DEPARTMENT OF COMPUTER SCIENCE

A Programming Language Where the Syntax and Semantics Are Mutable at Runtime

Christopher Graham Seaton

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Enginnering in the Faculty of Engineering

May 2007 | CSMENG-07

Declaration

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Enginnering in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Christopher Graham Seaton, May 2007

SUMMARY

Katahdin is a programming language where the syntax and semantics are mutable at runtime. The *Katahdin* interpreter parses and executes programs according to a language definition that can be modified by the running program. Commands to modify the language definition are similar to those that define new types and functions in other languages, and can be collected into language definition modules.

Katahdin can execute programs written in any language which it has a definition module for. The definition of more than one language can be composed allowing more than one language to be used in the same program, the same file, and even the same function. Data and code is shared between all languages, with a function defined in one language callable from another with no binding layer.

Traditional language development involves using a parser generator tool to generate a static parser from the language grammar. *Katahdin's* approach is to just-in-time compile a parser from the grammar as the program runs. The grammar can be modified as the program runs because the parser can be recompiled to match it.

Recent research into grammar theory with *parsing expression grammars* and parser theory with *packrat parsers* provided the basis for the techniques used in *Katahdin* and will be referenced in this thesis along with an introduction to my additions.

Katahdin is intended as a language independent interpreter and a development platform that can replace traditional static parser generators for language development. With further development *Katahdin* should be useable as a drop in replacement for interpreters and compilers for a wide range of languages, and as a platform for development of new languages.

Using a single runtime to execute any programming language will reduce development costs to businesses. For any combination of languages used in a system, only a single runtime has to be maintained, ported between platforms and supported. The ability to use multiple languages in the same program will allow businesses to use the most appropriate language for any part of the system. For example, an engineer could write the core of a program in *FORTRAN* and then add a user interface written in *Python*, without worrying about interfacing between two runtimes.

ACHIEVEMENTS

- Took recent research into grammar and parser theory that is currently being applied at compile-time and applied it at run-time
- Developed a just-in-time compiler for a packrat parser of parsing expression grammars
- Implemented the *Katahdin* interpreter, standard library and graphical debugger
- Implemented language definition modules for a subset of *Python*, *FORTRAN* and *SQL*
- Made more than 270 version control commits over 240 days (version control log submitted electronically)
- Wrote more than 27,000 lines of *C♯*, *Python* and *Katahdin* code, including prototypes and test routines
- Current implementation of interpreter and standard library is over 19,000 lines of *C♯* and *Katahdin* code
- Submitted theory and implementation details as a paper to the *Sixth International Conference on Generative Programming and Component Engineering 2006*

CONTENTS

Summary **1**

Achievements **2**

Table of Contents **3**

List of Figures **5**

Acknowledgements **6**

1 Background **7**
1.1 Programming Languages 7
1.2 The Advantages of Kathadin's Approach 8
1.2.1 A Single Runtime 8
1.2.2 Language Interoperability 9
1.2.3 A New Concept of Languages 10
1.3 Theory Background 11
1.3.1 Grammars 11
1.3.2 Parsing 12
1.3.3 Semantics 13
1.4 Applied Theory 14
1.4.1 Parsing Expression Grammars 14
1.4.2 Packrat Parsing 16

2 Technical Basis **19**
2.1 The *Katahdin* Language 19
2.1.1 Design 19
2.1.2 Implementation 20
2.2 Grammar 21
2.2.1 Annotations 22
2.2.2 Expressing Recursion 22
2.2.3 Expressing Precedence 25
2.2.4 Expressing Lexemes and White-Space 26
2.2.5 Building Parse Trees 29
2.3 Semantics 29
2.4 Parsing 31
2.4.1 Parsing Recursion 33
2.4.2 Parsing Precedence 36

3 Design and Implementation **38**
3.1 Prototyping 38
3.2 Platform 38

3.2.1 Platform Options 39
3.2.2 Version Control 40
3.3 Implementation 40
3.4 Implementation of the Language 41
3.5 Implementation of the Parser 42
3.5.1 Compiling the Parser 42
3.6 Implementation of the Debugger 43
3.7 File Handling 43
3.8 Standard Library 44
3.9 Language Modules 45
3.10 Testing 47

4 Current Status and Future Plans **48**
4.1 State of Implementation 48
4.2 Unsolved Problems 48
4.2.1 Memory Consumption 48
4.2.2 Speed 48
4.2.3 Error Handling 49
4.3 Future Development 49
4.3.1 Debugger 49
4.3.2 Language Modules 50
4.4 Conclusion 50

Bibliography **51**

LIST OF FIGURES

1.1 For-statement expressed in *Katahdin* 8
1.2 Interoperation between *FORTRAN* and *Python* 9
1.3 PEG grammar requiring backtracking 16
1.4 PEG rewritten to show effect of memoization 17

2.1 The factorial operator expressed in *Katahdin* as a function . . . 20
2.2 Demonstration of problems with prioritised alternative operator 21
2.3 Example *Katahdin* grammar expressions 22
2.4 Annotation syntax 22
2.5 *YACC* grammar fragment showing associativity 24
2.6 *Rats!* grammar fragment showing associativity 24
2.7 Subtraction operator expressed in *Katahdin* 24
2.8 Operator precedence expressed in *Katahdin* 27
2.9 Definition of a modulo operator in *Katahdin* 27
2.10 Definition of an identifier lexeme in *Katahdin* 28
2.11 Setting global white-space in *Katahdin* 28
2.12 Fields in the implementation of the add operator in *Katahdin* . 29
2.13 Use of the `token` operator in *Katahdin* 30
2.14 The factorial operator expressed in *Katahdin* as an expression . 30
2.15 Parsing algorithm for sequence expressions 31
2.16 Parsing algorithm for not-predicates 32
2.17 Parsing algorithm for and-predicates 32
2.18 Parsing algorithm for rules 33
2.19 Parsing algorithm for longest alternative expressions 34
2.20 Modification to parsing algorithm to support non-recursion . . . 34
2.21 Modification to parsing algorithm to support right-recursion . . 35
2.22 Demonstration of the `dropPrecedence` option 36

3.1 Example *Katahdin* syntax tree 41
3.2 Screenshot of the *Katahdin* debugger 44
3.3 The SqliteExpression 46

ACKNOWLEDGEMENTS

I would like to thank Dr. Henk Muller for his guidance throughout this project, particularly with regard to helping me write and submit my paper.

CHAPTER 1

BACKGROUND

1.1 Programming Languages

> "Programming languages are notations for describing computations to people and to machines."[4]

A programming language is an interface between the programmer and the computer. The interface is defined by the grammar that describes the syntax of how keywords, names and symbols are used to build constructions such as expressions and statements, and the semantics that describe what the computer should do for each construct.

Programs are processed by compilers and interpreters that translate the program into a form that can be executed by another program, or execute it immediately. In this thesis I use the informal term *runtime* to refer to any compiler or interpreter tool.

With traditional development tools, each runtime accepts only one language, and every time a new language is designed a new runtime is written from scratch. Almost all programming languages are extremely similar in their design and implementation, and there are no technical reasons why a separate runtime is needed for each language.

Programming languages are very tightly coupled with the implementations of runtimes, and the two concepts are confused. *C* is generally considered a compiled language because a compiler happens to be the most common runtime. There is no reason why *C* cannot be interpreted, and interpreters such as *CINT*[12] are available. In the same way, *JavaScript* is thought of as an interpreted scripting language for web browsers, but there is no reason why it could not be compiled and run on a supercomputer.

Katahdin takes the approach that a programming language is just another *application programming interface* (API), like a library of functions, and as runtimes can use multiple APIs, a single runtime should be able to accept multiple languages. The *Katahdin* interpreter has loadable language definition modules that define the syntax and semantics of a complete programming language, allowing it to execute programs written in any language it has a definition module for.

Figure 1.1: For-statement expressed in *Katahdin*

```
class ForStatement :  Statement {
    pattern {
        "for" "(" init:Expression ";"
            cond:Expression ";" inc:Expression ")"
        body:Statement
    }
    method Run() {
        init.Get...();
        while (cond.Get...()) {
            body.Run...();
            inc.Get...();
        }
    }
}
```

Language definition modules are written like function libraries using the *Katahdin* programming language. The *Katahdin* language is imperative, object-oriented and duck-typed. It should be instantly familiar to anyone who has used *C++*, *Java* or *C♯*. To define new language constructs, the *Katahdin* `class` statement can have an attached pattern which is entered into the grammar. The class is instantiated when its pattern matches. Semantics are defined for each construct in terms of methods written in the existing language. New constructs are therefore defined in terms of the existing constructs.

Figure 1.1 shows how the for-statement is defined in the *Katahdin* standard library, in terms of a while-statement. The details of the `Run()` method and the `init.Get...()` notation will be described later, but the example clearly shows how intuitive it is to define new language constructs in *Katahdin*.

After you have defined your own languages in *Katahdin*, languages beyond that could be defined in terms of your own languages. The semantics of *Python* could be expressed in terms of *C*, for example.

1.2 The Advantages of Kathadin's Approach

1.2.1 A Single Runtime

Running a system that is written in multiple programming languages is currently very difficult. For each language that you use a separate runtime needs to be chosen, evaluated, installed and possibly licensed. Each runtime needs to be maintained with security updates and bug fixes. If you move to a different platform all of the runtimes need to be ported. You may find that a runtime available on one platform is not available on another.

