Katahdin

Mutating a Programming Language's Syntax and Semantics at Runtime

Chris Seaton

The University of Bristol

chris@chrisseaton.com

Abstract

Katahdin is a programming language where the running program can modify the grammar that defines the syntax and semantics of the language. New constructs such as statements and expressions can be defined as easily as new types and functions and then used in the next line of the program. Programmers can extend the language with constructs that better express their programs without modifying the implementation of the interpreter. *Katahdin* is powerful enough that fundamental constructs such as for loops and array subscripting are implemented from scratch in the language itself. Existing languages can be implemented to either replace the grammar to turn *Katahdin* into an interpreter for that language, or to be composed with other grammars so that different parts of a program can be written in different languages. The development of *Katahdin* has addressed the problems of defining and parsing a modular grammar and has produced a working interpreter with which the advantages of being able to modify the grammar at runtime are easily demonstrated.

Keywords extensible grammar, modular grammar, packrat parsing, parsing expression grammar

1. Introduction

In most compilers and interpreters the grammar is predefined by the language's developer. Parsers are written from a grammar, sometimes by hand, sometimes with a parser generator[8] that produces parser program from the grammar automatically. The compiler or interpreter uses the parser to create a tree representation of the source code. The semantics of the language are usually informally defined and their implementation is scattered across the source code of the compiler or interpreter as methods that walk the parse tree.

To modify the definition of the language a programmer would need access to the implementation source code and knowledge of how it is designed. To add a new construct to the language, such as a statement or expression, they would have to modify the parser as well as write new code to express the semantics, probably in many different parts of the implementation. If a programmer took the time to do this they would still have the problem of distributing their modifications. Patches against an official release would have to be applied by other developers unless they were merged upstream.

There are many practical reasons why a programmer would want to modify the language that they use: to add constructs that better express their algorithms, to add constructs other languages have that they see as useful, and to reduce boilerplate code and repetitive patterns. Solutions such as *C* pre-processor macros and their prevalent use and abuse shows us that programmers need to be able to customise their languages. Languages without such a feature, such as *Java*, need a whole release cycle to elapse to add new language constructs. With a language that they can very easily modify, programmers can experiment with the language themselves. If they develop a useful construct they can distribute it as a module for others to import.

In *Katahdin* programmers can define new rules to add to the grammar, with methods that perform the semantic action of the rule. New rules are defined with a statement which can be part of a module, an if-statement, a method or anywhere else a statement can appear. This flexibility means that a programmer can store their constructs in modules, conditionally define them and generally control how the language is modified with control statements.

Katahdin can be seen as a generic language implementation. As easily as new statements and expressions can be defined to slot into the existing language, programmers can create a new grammar from scratch. Existing languages can be re-implemented in terms of *Katahdin* and either used on their own or a construct can be defined that lets programmers write parts of their programs in different languages.

This power is achieved with recent advancements in grammar theory and parsing technology and by taking the approach that time and space can be used much more lib-

erally than in most language implementations. In *Katahdin* the grammar is a mutable data structure and instead of writing a parser program that corresponds to the grammar, either by hand or with a parser generator tool, the data structure is walked node-by-node to parse the input source. Optimisations that trade off space with time are used to restore practical program run time.

2. Rationale

Katahdin is a programming language instead of a parser generator. The language is imperative, object-oriented, duck-typed and curly-braced. Programmers familiar with *C*, *Java*, *C♯*, *Python* or almost any popular imperative language should have no trouble using *Katahdin*.

New language constructs are defined as a special case of defining a new class. The semantics for the construct are implemented by defining methods in the class, using existing language constructs. This class is the complete syntactical and semantic definition for a new exponentiation operator:

```
class Pow : Expression {
  pattern {
    a:Expression '^' b:Expression
  }
  method Get() {
    return System.Math.Pow(@a.Get(),
      @b.Get());
  }
}
```

The `Get()` method is defined for all expressions. To implement the semantics of the exponentiation operator we call `Get()` for both of the operands and then call the library method `System.Math.Pow()`. The methods each construct has depends on convention. Expressions have `Get()` and `Set()` methods. If the `Set()` method is left out the expression cannot be assigned to: the default `Set()` implementation in Expression will throw an exception. Other constructs have appropriate methods, for example statements all have a `Run()` method. This class shows the `Run()` method being overloaded to implement a for-loop-statement:

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

The `body.Run...();` notation performs a call in the calling method's scope. This is so that references to variables in the for loop's body are resolved in the scope of the method which contains the for loop, and not in the scope of the method `ForLoop.Run()`.

