–313, New York, NY, USA, 1987. ACM.

[War07] Alessandro Warth. "LazyJ: Seamless Lazy Evaluation in Java." In *FOOL/-WOOD '07: (Informal) Proceedings of the International Workshop on Foundations of Object-Oriented Languages*, January 2007.

[WSM06] Alessandro Warth, Milan Stanojević, and Todd Millstein. "Statically scoped object adaptation with expanders." In *OOPSLA '06: Proceedings of the 21st ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 37–56, New York, NY, USA, 2006. ACM Press.

[WYO08] Alessandro Warth, Takashi Yamamiya, Yoshiki Ohshima, and Scott Wallace. "Toward a More Scalable End-User Scripting Language." In *International Conference on Creating, Connecting and Collaborating through Computing (C5)*, 2008.

[Yam] Takashi Yamamiya. "JavaScript Workspace." `http://metatoys.org/propella/js/workspace.cgi`.