Figure 1.2: Interoperation between *FORTRAN* and *Python*

```
import "fortran.kat";
import "python.kat";

fortran {
    SUBROUTINE RANDOM(SEED, RANDX)

    INTEGER SEED
    REAL RANDX

    SEED = 2045*SEED + 1
    SEED = SEED - (SEED/1048576)*1048576
    RANDX = REAL(SEED + 1)/1048577.0
    RETURN

    END
}

python {
    seed = 128
    randx = 0

    for n in range(5):
        RANDOM(seed, randx)
        print randx
}
```

With *Katahdin* there is a single runtime to maintain. Security updates and bug fixes to the runtime apply to all languages that you use. Only one runtime has to be considered when moving platform.

1.2.2 Language Interoperability

Interoperating between languages that are running on different runtimes is also complicated. If you are running two programs on separate runtimes the only way to share data between them is to set up an input-output channel. This has to be explicitly set up and limits performance. Sharing code such as functions and types is even more complicated, requiring software component protocols such as *CORBA* and *COM*.

When running on *Katahdin* data and code is shared between all languages. A function defined in one language can be called by another without any kind of binding or input-output. Figure 1.2 shows a program with a function written in *FORTRAN* that is called from *Python*.

Katahdin allows for genuine code reuse. If you have a simple function written in *Python* that you want to use from your *Perl* program, currently the only sensible way to achieve this is to rewrite the function. In *Katahdin* the function could be simply copy-and-pasted into the *Perl* program.

1.2.3 A New Concept of Languages

Katahdin makes programming languages just another tool for developers, instead of letting your language choice restrict your choice of platform and libraries. With *Katahdin* you are free to use whatever language you want for any part of the program. You can use the most appropriate language for each part for the program, depending on the task being solved, the available developers and the library of available code.

For example, a numerical processing program written in *FORTRAN* might have occasion to perform some kind of text processing, perhaps as part of input-output. *FORTRAN* is not best suited for text processing so the current best option would be to export data from the *FORTRAN* program, load it into a language such as *Perl* and perform the text processing there. This involves setting up two lots of input-output and managing two separate runtimes. With *Katahdin* the text processing part of the program could be written in *Perl*, in the same file as the *FORTRAN* code.

Beyond the library of definition modules for standard languages, *Katahdin* allows programmers to control their own languages. Programmers can add a new language construct such as a new expression or statement as easily as defining a new function. The concept of domain-specific languages, languages written for a particular purpose in one industry or business, can be extended to application-specific languages – extensions to a language implemented for just one application. *Katahdin* makes this possible because unlike parser generators, the grammar is mutable at runtime and new constructs can be added by a running program and do not involve modifying the implementation of the interpreter.

Even if programmers are not going to develop their own languages, this has concrete applications. For example, when *Sun* wanted to add a for-each-statement to *Java*, users had to wait a complete release cycle as the compiler was updated, builds made and distributed to users to install. With *Katahdin* the new construct could have been defined in a few lines by anyone and distributed as a module for programmers to put in their standard library.

1.3 Theory Background

1.3.1 Grammars

The theory behind the traditional description of the syntax of programming languages is Chomsky's hierarchy of formal grammars[8]. Chomsky described generative grammars for describing human languages such as English, where the concept is recursive rules that produce sequences of lexemes (words and punctuation in the context of human languages, and identifiers, keywords and symbols in the context of programming languages). Generative grammars are suitable for human languages where the focus is on building phrases and there is natural ambiguity, but less suitable for programming languages where the focus is on breaking phrases down, or *parsing* them, and ambiguity is a problem that has to be worked around.

Regular Grammars

Regular grammars are the most restrictive of the hierarchy, recognised by a *finite state automaton*. Regular grammars are applied in lexical analysis, described below, and the regular expressions (RE) often used to express search patterns for text strings.

Context-Free Grammars

The productions of *Context-free* grammars (CFGs) cannot refer to any text around the production, or rule, for context. CFGs are recognised by a *pushdown automaton*, an FSA with a stack data structure that allows it to store a state to return to later. This allows recursive rules.

Context-Sensitive Grammars

The productions in *Context-sensitive* grammars may apply depending on the surrounding text (the context of the production) and are recognised by *linear bounded automatons*, which are Turing machines where memory consumption is linear to the size of the input.

Unrestricted Grammars

Unrestricted grammars include all formal grammars and there are no restrictions on the type of productions in the grammar. Any Turing machine can parse text according to an unrestricted grammar but the complexity and use of resources by the parser are also unrestricted.

Most programming languages are defined using CFGs expressed in Backus-Naur form with informally defined extensions in the form of code actions to support the context-sensitive parts of the language. The syntax of languages including *C* and *Eiffel* is context-sensitive and are implemented using this method.

Opposed to the generative grammars of the Chomsky hierarchy are the recognition-based, or analytic grammars which conceptually focus on parsing texts according to a language instead of constructing them. Ford[11] gives a formal example of the difference between generative and recognition-based grammars for a language consisting of any number of pairs of the character a:

> A generative grammar $\{s \in a^* \mid s = (aa)^n\}$ constructs a string as any number of concatenated a's.
>
> A recognition-based grammar $\{s \in a^* \mid (|s| \bmod 2 = 0)\}$ accepts a string of a's if the length is even.

A *top-down parsing language*[6] (TDPL) is a recognition-based grammar that describes how to parse constructs instead of how to generate them. As will be shown, the *Katahdin* grammar is a form of *parsing expression grammar* and can ultimately be reduced to a TDPL.

1.3.2 Parsing

In traditional language development, parsing begins with a *lexical analysis* of the source text to identify lexemes such as identifiers, keywords and symbols. This is for performance to create an abstracted software architecture. Algorithms for lexical analysis are generally faster and a stream of identified tokens is simpler input for syntax parsing algorithms than a stream of characters.

The lexical definition of the language is described in a separate document from the syntax, using a regular grammar recognised by a *finite state automaton* (FSA). Tools to produce FSAs from regular grammars include *Lex* and *Flex*. The output of these programs is the input to the syntax recogniser.

Despite the described advantages, a separate lexical analysis stage fragments the language definition and restricts language design. The use of regular grammars to define the lexicon of languages is the cause of the irritating and confusing restriction on nesting `/* block comments */` in *C* and *Java* programs. Regular grammars have no state and so would not be able to remember the level of nesting at any point. *Katahdin* uses a single grammar for both lexical and syntactical recognition, allowing for more complex lexemes (including nested block comments) and avoiding the unnecessary fragmentation of the language definition.

The minimal program that can parse the context-free grammars traditionally used to define languages is a *pushdown automaton*. The pushdown automaton's stack allows the parser to recognise more complicated constructs than

the lexer, including nested constructs such as expressions with operands that are other expressions, and the nested block comments that the lexer could not handle.

The two most common algorithms for parsing CFGs are *LR(k)* and *LL(k)*. *LR(k)* parsing is often called *bottom-up* because it recognises the smallest constructs first by applying productions to group tokens, then grouping those constructs into larger constructs. *LL(k)* parsing is conversely called *top-down* because it recognises constructs by applying the largest constructs first and moving down to smaller constructs and finally tokens. Both types of parser often use a state machine and a set of look-up tables for moving between states. Tools to produce *LR(k)* parsers include *Yacc* and *Bison*. Tools to produce *LL(k)* parsers include *Antlr* and *JavaCC*.

The look-ahead variable *k* in *LL(k)* and *LR(k)* refers to the maximum number of tokens that need to be checked ahead of the current position to finally decide which production to take from a set of alternatives. In practice the value of *k* is often about 2 for *LL(k)* grammars and is almost always 0 or 1 for *LR(k)* grammars. The complexity and resource requirements of the parser increase with the variable *k*. As will be shown, the value of *k* required by a grammar does not need to affect parsing performance, and with *Katahdin* language designers can stop worrying about it.

All *LL(k)* and some *LR(k)* grammars can also be parsed by a *recursive descent parser*. Instead of using a state machine each production is implemented as a recursive function that consumes tokens and returns a syntax tree. A TDPL can be seen as a formal description of a recursive descent parser. Recursive descent parsers are often written by hand, as the produced program is very similar to the grammar and a tool does not have to be used. Recursive descent parsers are also the output of many *LL(k)* parser generators including *Antlr*. *Katahdin's* parsing algorithm is a *packrat parser*, a specialised form of recursive descent parser.

1.3.3 Semantics

The semantics of programming languages are rarely defined formally. Where the syntax is defined using formal grammars, at best the semantics are described in a carefully written English document, and at worst defined only by a *reference implementation*, where the semantics are defined as the behaviour of one implementation of the language, bugs and all. The semantics of languages such as *C* and *Java* are defined by documentation, where languages such as *Perl* and *Ruby* have a reference implementation.

Mechanisms do exist for formally describing semantics, such as attribute grammars, originally developed by Knuth for formally describing the semantics of context-free languages[15]. However, they are usually used to specify the context-sensitive parts of a language, to work alongside a context-free grammar. *Z Notation* is a non-executable programming language based on mathematical and set notation designed for formulating proofs of intended system

behaviour, particularly hardware systems. *Z* is considered hard to use and it does not produce an executable definition so there is no equivalent tool to the parser generators that produce executable code from a formal grammar.

The reality of language development is most people do not consider it important to formally define semantics, and so *Katahdin* does not tackle this issue. However, it is possible to improve on traditional language development, where the semantics are implemented throughout the source code of the implementation in the form of routines that walk syntax trees. *Katahdin's* language definition modules couple the syntax of language constructs with the complete implementation of their semantics in a single class definition.