The majority of the *Katahdin* programming language is implemented at runtime in this way. Language constructs such as loops, lists and dictionaries are defined in the standard library which is imported as the interpreter starts.

Programmers using *Katahdin* can continue to extend the language beyond the standard library with new constructs to fit into the *Katahdin* language, or define a new language from scratch. If there are multiple grammars defined, they can be composed so that multiple languages can be used in a single file. This class defines a new *Katahdin* Statement which can contain any *Python* statement:

```
import ''python.kat'';

class PythonStatement : Statement {
  pattern {
    ''python'' ''{''
      statement:Python.Statement ''}''
  }
  method Run() {
    statement.Run();
  }
}
```

After defining this class the programmer would be able to use a *Python* statement anywhere a statement is allowed by enclosing it in a *Python* block.

```
for (int i = 0; i < 10; i++) {
  python {
    for j in range(0, 10):
      print i * j
  }
}
```

3. Modular, Extensible Syntax Definition

New language constructs can be defined as part of the application but usually they will be collected into modules. When a module is loaded the language definition is extended with the constructs defined in it. A programmer may import unrelated modules written by different programmers that never intended their modules to be used together. This means that each module of new language constructs has to be defined independently and able to be merged with any existing syntax.

3.1 Background on PEGs

In *Katahdin*, programmers express syntax using a parsing expression grammar[6], with annotations to more elegantly express common patterns. Parsing expression grammars (PEGs) are a recognition based grammar, opposed to the generational grammars normally used to define the syntax of a language, such as the regular and context-free families of grammars. PEGs are unambiguous and can always be parsed in linear time which is important if we are to let the programmer arbitrarily modify the grammar.

PEGs can be used to define the lexicon of the language as well as the syntax, so a single grammar can be used for both. Most language implementations use two separate languages to define these, which increases the complexity and fragments the language definition.

Writing a PEG is very similar to writing a RE or CFG. Literal text is quoted, repetitions are expressed with the ? * + operators, alternatives with the | operator. There are special literals used to match the end of the text.

The behaviour of the | operator is different in PEGs to other grammars. Instead of matching any of the alternatives it matches the first one that it can, trying the alternatives in order. PEGs also have two semantic predicates, the & 'and' operator and the ! 'not' operator. & succeeds if it can match the sub pattern, ! succeeds if it cannot. Neither of the two predicates consumes any of the text that they match.

A PEG can express all *LL(k)* and deterministic *LR(k)* grammars and many others[6]. However there are some patterns that need fairly complex PEG rules to express. Unfortunately these include common patterns found in almost every language such as recursion, ambiguity and precedence. The complex PEG rules are difficult to maintain and make modular definition of the grammar hard, as will be shown later. *Katahdin* uses a modified PEG with an annotation operator to enable the grammar writer to better express these patterns. Annotations are applied to blocks, which are marked by curly braces, using the option keyword:

```
rule ← {option NAME = VALUE; ...}
rule ← {option NAME; ...}
```

The annotations control how the parser applies the rule, and are used to express the common grammar patterns outlined in the next sections.

3.2 Recursion and Ambiguity

A common problem when defining the grammar of a language is expressing rules that recurse and solving the ambiguity that this introduces. Mathematical operators, present in almost all programming languages, can have another instance of themselves as an operand. For example the subtraction operator:

```
a - b - c
```

With a simple definition of the subtraction operator, this expression could be interpreted in two ways, as either ((a - b) - c) or (a - (b - c)). In mathematics there is a convention of using the first interpretation to solve the ambiguity. This convention is referred to as the associativity of the operator and is implemented by controlling how rules in the grammar recurse.