1.4 Applied Theory

1.4.1 Parsing Expression Grammars

A *parsing expression grammar*[11] (PEG) is a recognition-based, or analytic formal grammar. PEGs are usually represented in Backus-Naur form, and so syntactically resemble most CFGs. The major difference between a CFG and a PEG is the alternative operator. In a CFG the alternative operator is represented as | and is applied as a non-deterministic choice. This is where the ambiguity is introduced in a CFG – the parser is free to choose between any of the alternatives. In a PEG the equivalent operator is represented as / and is applied as prioritised choice. The parser must try the alternatives in the order they are specified, starting at the same point for each alternative. The first successfully matching alternative is unconditionally taken. If no alternatives match, the alternative operator fails.

The repetition operators ?, * and + are also defined deterministically, as greedy operators that consume as much of the source text as they can. Support for non-greedy constructs is restored with the two *syntactic predicate*[18] operators, ! and &.

The not-predicate, $!e$, tries to match e. If it matches, the predicate fails, and conversely if e fails, the predicate matches. Whether or not e matches, none of the source text is consumed. The and-predicate, $\&e$ is similar but matches if e does, and not if e doesn't. It also consumes no source text.

PEGs were first described by Ford[11] in 2004, but he demonstrated that they can be reduced to a pair of TDPLs developed by Birman and Ullman in the 1970s, the TMG Recognition Schema[6] and Generalised TS[7].

Formal Description

Formally, a PEG is defined[11] as a 4-tuple $G = (V_N, V_T, R, e_S)$. V_N is a finite set of non-terminal symbols, which in *Katahdin* are the language constructs

represented by classes. V_T is a finite set of terminal symbols, the set of input symbols, which in the case of *Katahdin* is the *Unicode* character set. $V_N \cap V_T = 0$ so the sets of non-terminals and terminals do not overlap. R is a finite set of rules, each a tuple (A, e) that is a production of the form $A \leftarrow e$ where $A \in V_N$ (A is a non-terminal) and e is a parsing expression, defined as below. e_S is the starting parsing expression.

A parsing expression e is minimally defined[11] as:

- ϵ, an empty string which always matches
- a where $a \in V_T$, a terminal, or *Unicode* character in *Katahdin*
- A where $A \in V_N$, a non-terminal, or language construct in *Katahdin*
- $e_1 e_2$, a sequence of other parsing expressions
- e_1 / e_2, a prioritised choice between two parsing expressions
- $e_1 *$, zero-or-more repetitions of another parsing expression
- $!e_1$, a not-predicate

This is the minimal definition. Other operators are defined in terms of the minimal set[1]:

- **.**, any member of V_T, any *Unicode* character in *Katahdin*
- $a_1 - a_2$ where $a \in V_T$, a range of non-terminals, or *Unicode* characters in *Katahdin*
- $e_1 +$, one-or-more repetitions of another parsing expression, defined as $e_1 e_1 *$
- $e_1 ?$, one-or-more repetitions of another parsing expression, defined as e_1 / ϵ so either e_1 or the empty string which will always match
- $\& e_1$, an and-predicate, defined as $!(!e_1)$

Finally, a common pattern:

- **!.**, the end of input

Ford[11] has shown that the minimal set of PEGs expressions can express all *LL(k)*, languages, all deterministic *LR(k)* languages and also some context-sensitive languages.

[1] Described by Ford as part of his reductions[11]

Figure 1.3: PEG grammar requiring backtracking

$a \leftarrow b/c$
$b \leftarrow ex$
$c \leftarrow ey$

Evaluation

Although PEGs are a recent tool for describing grammars, their theory has solid foundations. Ford[11] showed how they can be reduced to TDPLs from the 1970s. The semantic predicates have also been successfully applied in the *ANTLR LL(k)* parser.

Two particular recent applications of PEGs give us a good insight into their capabilities. The *Rats!*[13] parser generator, published summer 2006 as this project was started, is a traditional parser generator producing *Java* code, similar to *Antlr*. It has aims similar to *Katahdin* with modular grammars that are composable, but only at as the development tools are being built. It would be suitable for creating a separate compiler program to support a custom language, but not for user modification of the grammar. Redziejowski's evaluation of PEGs[19], published in February 2007 after this project was well developed, consisted of expressing the *Java* 1.5 grammar in a PEG with successful results. Unlike *Rats!* and *Katahdin*, Redziejowski used a pure PEG. Redziejowski found that his PEG parsed *Java* in "acceptable" time and space, but thought that more work was needed on PEG as a language specification tool. *Rats!* had their approach to solving this problem, which will be described in reference to *Katahdin*. The next chapter presents *Katahdin's* own solutions to the problem of elegantly specifying a language instead of a parsing algorithm.

Application in *Katahdin*

The *Katahdin* grammar is a modified *parsing expression grammar*. Extra *annotation* operators, as described later, simplify description of common language constructs and making modular definition easier.

1.4.2 Packrat Parsing

Parsing expression grammars can be parsed by a simple top-down, recursive descent parser[6]. Such a parser has to backtrack[7] when parsing the PEG / operator – when one alternative fails the next is tried from back at the same starting position. This parsing algorithm is valid but risks exponential runtime. Consider the PEG fragment in figure 1.3.

Figure 1.4: PEG rewritten to show effect of memoization

$a \leftarrow e(b/c)$
$b \leftarrow x$
$c \leftarrow y$

To parse a at character n in the source text, the parser will try to match b and failing that c. When it goes to match b it tries to match e at n and then x. If x fails to match, b fails and the parser moves on to the second alternative, c. When the parser goes to match c it again tries to match e at n.

The parsing of e at character n in the production c is redundant as the parser already did that while trying to match b. It is possible to trade-off the time taken by the second parsing, with the space required to store the result of parsing e at n. To do this, whenever e is parsed the memory is checked to see if the memory contains the result of parsing e at n. If so, the memoized result is used as if it had been just parsed. When e is parsed for the first time, the result, if successful or not, is stored in the memory. This is equivalent to rewriting the fragment in figure 1.3 as shown in figure 1.4.

However, the grammar writer does not have to actually rewrite the grammar. The original clear definition of b as ex can remain while gaining the effect of rewriting to factor x out. This ability to keep clear definitions is one of the key advantages of PEGs.

A parser that does this is known as a *memoizing*, or *packrat* parser[10]. Like *parsing expression grammar*, the term *packrat parser* was coined by Ford in 2002.

The packrat parser's memory is a map of entry tuples to the result of parsing. A memory entry 2-tuple contains the name of the production that was being applied and the position in the source text where it was applied n. *Katahdin* extends the memory entry tuple to include other state information to support the extensions to the grammar and parser.

Evaluation

A packrat parser can parse any *LL(k)*, *LR(k)* [10] or TDPL grammar[9], as well as some grammars that require an unlimited look-ahead k. Packrat parsers are ideal for implementing PEGs[11] and are used by all of the small number of available PEG parser generators.

Ford showed[10] that packrat parsers alleviated the problems of the parser generators of the *YACC* lineage[14]. *Katahdin* works to continue to solve these problems.

Application in *Katahdin*

Katahdin employs a modified packrat parser to support the modified *Katahdin* PEG.

CHAPTER 2

TECHNICAL BASIS

2.1 The *Katahdin* Language

2.1.1 Design

Although *Katahdin* can be viewed as a generic runtime for any language, a base language is provided for implementing the standard library and writing other language definitions. The language should therefore be useable by programmers coming from as many different languages as possible and be powerful and flexible enough to express the semantics of many paradigms and languages.

I designed a language that is:

- **Free-form**. Languages such as *Python* and *occam* express scope using the *off-side rule*[16] where indentation is increased with the scope depth and statements are terminated by a line break. Programmers either love-or-hate this style, so I avoided it.
- **Curly-braced**. Beginning with *BCPL* in the 1960s, curly braces {} have always been used to express scope by the leading languages of the day, including as *C*, *C++*, *Java* and *C♯*. Almost all programmers will have experience of curly-braces.
- **Imperative**. PEGs can be conceived as an imperative grammar so it is natural that the language is also imperative. Most programmers will be familiar with the imperative paradigm.
- **Object-oriented**. Object-orientation is a design well understood by programmers and implemented by many languages. As will be shown, the design of the *Katahdin* grammar could also be described as object-oriented.
- **Dynamically typed**, also known as runtime or latent typing. As in most interpreted or scripting languages, *Katahdin* variables and functions are not typed. Objects carry their type with them, and type compatibility is resolved at the point of execution of an operator by *duck-typing*. Objects are automatically converted between types as needed. *Katahdin* has to