The subtraction operator is normally defined in languages as left-associative. That means that the operator can appear more than once in expressions, and that the operators should bind tighter towards the left side of the expression. This can be shown by making the implied order of evaluation explicit using parentheses:

```
a - b - c - d
(((a - b) - c) - d)
```

The most natural way to express the subtraction operator is with this expression:

```
exp ← exp '-' exp
```

This defines the subtraction operator to be an expression, followed by the text '-', and then another expression. This is how the programmer thinks of the operator as they use it, but the definition does not work in a PEG. A PEG parser would go to match exp, which instructs it to match exp, and so on. The parser loops trying to match exp without ever consuming any text. The rule is left recursive. That's actually what was wanted as the subtraction operator is indeed left recursive in that the left hand side can be another subtraction expression.

A common solution to the problem is to express left recursion as repetition or by introducing an extra grammar rule. These solutions require the grammar writer to express a parsing algorithm for the rule instead of naturally expressing the syntax:

```
exp ← number ('-' number)+
```

The extra rule, recommended by Ford for use in PEGs[4, page 40], is a suffix pattern:

```
exp ← number addSuffix
addSuffix ← '-' number addSuffix
addSuffix ← ε
```

My solution is to use annotation. The operator is expressed naturally, and annotated as left recursive. The parser can use this annotation to parse the rule left recursively, without infinite recursion, as is shown later. The annotation is an option applied to the obvious rule:

```
exp ← {option leftRecursive; exp '-' exp}
```

Other operators are right-associative, such as the exponentiation operator, and so require right recursion. The parentheses operators are recursive, but do not need to be marked specifically as left or right because the definition starts with a '(' character and so does not immediately recurse:

```
exp ← {option rightRecursive; exp '^' exp}
exp ← {option recursive; '(' exp ')'}
```

In *Katahdin*, the default is to be simply recursive if no other option is expressed. It can be negated to make rules non recursive to express non-associative operators.

3.3 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 a PEG the grammar writer is required to express operator precedence by writing a parsing algorithm. In the common solution show below, an expression is a term followed by any number of expression suffixes, which are an application of an addition or subtraction operator and then another term. A term is a factor followed by any number of term suffixes, which are the multiplication and division operators. A factor is a number. These rules are expressed as follows:

```
exp ← term expSuffix*
expSuffix ← '+' term
expSuffix ← '-' term

term ← factor termSuffix*
termSuffix ← '*' factor
termSuffix ← '/' factor

factor ← number
```

These rules have totally fractured the programer's view of the operators. Adding a new operator with the same precedence as addition requires understanding the algorithm that the original grammar writer used and defining a new expSuffix. The parser should be dealing with these problems, where currently with PEGs the grammar writer is having to write parsing algorithms.

Katahdin uses annotations to solve the problem of operator precedence. Operators are defined as before, annotated with their associativity. The precedence is then set with another annotation:

```
exp ← add
add ← {option leftRecursive; exp '+' exp}
add ← {option leftRecursive; exp '-' exp}
exp ← mul
mul ← {option leftRecursive; exp '*' exp}
mul ← {option leftRecursive; exp '/' exp}

precedence mul > add
```

The precedence-statement establishes a relation between two rules. There the precedence of multiplication is set to be higher than that of add.

Each operator is now defined with a single annotated rule, and an annotation to express the relationship between them. This much better matches the programmer's view of operators, and makes it easier to add new operators. For example, to add a new modulo operator we define just two new self-contained rules:

```
mod ← {option leftRecursive; exp '%' exp}
precedence mod = mul
```

3.4 White Space

White space is the non-printing characters such as spaces and new lines used along with punctuation to delimit tokens. The text 'foobar' is a single token unless white space or punctuation is used to separate the characters; 'foo bar', 'foo.bar'. Programmers also use white space to format their code for better human readability, and some languages, such as *Python* and *Haskell*, 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:

```
exp ← exp whitespace? '+' whitespace? exp
```

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!*[7] 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:

```
identifier ← {option whitespace = null;
  ('a'..'z')+}
```

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:

```
ws ←' ' |'\t' |'\n' |'\r' '\n'?
program ← {option whitespace = ws;
  statement*}
```

3.5 Parse Trees

The grammar described so far can be used to verify the conformance of a text. We want the parser to output a data structure that represents the syntax applied to the text. The most simple form of parse tree is the concrete syntax tree. Each rule that is matched forms a new sub-tree, with branches being the tokens or rules matched within the rule. The root node of the sub-tree is the name of the rule matched. The following shows an example text and the parse tree produced:

```
'a + b * c'
(add 'a' '+' (mul 'b' '*' 'c'))
```