Figure 2.1: The factorial operator expressed in *Katahdin* as a function

```
function factorial(n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

support dynamically and statically typed languages, and dynamic typing seemed the most general of the two typing disciplines.

Figure 2.1 briefly illustrates the flavour of the language by implementing the factorial function[1].

2.1.2 Implementation

The *Katahdin* language is loaded into the grammar data structure during the interpreter's bootstrapping stage at start-up. The language's grammar is expressed in a small *S-expression* style language that is parsed by a hand-written recursive-descent parser. Only the minimal language is defined in the interpreter. This base provides enough power for the standard library, written in *Katahdin* and imported at start-up, to define more complex language constructs.

For example, the for-statement illustrated in figure 1.1 is taken verbatim from the standard library. It defines the higher level for-statement in terms of the base construct while-statement. The for-each-statement is then defined in terms of a for-statement and so on. It is testament to the power of *Katahdin* that operators as fundamental as array subscripting can be implemented from scratch in the language itself.

Beyond the grammar and parsing, the rest of the *Katahdin* language is implemented using traditional techniques. A concrete syntax tree is created by the parser, which is walked to produce a code tree. *Katahdin* evaluates the code tree directly by walking it. An optimisation, discussed in chapter 4, would be to transform to an optimal form, and compile it to machine instructions. The language implementation is further described in chapter 3.

[1] Of course, in *Katahdin* the natural way to implement factorial is not to define a function, but a new operator, as is illustrated in figure 2.14

Figure 2.2: Demonstration of problems with prioritised alternative operator

Module 1:
$e \leftarrow e_1 / e_2$
$e_1 \leftarrow$ '1'
$e_2 \leftarrow$ '2'
Module 2:
$e_2 \leftarrow$ '12'

2.2 Grammar

Grammars in *Katahdin* are expressed using a modified PEG. The notation is very similar to writing a CFG or RE.

Literal text is quoted, `"text"`, and can include standard C-style escape sequences. A range of literal characters can be expressed with the range operator `"a".."z"`. The operator `.` matches any character and is rendered as a full-stop character in the grammar source code. Other constructs are referenced by name, `Expression`.

A sequence can be represented by simply delimiting with spaces, `a b c`. A repetition is represented with the `?*+` operators, meaning zero-or-one, zero-or-more and one-or-more respectively. Expressions can be grouped in parentheses, `()`, to control order of evaluation.

Semantic predicate operators `&!` match an expression but never consume any of the text. The and-predicate `&` matches if the expression does, and fails if it does not. The not-predicate `!` matches only if the expression fails, and conversely fails if it matches.

Alternatives are represented by the `|` operator. Unlike a PEG, which uses the operator `/`, alternatives are chosen for the longest-match and the order the alternatives are listed does not give priority. This may seem like an extreme divergence from a standard PEG, but it is the best solution for modular grammars, where you could reference a construct that is modified by a different module. If the order alternatives are tried is specified by the first module, it can be hard for a second module to add new constructs. For example, figure 2.2 shows two modules. The first module defines e as an alternative between e_1, which is the text '1', and e_2, which is the text '2'. The second module wants to add a new definition of e_2, the text '12'. With prioritised choice the second module's definition is ignored because the e_1 always matches the character '1' before the second module's definition is considered. Longest-matching means whichever definition matches the most source text is used, allowing different modules to work together. PEGs are designed to be greedy, and longest-matching conforms to this. As will be shown, prioritised choice is still part of the *Katahdin* grammar and is employed to implement operator precedence.

Figure 2.3: Example *Katahdin* grammar expressions

Expression to match a universal end-of-line

```
"\r" "\n"?  | "\n"
```

Expression to match a comma-delimited parameter list

```
"(" (Value ("," Value)*)?  ")"
```

Expression to match a C-style line comment

```
"//" (!EndOfLine .)*
```

Figure 2.4: Annotation syntax

```
{
    option name = value;
    option name;
    expression...
}
```

2.2.1 Annotations

A PEG can express all *LL(k)*, deterministic *LR(k)* and some context-sensitive languages, but we want a grammar that can express them all *well*. That is, the grammar should succinctly express the language as the programmer conceives it, and not as a parsing algorithm. To solve the problems described in the following sections, the *Katahdin* grammar includes a mechanism for annotating expressions as conforming to a common pattern that the parser understands how to apply. Annotations are applied to blocks, delimited with curly braces, and the `option` keyword, as shown in figure 2.4.

There are several options defined in the *Katahdin* grammar, described below. They can be set to a specific value by using the = operator, or the = and value can be omitted to set the option to the value *true*. More than one option can be set in a block.

2.2.2 Expressing Recursion

A common problem when writing a grammar is expressing rules that recurse and solving the ambiguity that they introduce. For example, mathematical operators, which are present in almost all programming languages, are defined recursively – subtract is an expression and is defined as two expressions with a - symbol between them. Those two expressions could be add expressions themselves. We most naturally conceive the definition of subtract as:

```
Expression ← Number
Expression ← Expression '-' Expression
```

For example, this definition can be applied to the following source text:

$$a - b - c$$

Based on our definition, this expression is ambiguous and has two interpretations:

$$((a - b) - c)$$
$$(a - (b - c))$$

In mathematics there is a convention of taking the first interpretation to solve the ambiguity. This is known as *associativity* – the subtraction operator is *left-associative* and binds tightest to the left of the expression. Even though associative operators are a fundamental part of almost all programming languages and always present the same problems, implementing associativity is still unnecessarily complicated with many traditional language development tools.

For example, if we used our definition of the subtract operator in a standard PEG or *LL(k)* grammar, the parser would fail. The definition of `Expression` begins with `Expression`, so the parser will be forever trying to match `Expression` and will eventually run out of stack space.

One solution is to rewrite recursion as repetition:

```
Expression ← Number ('-' Number)*
```

This is not ideal because instead of expressing the operator as it is used, we are now expressing an algorithm to parse it. We are no longer specifying a grammar, but a parser. Also, `Expression` now matches once for any number of subtraction operators, instead of once per operator which is a better model of the language.

Another method, given by Ford as a solution to the problem in PEGs[9, page 40], is to use a separate suffix pattern:

```
Expression ← Number  Suffix
Suffix ← '-' Number  Suffix
Suffix ← ϵ
```

Figure 2.5: *YACC* grammar fragment showing associativity

```
%left '-'
%%
expression :  expression '-' expression
        | number ;
```

Figure 2.6: *Rats!* grammar fragment showing associativity

```
Expr "-" Expr -> Expr left, cons("Minus")
```

This is even worse. Where we had one rule for one operator, we now have three. This grammar doesn't match our conception of the subtraction operator at all. These two parsing algorithms are used in almost all grammars written for tools such as *Antlr*, but there is a better solution.

LR(k) parser generators, such as *YACC*[14], *BISON* and their descendants allow the grammar writer to express a natural definition and then declare the associativity, as shown in figure 2.5.

YACC's particular implementation is not relevant to *Katahdin* because it uses the very different *LALR(1)* table-based algorithm, but we adopt the idea.

Rats!, which uses a PEG and packrat parser, also takes a similar approach, shown in figure 2.6.

Katahdin's solution is to use an annotation. The operator is expressed naturally and then annotated as being left-associative. *Katahdin* uses the term *left-recursive*, as the name describes what the option does, rather than what it is for. The complete expression for the subtraction operator in *Katahdin* is therefore shown in figure 2.7.

Rules can be annotated with the options `leftRecursive` or `rightRecursive`. Right-associative operators (represented in *Katahdin* as right-recursive) in-

Figure 2.7: Subtraction operator expressed in *Katahdin*

```
class Subtract :  Expression {
    pattern {
        option leftRecursive;
        Expression '-' Expression
    }
}
```

clude the exponentiation, or power, operator. The default is for rules to be simple `recursive` to allow any recursion. This option can be negated by assigning `false` to it to prevent recursion.

The *Katahdin* method is superior to *YACC's* because the declarative component is coupled with the syntax expression.

2.2.3 Expressing Precedence

Precedence, or *priority*, is a property of operators that describes the order in which they should be applied when there is more than one operator in an expression. In mathematics and most programming languages the $*$ and $/$ operators have higher precedence over the $+$ and $-$ operators and so are applied to operands around them first. They are said to bind tighter as they take operands from around them before other operators do. The $+$ and $-$ operators are applied next. When there is more than one operator with the same precedence, they are applied together in a single left-to-right pass:

$$a + b * c / d - e$$
$$a + ((b * c)/d) - e$$
$$((a + ((b * c)/d)) - e)$$

As with the other property of operators, their associativity, in most other grammars and standard PEGs the grammar writer is required to express operator precedence by writing a parsing algorithm. In the example below, a combination of the repetition and suffix patterns is used to implement associativity. Precedence is implemented by making the rules cascade, so multiplicative operators are only considered as an operand to additive operators:

```
Expression ← Term  ExpressionSuffix*
ExpressionSuffix ←  '+'  Term
ExpressionSuffix ←  '-'  Term

Term ← Factor  TermSuffix*
TermSuffix ←  '+'  Factor
TermSuffix ←  '-'  Factor

Factor ← Number
```

As with the solutions to associativity, this is an expression of a parsing algorithm and not a grammar. A further problem is that for a single number, both the `Factor` and `Term` rules will match even though those language constructs are not present. Modularity is reduced because adding a new operator in the middle of the precedence order will require changing unrelated rules either side of this position, and if another module had already inserted rules, you would not know which rules you had to modify.

The *YACC* solution is to read the order of precedence from the order that operator's associativity is set. This is simple and obvious, but doesn't allow for inserting a rule into the precedence order after it has been established. Consequently, the whole precedence order has to be defined as one operation, and away from the expression of the actual patterns.

The *Katahdin* solution shown in figure 2.8, is to naturally express the operators as before, and to annotate them using the *precedence-statement*, another form of grammar annotation. The precedence-statement establishes a precedence relationship between two rules. Multiple precedence-statements build up a precedence order between many rules.

The *Katahdin* solution makes it easy to define a new rule and insert it into the order of precedence. For example, figure 2.9 shows a new *C*-style modulo operator defined with left-associativity and the same precedence as the division operator.

2.2.4 Expressing Lexemes and White-Space

White-space is the non-printing characters such as spaces and new lines used along with punctuation to delimit lexemes. The text 'ab' is a single lexeme unless white space or punctuation is used to separate the characters; 'a b', 'a.b'. Programmers also use white space to format their code for better human readability, and some languages, such as *Python* and *occam*, use white space to express scope.

Traditional language development uses two different tools for the lexical and syntactical analysis. White-space is normally only a lexical issue and so is discarded at that stage. Where it does have relevance to the syntax, it is passed as a token, like any other printing operator. At the syntax analysis stage, the grammar writer does not need to deal with the white-space. In a PEG the two stages are combined for a single definition of the language. A pure PEG needs to explicitly allow white space wherever it is permissible:

```
Expression ← Number  WhiteSpace? '+'  WhiteSpace?  Number
```

However, this is not convenient and doesn't fit with how programmers naturally conceive syntax. Another option is to always allow white-space unless it is explicitly disallowed. The *Rats!* parser generator takes this approach by separating lexical and syntactical rules. White-space is then not matched within a lexical rule. In *Katahdin* I took the same approach but as I have not created a separation between lexicon and syntax, an annotation is used to disable white-space, as shown in figure 2.10.

The white-space annotation is passed down into rules that are matched so that it can be used to set the global white-space pattern for a language, as shown in figure 2.11.

Figure 2.8: Operator precedence expressed in *Katahdin*

```
class AddExpression :  Expression {
    pattern {
        option leftRecursive;
        Expression '+' Expression
    }
}

class SubExpression :  Expression {
    pattern {
        option leftRecursive;
        Expression '-' Expression
    }
}

precedence SubExpression = AddExpression;

class MulExpression :  Expression {
    pattern {
        option leftRecursive;
        Expression '*' Expression
    }
}

class DivExpression :  Expression {
    pattern {
        option leftRecursive;
        Expression '/' Expression
    }
}

precedence DivExpression = MulExpression;
precedence MulExpression > AddExpression;
```

Figure 2.9: Definition of a modulo operator in *Katahdin*

```
class ModExpression :  Expression {
    pattern {
        option leftRecursive;
        Expression '%' Expression
    }
}

precedence ModExpression = DivExpression;
```

Figure 2.10: Definition of an identifier lexeme in *Katahdin*

```
class Identifier {
    pattern {
        option whitespace = null;
        ("a".."z")+
    }
}
```

Figure 2.11: Setting global white-space in *Katahdin*

```
class Whitespace {
    pattern {
        "\r" "\n"?  | "\n"
    }
}
class Program {
    pattern {
        option whitespace = Whitespace;
        Statement*
    }
}
```

Figure 2.12: Fields in the implementation of the add operator in *Katahdin*

```
class AddExpression :  Expression {
    pattern {
        option leftRecursive;
        a:Expression '+' b:Expression
    }
}
```

2.2.5 Building Parse Trees

The grammar described so far can be applied to verify the conformance of a source text. To implement the semantics, a data structure representing the source text according to the grammar is needed.

In most parser generators, the grammar writer annotates tokens that they want to make root nodes in the syntax tree. Unannotated tokens become leaves. For example, in the *ANTLR* parser generator a caret operator can be applied to text tokens to make them a root node:

```
expression :  term (ADD^  term)* ;
```

In *Katahdin* there is no concept of root nodes because an object is instantiated for each matching rule. Instead of child trees, the object has fields that are set to text values or other objects. Fields in the object are set with the `:` field operator. This object-oriented approach to tree building in the grammar reduces work for the author because they don't have to think about how the tree should be constructed and which token to make the root.

Figure 2.12 shows the add operator with its two operands labelled `a` and `b`.

With rules describing lexemes, the grammar writer will want the complete text, not a list of characters. This is expressed with the `token` operator that returns all the text that matched within it as a single value, as shown in figure 2.13.

2.3 Semantics

In most programming languages implementations, the parser produces a syntax tree data structure by applying the grammar to the source text. The nodes of the syntax tree are *walked* or *visited* by functions that either emit code that executes, in the case of a compiler, or actually executes, in the case of an interpreter, the semantic meaning of each construct the programmer used. The tree

Figure 2.13: Use of the `token` operator in *Katahdin*

```
class Identifier {
    pattern {
        option whitespace = null;
        text:token("a".."z")+
    }
}
```

Figure 2.14: The factorial operator expressed in *Katahdin* as an expression

```
class FactorialExpression :  UnaryExpression {
    pattern {
        option leftRecursive;
        a:Expression "!"
    }

    method Get() {
        return factorial(this.a.Get...());
    }
}
```

walker is either written by hand or in a separate grammar[2], so each construct's semantics are separated from the syntax.

In *Katahdin* each language construct is defined as a class with a rule. The class is instantiated when the rule matches, with its fields set to values of sub-expressions that matched. To add the semantic definition, the programmer defines methods in the class. For example, all expressions have a `Get()` method that evaluates the expression. Figure 2.14 shows an implementation of the factorial operator, where the `Get()` method calls the function that was defined in figure 2.1. Note that when the `Get()` method is defined the operator isn't declared yet, so we cannot define the factorial operator in terms of itself and must call a recursive function.

The `Get()` method is an informal convention. Expressions also have a `Set(v)` method for assignment, and statements have a `Run()` method that does not return a value. Grammar writers are free to use any method convention that they choose to implement their semantics.

Calls by a method such as `Get()` are often called *in the parent's scope*. This is expressed using the `a...()` call operator. Calling in the parent's scope means that variables the user referenced are resolved in the original method's scope,

[2]In *ANTLR* a grammar can be defined to parse the tree data structure and perform actions on each matched node.

Figure 2.15: Parsing algorithm for sequence expressions

```
define parse((e_1, ..., e_n), m, s)
    p_0 = p
    t = new parse tree
    for e ∈ (e_1, ..., e_n)
        t_e = parse(e, m, s)
        if t_e == fail
            p = p_0
            return fail
        else
            t = t + t_e
        end
    end
    return t
end
```

and not the scope of the method that implements the constructs semantics.

2.4 Parsing

The *Katahdin* grammar is parsed by a modified packrat parser. Unlike some other grammars, TDPLs including PEGs describe the parsing algorithm. The *Katahdin* grammar has some modifications that require more work, and these are described below, but the basic parsing algorithm for the *Katahdin* grammar is very simple.

Shown in figure 2.15 is the parsing algorithm for sequence expressions expressed in pseudo-code. Input to the function is the sequence of expressions, $(e_1, \cdots, e_n)$, the parser memory, m, and a state, s. The function stores the current position of the parser in the source text, p, and creates a new parse tree data structure, t.

For each expression e in the sequence, the parse function is called. If the expression e fails to parse, the parser is returned to the position in the source text it was at at the start of the function call, and the sequence expression fails, returning an instance of the parse tree data structure that represents failure.

If the expression e matches, its parse tree is used to extend the sequence's parse tree, creating a list of matched values. If all expressions match, the successful parse tree is returned.

You can see that if the expression e fails, all of the intermediate data is discarded. The memoization part of the packrat parser is implemented elsewhere and is described below.

Figure 2.16: Parsing algorithm for not-predicates

```
define parse(!e, m, s)
    p_0 = p
    t = parse(e, m, s)
    if t == fail
        return succeed
    else
        p = p_0
        return fail
    end
end
```

Figure 2.17: Parsing algorithm for and-predicates

```
define parse(&e, m, s)
    p_0 = p
    t = parse(e, m, s)
    if t == fail
        return fail
    else
        p = p_0
        return succeed
    end
end
```

The repetition expressions are defined like the sequence expression, except that after a minimum number of repetitions if e fails, the function stops repeating and succeeds. If the minimum repetitions are not met, the function fails.

Figures 2.16 and 2.17 show the parsing algorithm for the two predicate operators.

Figure 2.18 illustrates the packrat algorithm in the function that parses rules. Passed to the function is the tuple, (r, e), the rule and its root expression. The function creates a memory tuple key that combines the rule and the position in the source code. Any application of rule r at position p will return the same parse tree, so we look up the tuple key in the memory, m, before parsing. If the memory contains a parse tree against the tuple key, it is returned. If not, the rule's root expression is parsed, and the result is stored in the memory and returned, whether or not it succeeded in parsing. Storing a fail in memory is no different to storing a success.

In *Katahdin*, the standard alternative expression, as expressed with the | oper-

Figure 2.18: Parsing algorithm for rules

```
define parse((r, e), m, s)
    k = (r, p)                          Create a memory entry tuple key
    if k ∈ m                            See if the tuple key is in the memory
        return m_k
    else
        t = parse(e, m, s)
        m_k = t                         Store the parse tree in memory by the tuple key
        return t
    end
end
```

ator, returns the longest match. Figure 2.19 shows the algorithm.

2.4.1 Parsing Recursion

The parser supports four kinds of recursion, set with the `leftRecursive`, `rightRecursive` and `recursive` options. Rules have the `recursive` option set by default, but this can be negated to make a rule non-recursive.

Non-recursive rules

When the `recursive` option is set to `false` a rule is non-recursive. The parser function for rules is modified to enter each rule into an exclusion list when it begins parsing the rule, and removing it after it has parsed. Before adding to the list, the function checks if the rule is already in it. If so, the rule fails. Figure 2.20 shows the modifications made.

Recursive rules

When the `recursive` option is set to `true`, as is the default, a rule is recursive, and is not added to the exclusion list.

Right-recursive rules

Right-recursive rules, marked with the `rightRecursive` option, are parsed by adding the constraint that a rule can only recurse into itself on the right-hand side of the expression. Right-hand side can be simplified to anything beyond the first node in a sequence.

Figure 2.19: Parsing algorithm for longest alternative expressions