More useful is the abstract syntax tree of the text. In this tree, only meaningful tokens are added to the tree. The '+' and '*' tokens are not meaningful because we already have the name of the rules matched. Now the add and mul nodes both have just two children: the two operators. This kind of parse tree better fits the way programmers conceive a grammar:

```
'a + b * c'
(add 'a' (mul 'b' 'c'))
```

A parser cannot always determine what should be included in the abstract tree and what should not, so generating an AST requires the grammar writer to annotate nodes. The *Antlr* parser generator produces a concrete syntax tree by default, but allows the grammar writer to annotate with a caret tokens that they want to use as the root node for a new sub-tree in the AST:

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

In the above rule, which follows the pattern for left recursion explained earlier, each time the ADD token is matched a new sub-tree is created, rooted with the ADD node:

```
'a + b + c'
(ADD (ADD 'a' 'b') 'c')
```

This method was used for *Antlr* because in its grammar each rule might be matching more than one syntactic construct. The exp rule above matches any number of add operations. In *Katahdin* each rule matches a self contained syntactic construct, so each rule produces a single sub-tree rooted with the name of the rule matched. In the implementation, each rule is attached to a class definition, which is instantiated when the rule is matched:

```
class Add {
  pattern {
    Expression '+' Expression
  }
}
```

Each time Add is matched, an Add object is instantiated.

3.5.1 Fields

Each node in the *Katahdin* grammar returns an object when parsed. Text nodes return the text matched as a string object, repetitions and sequences return a list of the objects returned by the match. Using the ':' label operator the grammar writer can assign these results to a field in the object instantiated for the match.

```
class Add {
  pattern {
    a:Expression '+' b:Expression
  }
}
```

When the Add rule is matched and the Add class is instantiated, the results of parsing the two references to Expression is assigned to the fields a and b in the object. We don't need to store the text '+' as the class of the object tells us the type of expression we matched. As '+' is not labelled it is discarded. Text nodes are never matched by default as they are rarely needed, but this can be overridden with an option annotation. If a repetition or sequence is labelled it assigns a list object containing all the matched nodes. If any of the matched nodes were lists, the items in the list are appended instead of the list object. For example, to match the parameters of a function call, delimited by commas:

```
'(' parameters:(Expression
  (',' Expression)*) ')'
```

The parameters field will be set to a list of the Expression objects instantiated for each parameter.

3.5.2 Tokens

As we are matching lexemes with the grammar, we need rules to match identifiers, numbers and so on. With the scheme for building parse trees described above we can obtain a list of the characters in an identifier, or digits in a number, but grammar writers are more likely to expect

a single string when matching these constructs. For this, *Katahdin* provides the token operator. The token operator matches a sub-pattern and returns the text that was matched instead of the parse tree:

```
id ← text:token{('a'..'z')+}
```

4. Parsing

The PEG that the *Katahdin* grammar is based on can be parsed by a simple top down[2], recursive descent parser. Such a parser is required to backtrack[3] when parsing the alternative | operator. Each alternative fails or succeeds to match. If it fails, the parser returns to the point in the text where it started trying to match the alternative, and tries to match the next alternative. When one of the alternatives succeeds and matches, the alternative operator matches and the parser moves on. This parsing algorithm is valid but risks exponential run time. In the example below, to match a we try the alternatives b, c and d, one after the other. All of those rules start with a reference to the expression e, so in a worst case scenario we try to match e three times.

```
a ← b | c | d
b ← e f
c ← e g
d ← e h
```

The parsing of e on the second and third times is redundant. We already worked out whether or not the rule matched at that point in the text, and had the resulting parse tree. A time-space trade off is therefore to store the result of parsing e at that location, and when we try to parse it again, retrieving that cached result. This is known as packrat parsing[5].

The store is a map of tuples of the rule being matched and the current parser state, to the resulting parse tree or the null value if no parse tree matched. The parser state is a tuple of variables that effect how the parser could match a rule. For example, the white space rule, the current level of precedence being matched, and most importantly the position in the text. Before every rule is matched, the store is checked to see if the rule was previously applied when the parser was in this state. If it was, the value from the store is returned, otherwise the rule is applied as normal and the result stored. This is equivalent to factoring out the common rule e.

```
a ← e (b | c | d)
b ← f
c ← g
d ← h
```

The *Katahdin* parser has two important extensions to packrat parsing to support the addition of recursion and precedence annotations to the *Katahdin* PEG, described in the following sections.