```
define parse((e_1|···|e_n), m, s)
    p_0 = p
    p_longest = 0                          Variable for position of end of longest match
    t_longest = 0
    for e ∈ (e_1|···|e_n)
        t_e = parse(e, m, s)
        if t_e ≠ fail
            if p > p_longest               If this e matched the most so far
                p_longest = p
                t_longest = t_e
            end
        end
        p = p_0
    end
    if p_longest > 0                       If there was any match at all
        p = p_longest                      Move to the end of the longest match
        return t_longest
    else
        return fail
    end
end
```

Figure 2.20: Modification to parsing algorithm to support non-recursion

```
define parse((r, e), m, s)
    ...
    if r ∈ s_excluded
        return fail
    else
        push(s_excluded, r)
        t = parse(e, m, s)
        pop(s_excluded
        ...
    end
end
```

Figure 2.21: Modification to parsing algorithm to support right-recursion

```
define parse((e_1, ··· , e_n), m, s)
    ...
    for e ∈ (e_1, ··· , e_n)
        ...
        if right-recursive
            pop(s_excluded)
        end
    end
    ...
end
```

The rule is parsed as if were non-recursive, with the algorithm shown in figure 2.20. However, the sequence parse function is modified to remove the rule from the exclude list after the first node matches, as shown in figure 2.21.

For example, when parsing the source text `a ^ b ^ ^ c`, the parser would go to match a power rule. The power rule is excluded, and then the first operand is matched. As the power rule is excluded, only the identifier expression `(a)` matches. The exclusion is then lifted and for the second operand the rule can recurse and match another power rule, `(b ^ c)`. The first call to the power rule then matches `(a ^ (b ^ c))`.

This effectively implements the following pattern:

```
Power ← Number ('^' Power)?
```

Left-recursive rules

Left-recursive rules, marked with the `leftRecursive` option, are implemented by parsing the rule as non-recursive, and then reapplying the rule with the left-hand side set to the result of the previous match.

For example, when parsing the source text `a + b + c`, the parser disallows recursion and the add rule matches `(a + b)`. The add rule is then applied again, using that result for the left-hand operator, instead of parsing from the source text. From the second `+` operator the second invocation of the add rule continues to parse normally, matching `(a + b) + c)`.

This effectively implements the following pattern:

```
Add ← Number ('+' Number)*
```

Interestingly, this method of matching the operators from those furthest down

Figure 2.22: Demonstration of the `dropPrecedence` option

```
class Parentheses :  Expression {
    pattern {
        option dropPrecedence;
        "(" Expression ")"
    }
}
```

the syntax tree first could be called a *recursive-ascent* parser, although the term usually refers to a method of *LR(0)* parsing[17].

To implement this algorithm, left-recursive rules initially behave according to the non-recursive algorithm. After successfully matching, they store the result in the status to be used as the left-hand side, and call themselves again. Rules check the left-hand side variable in the status, and if it is set and is of a compatible type (if the rule is an expression, add and multiply rules in the left hand side would be compatible as they are subclasses of expression), return the left-hand side as if it had just been parsed.

2.4.2 Parsing Precedence

To implement precedence, the parser maintains a current precedence level. When entering a rule the precedence level of the rule is compared to the current level. Rules fail if they are not of higher precedence than the current level. While parsing the rule, the precedence level is raised to that of the rule. This prevents rules of a higher precedence matching operands that are of lower precedence. For example, a multiply rule cannot have add rules as its operands, as add has a lower precedence than multiply.

Some language constructs, such as parentheses, override precedence. The expression `a * (b + c)` overrides the normal precedence of operators by putting the add operator in brackets as an operand to the mulitply operator. In *Katahdin* this can be annotated with the `dropPrecedence` option that instructs the parser to reset the current precedence level to zero, as show in figure 2.22.

Unlike standard PEGs, *Katahdin* uses a longest-match alternative operator, instead of prioritised choice, however prioritised choice is used to solve a further problem of precedence. Consider a grammar with add and subtract operators, defined with lowest precedence, multiply and divide operators with higher precedence, and numbers with highest precedence. The algorithm described so far would restrict the operands of the add operator being either multiply, divide or a number. With the *Katahdin* longest-match alternative operator, all three rules would be tried before the longest being returned, however multiply and divide have a higher precedence and so should be tried first, and if either of those two match the alternative should return it. Only if both multiply and

divide fail should number be tried, as it has higher precedence.

To implement this, the alternative operator sorts the alternatives into groups by precedence:

    (add subtract) (multiply divide) number

The alternative groups are tried with a standard PEG prioritised choice algorithm, but within each group the longest-match is taken.

This is also an optimisation – groups of rules that would fail the precedence test if they were parsed can be quickly discarded rather than by testing each rule.

CHAPTER 3

DESIGN AND IMPLEMENTATION

3.1 Prototyping

Katahdin development began with a prototype parser written in the *Python* programming language. The grammar parser was created with the *ANTLR* parser generator and then separate input files were parsed using the *Katahdin* parser according to that grammar. Using this experience I produced a second *Python* prototype, before starting development of the production implementation.

I also prototyped a simple programming language implementation using *C♯* and the *.NET* platform, to see if it was a feasible option.

3.2 Platform

My parser prototypes were written in the *Python* language because it is a good *rapid application development* tool, with high level constructs and dynamic typing. However, *Python's* runtime speed performance is low and the scripting nature of the language means that it is not best suited for large-scale systems software such as *Katahdin*.

My requirements for a platform were:

- **Runtime speed performance** The platform has to run reasonably fast, as *Katahdin* was not designed with performance as a goal, a fast base was needed to build on.
- **Dynamic code generation** In order to achieve reasonable speed, I wanted to compile the parser as described later. This requires that the platform has some facility for dynamically generating code that can be executed as fast as the compiled interpreter.
- **Available languages** As has been described, the choice of a traditional platform dictates which languages can be use. I needed a language that I was proficient in, or that would be easy for me to learn.
- **General purpose features** *Katahdin* will be used to implement the semantics of many languages, so a platform that is too specialised might

not have the features needed for some languages. For example, the *Ruby* platform has very poor threading support, so implementing languages that have powerful threads would be hard.

3.2.1 Platform Options

New platform from scratch

One option was to use a compiled programming language and develop my own platform from scratch. This would have given me lots of freedom to implement the system without restricting it to fit in the features of a platform. Libraries could have been used for basic data structures and facilities such as garbage collection, so I don't think that development time would have suffered significantly.

Languages such as *C* that allow programmers to directly access memory and the system API can easily generate compiled machine instructions at runtime. However, researching how to write and encode machine instructions would be very time consuming, and I personally use a different hardware architecture to the departmental lab computers so the work would be duplicated. One possibility would be the *libjit*[1] library from *Southern Storm Software*, which provides a well developed *C* interface for compiling bytecode to machine instructions for several architectures.

Apart from *C* I also considered writing a platform in *C++* and *Objective-C*.

I finally rejected the idea because I thought that bringing together lots of different libraries to develop a new platform would be too cumbersome, and I wanted well developed, integrated platform.

Java

The *Sun Microsystems Java* platform is a virtual machine for compiled bytecode, often written using the *Java* programming language. *Java* is fast, has a huge standard library of data structures and utilities and is easy to use, with exception handling and garbage collection.

There are several interpreters available for the *Java* platform that use code generation, such as *Jython*, but the method used is impractical. The interpreter manually writes a *Java* class file to a data stream that is then read as if it was loading a file from disk. The interpreter has to perform the instruction encoding and the smallest unit of code that can be dynamically generated is a complete class definition.

I rejected *Java* because of its poor support for dynamic code generation.

.NET

The *Microsoft .NET* platform was created as a competitor to *Java* and is extremely similar in design and application. Like *Java* it is fast, has a good standard library, exception handling and garbage collection. The *C♯* language is the *.NET* equivalent of the *Java* language. Unlike *Java*, *.NET* and *C♯* are independently standardised as the *Common Language Infrastructure* (CLI), *ECMA-334* and *ECMA-335*.

Most importantly, *.NET* includes excellent support for dynamic code generation. The standard library includes a high level interface for emitting bytecode instructions in blocks as small as standalone methods, using the `System.Reflection.Emit` API.

I chose to develop *Katahdin* using the *.NET* platform and the *C♯*language because of the speed, the good standard library including support for dynamic code generation, and because I am personally experienced using *.NET* and *C♯*.

However, I did not use the *Microsoft* implementation of *.NET* because it is not available for the *Linux* or *Apple MacOS X* operating systems which I use. Instead I used the popular and mature *Novell* implementation, *Mono*[2].

3.2.2 Version Control

Throughout development I used the *Bazaar*[3] distributed version control system. Version control allowed me to track the changes that I made and reverse them when I wanted to. I also used *Bazaar* as a synchronisation tool between the departmental laboratory and my own computer, and as a daily form of backup. Unlike *CVS* and *Subversion*, *Bazaar* is distributed, with each checkout having a complete copy of the revision history, and so allowed me to work off-line.

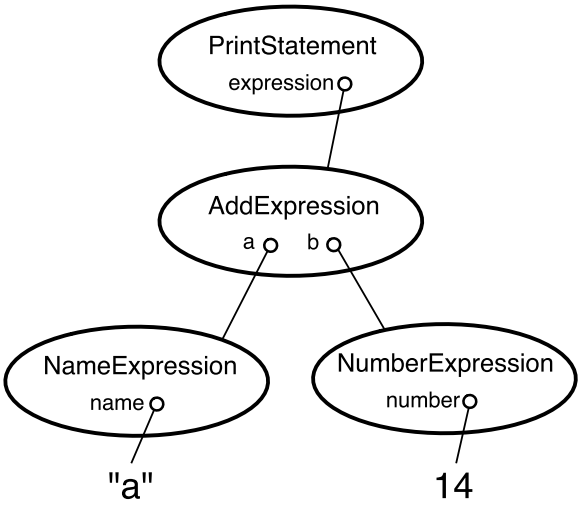
The log of my version control commits has been electronically submitted.

3.3 Implementation

My implementation of the *Katahdin* language interpreter is intended as a production quality software system. The current status is described in detail in the next chapter, but as an overview the implementation is a large but mature and stable piece of software of over 19,000 lines of code. There is a lot of missing minor functionality that would be trivial to implement but has never been exercised by a test program or demonstration and so has not been looked at yet. For example, conversion is undefined between many types, some operators can only be applied to certain types, and so on.

Work is also needed on areas including performance and error handling. These are described in the next chapter.

Figure 3.1: Example *Katahdin* syntax tree



3.4 Implementation of the Language

The implementation of the *Katahdin* language, beyond the parsing stage, is similar to most traditional interpreted languages. The actual *Katahdin* executable is a command-line program that accepts a list of files to interpret. The standard library is prepended to this list unless explicitly turned off.

```
$ katahdini program.kat
```

Each file is parsed by the *Katahdin* parser, producing a syntax tree data structure. Unlike most traditional language implementations, the syntax tree is strongly typed. Figure 3.1 shows the *Katahdin* syntax tree data structure for the statement `print a + 14;`. Each node in the tree is an instance of the class that defined the matched rule. Children are referenced by fields in the objects. The leaf objects are standard `String` and `Int32` nodes.

To execute a program expressed in a syntax tree, the program is translated to a code tree. While the syntax tree expresses the program, the code tree expresses how to execute it. This data structure has nodes that perform the semantic operation of each construct in the syntax tree. Each node in the syntax tree has a `BuildCodeTree()` method to generate the code tree for that node, and nodes with children recursively call the child nodes' `BuildCodeTree()` method and combine the trees.

When defining new language constructs the `Run()`, `Get()` and `Set()` methods, as described in 2.3, effectively implement `BuildCodeTree()`. However, instead of the grammar writer creating and returning a code tree, the code tree of the method itself is returned.

The code tree is then either evaluated, by walking the tree, or stored for evaluation later if the code tree is the body of a function or method.

3.5 Implementation of the Parser

As a rule is defined, an expression tree data structure is created. Each operator in the *Katahdin* grammar has a corresponding class in the parser. Each class has a `Parse()` method which implements the *Katahdin* parsing algorithm and returns a parse tree data structure.

To parse each input file the root rule is applied by calling its `Parse()` method. As the expression tree is walked through the mutually recursive calls to the nodes' `Parse()` methods, the grammar is applied to the source text and the parse tree is built up. If the root rule fails, or the grammar does not parse up to the end of the source text, an error is detected.

Shared parsing data structures such as the memory, the exclude list and the left-hand side are passed to methods in a `ParserState` data structure. The memory is a key data structure for performance. While running my simple demo programs the memory contains several thousand intermediate parse trees, and a look-up is performed in the `Parse()` method every time the parse tries to apply a rule. I used a hash map data structure, with the key being a non-linear combination of the rule's unique identifier and the position within the source text.

3.5.1 Compiling the Parser

The implementation of the parser as described above is fast enough to be useable, but is still very slow because of the huge number of method calls and the generic implementation of the methods that have to work for all parameters.

To improve performance I applied a technique from traditional development tools – the parser generator. A parser generator takes a grammar definition and produces a program to parse it. This is faster because the overhead of the code to walk the grammar data structure is removed and instead of a method call per expression, a single method can parse an entire rule. Also, the implementation of each expression node can be changed based on the parameters. For example, a compiled zero-or-more repetition node can entirely omit the code to check the minimum number of repetitions.

In *Katahdin* the parser *just-in-time* compiles the expression tree of each rule into a single method of *.NET* bytecodes. The *.NET* runtime compiles the bytecode to the machine instructions of the hardware architecture and creates a callable method object.

The parser compiler is a *just-in-time* compiler because each rule's method is compiled only as it is called for the first time. This is implemented using a

trampoline method that calls the compiler the first time, and calls the method from there on in. If the grammar changes, with the definition of a new rule or a change in the precedence, all affected rules have their compiled methods discarded and their trampolines reset, ready to be compiled again the next time they are called.

The compiled grammar reduces runtime for the `demos/fortran-python/fusion.kat` demonstration program from **4 minutes 40 seconds** down to **26 seconds**, a **90%** reduction.

There is potential for the application of *adaptive optimisation*[5] here. The compiled rules could be instrumented with a profiler and the most frequently called or slowest rules recompiled with more aggressive, time consuming optimisations. The *Java Hotspot* virtual runtime uses this technique.

3.6 Implementation of the Debugger

The *Katahdin* debugger is a graphical interface for viewing the internal data structures of the *Katahdin* interpreter. I found it impractical to use a standard debugger to understand why the parser failed when it did. Usually it involved the interaction of several very large data structures and errors were easier to spot if I could control the amount of data by displaying it in a tree form, able to collapse and expand branches.

The debugger also has a trace feature, which creates a tree of all the rule calls and the evaluation of every operator as the parser runs. This makes it very easy to see where the parser took a wrong turn. The screenshot in figure 3.2 shows how much information the debugger gives you. You can see the grammar, with the tree of expression operators, the parse tree which shows the output data structure from the parser, and the parse trace. In the parse trace you can see how the parser has applied the `Whitespace` rule between tokens in the print-statement, which were not referenced in the original grammar, but have been automatically inserted. You can also see the runtime object window, which lets you examine all objects used by the interpreter.

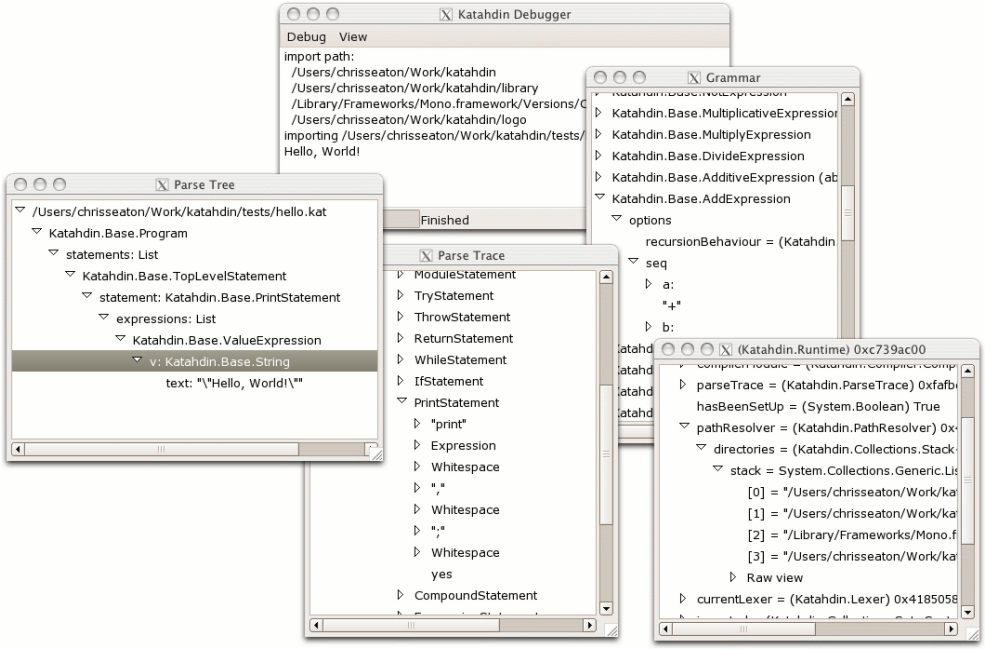
The user interface is written using *Gtk♯*, a *C♯* binding of the *Gtk+* library. Currently, the debugger does not work with the *just-in-time* compiler for the parser, and so is very slow.

The *Katahdin* debugger currently has no tools for debugging code running in *Katahdin* – it is a debugger for *Katahdin*, not a debugger for *Katahdin* programs. However, it could be extended to provide this functionality.

3.7 File Handling

Files to be interpreted are passed to *Katahdin* on the command line. Large programs will be broken down into many files and libraries will always be written

Figure 3.2: Screenshot of the *Katahdin* debugger



in separate files. A program can instruct the runtime to interpret another file using the import statement, similar to `#include` in *C*, but not unlike `imports` in *Java* and `using` in *C♯* which just reference namespaces for resolving type names.