4.1 Precedence

Precedence is set in the grammar with an annotation of the form:

```
precedence A = B
precedence A < B
precedence A < B
```

Each precedence-statement establishes a relationship between two rules. Multiple statements build up an order between many rules.

The parser maintains a current precedence. When entering a rule the precedence is raised to that of the rule if it is higher. When the rule is left the precedence is lowered back to where it was previously.

Rules fail if their precedence is not higher than the current precedence. This prevents the operands of a multiplication operator matching an addition expression. Some expressions, such as parentheses, can contain expressions of any precedence. This is indicated with the dropPrecedence option:

```
exp ← paren
paren ← '(' {option dropPrecedence; exp} ')'
```

The parser resets the current precedence to the lowest value when it enters a block with the dropPrecedence option.

The semantics of the alt operator is modified from that of a standard PEG to support the precedence annotations. The alternatives are sorted into groups by precedence. When matching the alt, the groups are tried in order of increasing precedence. Within each group all the alternatives are tried and the longest successful match returned. After an alternative has been matched, no more groups are tried. Rules with no precedence set are considered to have the highest precedence, and so are tried last. Within each precedence group in an alt, all alternatives are tried and the longest match is returned.

For example, add, sub, mul and div expressions are defined as usual in two groups of increasing precedence (add sub) (mul div). When parsing an expression the parser would first try to parse both add and sub. The longest successful match would be returned. If neither matched, mul and div would tried in the same way.

Normally in a PEG the first match is taken but in *Katahdin* this only applies within each precedence group. *Katahdin* was designed like this because in a normal PEG the grammar writer has control over what rules are defined and in what order. In *Katahdin* the grammar is modular. Someone writing a module's grammar does not know what their grammar will be composed with and cannot control where their

rules come in lists of alternatives that other modules have modified. The rule that the longest alternative is taken is easy to understand for programmers and users, and better allows modular definition of the grammar.

4.2 Recursion

The parser supports four kinds of recursion, set with the recursive, leftRecursive and rightRecursive options. Rules are simply recursive by default. The recursive option can be negated to make a rule non-recursive, or set leftRecursive or rightRecursive to make a rule left or right recursive.

4.2.1 Simply Recursive Rules

Simple recursion is the behaviour of the algorithm described so far. Simply recursive rules can be recursed into by the parser from any reference in its definition.

4.2.2 Left Recursive Rules

Left recursive rules are parsed by disallowing recursion totally, and then reapplying the rule as many times as possible, using what was already matched as the left hand side for each new application.

```
exp ← number
exp ← add
add ← {option leftRecursive; a:exp '+' b:exp}
```

When parsing add, the parser will not recurse when matching a or b, but after a successful match it will apply add again, using what was just matched for a and starting at '+'.

In the general case, when reapplying the rule the parser will make any reference to the recursive rule succeed, yielding the previously matched tree.

```
1 + 2 + 3
```

When add is applied to the above text, recursion is disallowed so a and b match the numbers 1 and 2 respectively.

```
(1 + 2)
```

As add is annotated as being left recursive, the rule is then applied again, passing that (1 + 2) should be yielded for the left hand side and starting at the second '+'.

```
((1 + 2) + 3)
```

The parser tries to apply the rule again for a third time, but in this case fails when '+' fails to match. The previous successful tree is yielded from the original application of add.

This effectively implements the following pattern:

```
add ← number ('+' number)*
```

When left recursing an add rule we want to try to match sub as well. In the example below, (1 + 2) would be matched by add, and then the add rule would left recurse to match a sub rule:

```
((1 + 2) - 3)
```

This is implemented by left recursing into any rule of the same precedence in the alt that lead us to match the first rule. The set of rules to left recurse into is set by the alt operator from the current precedence group, and contains all left recursive rules.