```
import "second-file.kat";
```

To find a file that is imported, *Katahdin* searches a list of directories, beginning with the same directory as the importing file and then moving to the list of directories specified by the environment variable `$KATAHDIN`, defined with the same format as the system `$PATH` variable.

3.8 Standard Library

The *Katahdin* standard library is automatically imported as the interpreter starts and defines data structures and higher level language constructs in terms of the base language. The standard library also provides a utility layer between *Katahdin* and the features of the underlying *.NET* platform. For example, the subscript operator defined in `library/subscript.kat` is defined for several different *.NET* interfaces so that it can be used with as many *.NET* types as possible.

Figure 1.1 in chapter 1 shows an the definition of the for-statement taken straight from the standard library.

3.9 Language Modules

A language definition module in *Katahdin* is a module that defines all of the language constructs used in a language.

There are several ways to use a language module. One option is to use the *Katahdin* runtime as if it were an interpreter for another language. The standard library includes a directory of file extensions and associated language modules in `library/languages.kat`. If a file is imported that does not have a .kat extension, the runtime will look up the language module to load and use. Files written in any language that is listed in the directory can then be run by the *Katahdin* interpreter:

```
./katahdini katahdin.kat python.py fortran.f
```

Another way to use a language module is to import the module and then use the language only within part of the file. The language modules all define a new *Katahdin* language statement that switches language just for the contents of a pair of curly braces, as shown in figure 1.2 in chapter 1, and below:

```
import "python.kat";

python {
    Python code here...
}
```

Finally, language modules can define a finer integration between two languages. For example, the language definition module for *SQLite*, an embedded database, defines an expression that runs an SQL expression against a database connection object. The module defines the *Katahdin* language expression shown in figure 3.3. This allows SQL code to be written in the same line as *Katahdin* code:

```
database = new Mono.Data.SqliteClient.SqliteConnection(···);

database.Open();

database?  insert into films values(
    "Indiana Jones and the Last Crusade",
    1989, "Steven Spielberg");

print database?  select director from films
    where title = "Indiana Jones and the Last Crusade"
```

So far, I have developed language definition modules for a subset of the *FORTRAN*, *Python* and *SQLite* languages. Although the features implemented by

Figure 3.3: The SqliteExpression

```
class SqliteExpression :  Expression {
    pattern {
        option recursive = false;
        database:Expression "?"
            statement:Sqlite.Statement
    }

    method Get() {
        // Evaluate the operands

        database = this.database.Get...();
        sql = this.statement.GetSql...();

        // Execute the command

        command = database.CreateCommand();
        command.CommandText = sql;

        reader = command.ExecuteReader();

        // Read each row into a list

        rows = [];

        while (reader.Read()) {
            row = [];

            for (n = 0; n < reader.FieldCount; n++)
                row.Add(reader.GetValue(n));

            rows.Add(row);
        }

        return rows;
    }
}

precedence SqliteExpression > CallExpression;
```

these modules are minimal, they are written without hacks and show that a full language could be defined. Writing the modules helped me work out a lot of the problems in my design, as I was seeing how it would actually be used.

For example, the off-side rule in *Python* (scope set by the indentation, as described in chapter 2) required parsing the white-space at the start of a line, instead of ignoring it as the *Katahdin* language does. If I had simplified the parsing problem by always ignoring white-space, this would not have been possible.

FORTRAN was an interesting language to implement because all subprograms are defined as *call-by-reference*, passing pointers to the variables that you pass as parameters instead of their values. This was totally different behaviour to the language I was using to write the module, *Katahdin*, but I was able to model the behaviour successfully. I was also able to make *FORTRAN's* calling convention work well with other languages. Figure 1.2 in chapter 1 shows a *Python* program calling a *FORTRAN* subprogram, passing *Python* parameters in as normal, where they are turned into references by the called *FORTRAN* subprogram. Both the *Python* and *FORTRAN* programmers write their programs as they expect for their languages and *Katahdin* handles the interoperability.

3.10 Testing

Early on in development *Katahdin* suffered from regressions, where the parser worked for a particular input, but then I moved onto developing a different area and made changes to the parser behaviour that broke it for the previous input. As by then I was running with different input, I didn't notice the error.

To combat this I set up a permanent library of unit-test routines, grammars and source text inputs that I ran automatically before each version control submission. As a result, I rarely had broken code in version control and I could modify the parser freely, knowing that I could very quickly verify that what I had done didn't break anything.

I also used the test library as a performance metric while experimenting with optimisation techniques.

CHAPTER 4

CURRENT STATUS AND FUTURE PLANS

4.1 State of Implementation

The *Katahdin* concept has been proven and the implementation is mature and stable. The current *Katahdin* code base will provide a solid base on which to solve the remaining problems to make *Katahdin* ready for production use.

4.2 Unsolved Problems

4.2.1 Memory Consumption

Memory consumption is high in *Katahdin*, consuming up to 50 MB of private address space when running my demo programs. I am not too concerned about memory consumption as the design already makes the conscious decision to use the packrat algorithm that uses a lot of memory in a trade-off for speed, and this trade-off can be tweaked to reduce the memory used.

Memory consumption would be a problem if I wanted to run *Katahdin* on smaller mobile or embedded devices. The *Mono* platform runs well on many such devices, so if memory consumption could be reduced, it would be possible to use *Katahdin* as a runtime for the mobile and embedded devices that *Java* already targets. This would be an excellent business opportunity, as it would allow developers to use existing languages on mobile and embedded devices, without having to port each of their runtimes.

Many of the speed optimisation issues discussed below also affect memory consumption.

4.2.2 Speed

The speed of the *Katahdin* interpreter is the biggest problem. Currently the interpreter takes about fifteen seconds to parse the standard library at start-up, where most interpreters start instantly. Although this may not seem a lot, it makes *Katahdin* unsuitable for applications where a program may be run many times for little jobs, as in shell scripting or traditional CGI web services.

However, the optimisation strategies that have been tried so far, such as just-in-time compiling the parser, have been very successful and I am confident that performance could be drastically improved if development was focused on this goal. Techniques such as adaptive optimisation have already been described, but time would also be well spent tidying up the existing implementation to look for obvious optimisation opportunities.

Other people have examined the speed performance of packrat parsers applied to PEGs and found them acceptable. The *Rats!* implementation has been found to outperform some traditional *LR(k)* parser generators[13].

4.2.3 Error Handling

Error handling is an issue that I have not touched on at all. A source text will either parse according to a PEG or it will fail, and if it fails it is very hard to work out what the problem was. Ideally, a language implementation that detects an error should report the position, a description of the error, a guess at what caused it and a suggestion of what to do.

Traditional grammars can do this because they use look-ahead to decide which route to take, and then commit to that route. If a rule fails it is an error. In a PEG, failure is a normal part of operation because the decision of which route to take is made based on finding out which doesn't fail. If there was an error with a missing semicolon at the end of a statement in a function in a traditional runtime, the compiler would know that it was parsing a statement and that there should be a semicolon and so would report an error. In the same situation a PEG parser would see that there wasn't a semicolon and so the statement rule would fail, the function rule would also fail because it could not pass a statement, and finally the root rule would fail and the parser would report that source text failed.

Although *Katahdin* has the ability the report the position of the furthest character that was parsed, and what the parser was trying to match at that point, serious error handling would need further research and concentrated development to implement, and is a critical feature that needs to be in place before *Katahdin* could be used in production.

4.3 Future Development

4.3.1 Debugger

To be useable for serious development *Katahdin* needs a custom debugger. This could be built from the internal debugger that I described in 3.6 but would still be a substantial development project. At present there is no way to step through *Katahdin* statements, set breakpoints or examine the stack. These are all more complicated operations in *Katahdin* than other language

implementations because there is no line between the programmer's code and the implementation of the language they are using. For example, a raw stack-trace would show all the method calls that implement language constructs as low level as for-statements, which the programmer might or might not care about. The stack-trace of a program written in a language defined in terms of another language that was then defined in the *Katahdin* base language would be unreadable, as each language construct used would result in several calls on the stack.

4.3.2 Language Modules

Ideally, *Katahdin* needs a large library of well written language definition modules that interoperate well. My business model is to provide general consulting services, so over time the library would grow as we work with clients using more languages, but a minimal standard library of languages would be needed to start promoting *Katahdin*.

The language definitions of *Python* and *FORTRAN* should be developed until they can run off the shelf applications. Test suites are available for both of these languages that could be used to verify our language definitions. I would then move on to develop language definition modules for languages widely used in industry such as *Java* and *Visual Basic*.

As *Katahdin* will be released as open source, it is hoped that a community of developers can be created to develop and share new language constructs and language definitions.

4.4 Conclusion

Katahdin has been a successful and rewarding project that has taken recent and active research from several people and applied it in a new direction to create a unique software development tool. The original aim of creating a programming language where the syntax and semantics are mutable at runtime has been achieved. With a strong theoretical background and a mature, stable and easily demonstrated implementation, *Katahdin* could be ready for use in production systems with a year's further development.

Ford's opinion is that the "expressiveness of PEGs ... introduces new syntax design choices for future languages"[11] and perhaps the next generation of languages could be developed as language modules on top of *Katahdin*.

BIBLIOGRAPHY

[1] `http://www.southern-storm.com.au/libjit.html`. Accessed May 2007.

[2] `http://www.mono-project.com/`. Accessed May 2007.

[3] `http://bazaar-vcs.org/`. Accessed May 2007.

[4] Ravi Sethi Alfred V. Aho, Monica S. Lam and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Second edition, 2007.

[5] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[6] Alexander Birman. PhD thesis, 1970.

[7] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23, 1973.

[8] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, IT2:113–124, 1956.

[9] Bryan Ford. Master's thesis, 2002.

[10] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. *Symposium on Principles of Programming Languages*, 2002.

[11] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *Symposium on Principles of Programming Languages*, 2004.

[12] Masaharu Goto. The cint c/c++ interpreter. `http://root.cern.ch/root/Cint.html`. Accessed May 2007.

[13] Robert Grimm. Better extensibility through modular syntax. *Programming Language Design and Implementation*, 2006.

[14] Stephen C. Johnson. *Unix Programmer's Manual*, 1979. Yacc: Yet Another Compiler-Compiler.

[15] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[16] Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[17] René Leermakers. Non-deterministic recursive ascent parsing. In *Proceedings of the fifth conference on European chapter of the Association for Computational Linguistics*, pages 63–68, Morristown, NJ, USA, 1991. Association for Computational Linguistics.

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

[19] Roman R. Redziejowski. Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Proceedings of the Concurrency Specification and Programming Workshop*, 2007.