4.2.3 Right Recursive Rules

Right recursive rules are parsed by adding the constraint that the parser will only recurse back into the rule on the right hand side of the expression. The right hand side is taken to be everything after the first node that is matched.

```
exp ← number
exp ← pow
pow ← {option rightRecursive; a:exp '^' b:exp}
```

When parsing pow, recursion is disallowed for the left hand side, a, just as if the rule were non-recursive. Recursion is allowed for the rest of the rule, including b. Normally rules fail if their precedence is not higher than the current precedence. On the right hand side of a right recursive rule, as well as allowing recursion, rules are allowed to match if their precedence is equal to the current precedence.

```
1 ^ 2 ^ 3
```

When pow is applied to the above text the left hand side is not allowed to recurse and so can only match a number, 1. The right hand side is allowed to recurse and so matches another pow expression, (2 ^ 3).

```
(1 ^ (2 ^ 3))
```

This effectively implements the following pattern:

```
pow ← number ('^' pow)?
```

4.2.4 Non-Recursive Rules

Non-recursive rules are implemented by totally disallowing recursion, as while parsing the left hand side of a right recursive rule.

4.3 Implementation

Katahdin is implemented as an interpreter, written in the *C♯* programming language using Novell's implementation of the .NET framework, Mono. *C♯* and Mono were chosen over

Java and other possible platforms because the class library includes efficient support for emitting code at runtime.

The interpreter program supports a minimal base language at start up. The base language includes class statements and all of the *Katahdin* PEG operators. The only language statements and expressions that are included in the base are those that are best compiled to specific bytecode instructions or cannot conveniently be expressed in terms of other constructs. For example function and class statements, arithmetic operators, method calling, exception handling and so on are all implemented in base. Array subscripting is implemented as a method call so is defined in the standard library in terms of the base method call operator.

The rest of the language is defined in the standard library which is imported before the user's program is run. The standard library builds higher level language constructs from lower level constructs. For example, the for-loop-statement is defined in terms of the while loop, and then the for-each-loop-statement in terms of the for-loop-statement.

4.4 Optimisations

Instead of the parser walking each node in the grammar, the nodes in a rule can emit code that is compiled into single method. The .NET runtime allows a program to emit bytecode instructions that it can just-in-time compile (JIT) to machine instructions to be run by the processor. The code that each node emits depends on its parameters. For example, if a text node is used where white-space has been disabled, no white-space handling code will be emitted.

This makes *Katahdin* similar to parser generators in that it is generating parsing methods from the grammar, but *Katahdin* directly generates bytecode at runtime and does not have to invoke a compiler. When the grammar changes, *Katahdin* discards the old parse methods and compiles the new grammar in its place.

There is scope to employ adaptive optimisation[1], as the *Java HotSpot* virtual machine does, instrumenting the parse methods and recompiling with optimisations based on how the methods are being used.

These techniques could also be applied to the code intermediate tree data structure, which currently is interpreted node-by-node.

5. Applications

There are many reasons why a programmer would want to add new language constructs at runtime. The *Katahdin* standard library creates many of the standard language features in *Katahdin* code, where most language implementations would have custom code in the interpreter or compiler.

An assert-statement checks a condition and throws an exception if it fails. They are used for debugging so the programmer will want to know where in the source code an assertion has failed, and may want to disable them entirely when using the software in production.

In the *C* programming language to implement assert we need to use the pre-processor and the special hook macros `__FILE__` and `__LINE__`:

```
#ifdef NDEBUG
  #define assert(cond)
#else
  #define assert(cond) \
    printf(``Error: file %s, line %d'', \
      __FILE__, __LINE__)
#endif
```

The *Java* programming language has no pre-processor and so no way to conditionally compile code, and a method call cannot be used as there would be no way to disable the calls. To implement assert for version 1.4, Sun had to modify the source of *Java* compiler to add a new statement and make a new release. To be able to use the assert-statement all users of *Java* had to update their installations. In *Katahdin*, assert is implemented as part of the standard library in the *Katahdin* language itself, without any special support from the runtime. If the standard library didn't include assert, programmers would be free to implement it themselves without modifying the runtime. Unlike the *C* implementation, the *Katahdin* assert is a statement like any other:

```
class AssertStatement : Statement {
  pattern {
    'assert' conditionSource:token{
      condition:Expression} ';'
  }
  method Run() {
    if (!debug)
      return;
    if (!@condition.Get...() as System.Bool)
      throw @conditionSource.Trim()
        + ' failed '
        + @condition.Source.ToString();
  }
}

assert n <= 100;
```

5.1 Inline Sql

Performing Sql queries to a database through a client library API is a very common operation. In most languages programmers pass the Sql as a string to a client library.

The *Katahdin* solution is to define a grammar for the flavour of Sql that the database uses. Then define an SqlExpression which references the Sql grammar. Instead of each of the `Get()` methods in the Sql grammar interpreting the tree, they return strings which are built up to form a full query which can be passed to the existing client library.

The syntax of the Sql is checked at compile time along with the program code and injection attacks are excluded because query building, string quoting and character escaping are handled by the semantic actions of the grammar rules.

```
import ‘‘sql.kat’’;
class SqlExpression : Expression {
  pattern {
    option recursive = false;
    database:Expression ‘?’
      statement:Sql.Statement
  }
  method Get() {
    // Evaluate the operands
    database = @database.Get...();
    sql = @statement.Get...();
    // 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;
  }
}

children = contactDatabase? select name
  from contacts where age < 18;
```

5.2 Multiple Programming Languages

In the previous example it was made possible to use both *Katahdin* and Sql in the same file. Sql is a simple language used in a very specific way, but the same technique can be used to define the grammar of a general purpose language and allow it to be used from within *Katahdin*.

If the grammar of *Python* is defined in a module python.kat, we can import it and define a new *Katahdin* statement that allows the programmer to use *Python* statements:

```
import ‘‘python.kat’’;

class PythonStatement : Statement {
  pattern {
    ‘‘python’’ ‘‘{’’
      statement:Python.Statement ‘‘}’’
  }
  method Run() {
    statement.Run();
  }
}

for (int i = 0; i < 10; i++) {
  python {
    for j in range(0, 10):
      print i * j
  }
}
```

The using-statement changes the root node in the grammar and makes *Katahdin* parse the rest of the program as a different language:

```
import ‘‘python.kat’’;
using Python.Program;

def factorial(n):
  if n == 0:
    return 0
  else:
    return n * factorial(n - 1)
```

In this way, *Katahdin* becomes a generic language implementation, taking both a language definition and program source text as input.

With *Katahdin* different languages can be used in the same file, method, statement, or even expression. There is nothing special about the *Katahdin* language constructs compared to any that the programmer defines, so all languages are on the same level, evaluated in the same pass.

It is not expected that developers will want to write each line of their program in a different language – there are more practical applications. For example, a research group might have a standard pseudo random number generator procedure written in *FORTRAN*. They want to use the same procedure for all their applications to maintain result consistency. If a programmer wanted to write a new application in *Python* they would normally have to translate the procedure, carefully checking that the semantics were exactly the same. Apart from the time taken to translate, there is now the problem of maintaining twice the code. In *Katahdin*, the programmer could write his program in straight *Python* and have just the pseudo random number generator procedure written in *FORTRAN*. The *FORTRAN* procedure could be in the same file as the *Python* code, with no need for an external library or binding layer.

6. Conclusions

Katahdin opens up the exciting possibility of programmers routinely customising the languages that they use. A community of developers could create a centralised library of language constructs from which programmers would pick and choose to create the best language to express their ap-

plication. New ideas for language development could be experimented with more rapidly than is currently possible.

Libraries written in one language could be used without bindings by any other language. Legacy applications written in little-used languages could be modified line-by-line in whatever language you currently have developers trained in. With many language definitions, *Katahdin* could become a single runtime for running any language. It would be feasible to port the runtime to different platforms such as *Java* and native *C* so that any language could be run on any platform.

Katahdin uses traditional language development techniques but moves them to the runtime and allows them to use mutable data structures. Traditional parser generators take a grammar and produce a program to parse it, *Katahdin* also does this but at runtime, emitting code just-in-time to parse the rules and recompiling the parser if the grammar data structure changes.

Although *Katahdin* is fast enough to be useable (parsing the standard library takes a few seconds on a modest workstation) the performance has not been analysed in any detail. Also unknown are the problems that will surely surface when programmers compose unrelated grammars that were written by different programmers and were never intended to work together. Checks for conflicting rules and ambiguity detection will be needed, but currently *Katahdin* accepts any grammar and will try to parse it.

Acknowledgments

I would like to thank my project supervisor, Dr Henk Muller, for his guidance through my project and his help in preparing this paper.

References

[1] 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.

[2] Alexander Birman. PhD thesis, 1970.

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

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

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

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

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

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