Practical Packrat Parsing

Robert Grimm
rgrimm@cs.nyu.edu

Technical Report TR2004-854
Dept. of Computer Science
New York University

ABSTRACT

A considerable number of research projects are exploring how to extend object-oriented programming languages such as Java with, for example, support for generics, multiple dispatch, or pattern matching. To keep up with these changes, language implementors need appropriate tools. In this context, easily extensible parser generators are especially important because parsing program sources is a necessary first step for any language processor, be it a compiler, syntax-highlighting editor, or API documentation generator. Unfortunately, context-free grammars and the corresponding LR or LL parsers, while well understood and widely used, are also unnecessarily hard to extend. To address this lack of appropriate tools, we introduce *Rats!*, a parser generator for Java that supports easily modifiable grammars and avoids the complexities associated with altering LR or LL grammars. Our work builds on recent research on packrat parsers, which are recursive descent parsers that perform backtracking but also memoize all intermediate results (hence their name), thus ensuring linear-time performance. Our work makes this parsing technique, which has been developed in the context of functional programming languages, practical for object-oriented languages. Furthermore, our parser generator supports simpler grammar specifications and more convenient error reporting, while also producing better performing parsers through aggressive optimizations. In this paper, we motivate the need for more easily extensible parsers, describe our parser generator and its optimizations in detail, and present the results of our experimental evaluation.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization, parsing, translator writing systems and compiler generators*

General Terms

Design, languages, performance

Keywords

Parser generators, parsing expression grammars, packrat parsers, extensible programming languages

1. INTRODUCTION

A considerable number of research projects have been exploring language extensions that improve the expressivity of object-oriented programming languages, with many projects focusing on Java as a base language. Examples include generics [9, 10, 19, 27, 35], aspect-oriented programming [20, 21], multiple dispatch [11, 25], pattern matching [22, 24], and support for controlling information flow [26]. Other projects have been exploring extensibility mechanisms for object-oriented languages, focusing either on macro systems [2, 3] or on the compiler itself [28].

Taken together, these efforts illustrate that programming languages are not static entities, but rather in constant flux. In fact, Sun has incorporated generics, among several other new features, into the upcoming 1.5 release of Java [8, 17]. An important challenge, then, is how to gracefully evolve programming languages [32] and, more specifically, how to provide language implementors with the appropriate tools for tracking ever changing languages. While our larger research agenda aims to explore how to express, compose, and efficiently implement language extensions for C-like languages, for the purposes of this paper we focus on the extensibility of programming language grammars and their parsers. After all, parsing program sources is a necessary first step for *any* language processor, be it a compiler, interpreter, syntax-highlighting editor, API documentation generator, or source measurement tool.

Unfortunately, context-free grammars (CFGs) and the corresponding LR or LL parsers [1], while well understood and widely used, offer only limited extensibility [7, 15] and thus represent an unsuitable foundation for managing the evolution of programming languages and their implementations. On the other hand, parsing expression grammars [4, 5, 15] (PEGs) and packrat parsers [13, 14] provide an attractive alternative. While PEGs share many constructs with the familiar EBNF notation [18, 36], a key difference is that PEGs rely on *ordered* choices instead of the unordered choices used in CFGs. As a result, PEGs can avoid unnecessary ambiguities and are more easily modifiable. Additional flexibility is offered through *syntactic predicates*, which match expressions but do not consume the input, thus providing unlimited lookahead, and through the *integration of lexing with parsing*, which greatly simplifies the addition of new tokens to a grammar.

Parsing expression grammars can be implemented by so-

called *packrat parsers*. They are recursive descent parsers, which perform backtracking but also memoize all intermediate results (hence the name), thus ensuring linear-time performance. So far, Ford [13, 14] has implemented several handwritten packrat parsers as well as a packrat parser generator, called Pappy, for and in Haskell. As a lazy, functional programming language, Haskell certainly provides a convenient platform for implementing this memoizing parsing technique. However, the choice of target language (arguably) also limits the accessibility of packrat parsers.

To make packrat parsers more widely accessible, this paper introduces *Rats!*,[1] a packrat parser generator for Java. By leveraging Java's object-oriented features, parsers generated by *Rats!* have a simple and elegant implementation of memoization in a strict, imperative programming language. Compared to Pappy, Ford's packrat parser generator for Haskell, *Rats!* features more concise grammar specifications, has better support for debugging grammars and for reporting parser errors, and, through aggressive optimizations, generates better performing code. Our parser generator has been implemented within our own framework for building extensible source-to-source language processors and includes optional support for using the framework in generated parsers, thus simplifying the implementation of other language processors. *Rats!* has been released as open source and is available at `http://www.cs.nyu.edu/rgrimm/xtc/`.

The rest of this paper is organized as follows. In Section 2, we motivate our work and review other approaches to parsing programming languages. We follow with an overview of our parser generator in Section 3. We then discuss our object-oriented implementation of packrat parsers in Section 4 and describe the optimizations performed by *Rats!* in Section 5. In Section 6, we present the results of our experimental evaluation. In Section 7, we follow with a discussion of our framework for building extensible source-to-source language processors. Finally, we outline future work in Section 8 and conclude in Section 9.

2. MOTIVATION AND RELATED WORK

From the perspective of language extensibility, using a parser generator to create a parser has an important advantage over a handwritten parser: the grammar provides a concise specification of the corresponding language. As a result, we generally expect it to be easier to modify the machine-generated parser than the handwritten one. However, LALR(1) grammars for the popular Yacc tool [23] and similar parser generators are fairly brittle in the face of change. For example, Brabrand et al. [7] suggest adding the following, admittedly clumsy, extension to a Java grammar for synchronization on two objects:

```
GuardingStatement : SYNCHRONIZED
  '(' Expression Expression ')' Statement;
```

Yet, this simple modification results in 29 shift/reduce and 26 reduce/reduce conflicts. To make matters worse, none of the conflicts occur in parser states related to the new production, making it unnecessarily hard to correct the grammar.

LL(k) parser generators do not suffer from this brittleness, but, instead, need to disambiguate alternatives that share a common prefix. A grammar writer can avoid the need for disambiguation by factoring such prefixes by hand, but this requires extra effort and obfuscates the language specification. JavaCC [12] supports explicit lookahead expressions, but they still tend to obfuscate the language specification. In contrast, ANTLR [31] supports a global lookahead flag, but, in practice, still requires local options to fine-tune the lookahead. In either case, the need for explicit lookahead specifications complicates the grammar and makes it more difficult to modify.

An additional problem common to both LR and LL parser generators is the separation of lexing and parsing, which can make it unnecessarily hard to add new tokens to a grammar. As an example, consider adding support for character classes, such as "`[0-9a-fA-F]`" for hexadecimal digits, to the grammar of a parser generator. At the grammar-level, the corresponding parsing expression should look as follows (with '`/`' denoting the ordered choice operator):

```
'[' ( Char '-' Char / Char )+ ']'
```

However, because the definition of the `Char` token overlaps almost all other tokens, its addition results in a substantial number of ambiguity errors for the *lexical* specification. Common workarounds are the use of separate lexer states, as supported by Lex [23], or the composition of different lexers, as supported by ANTLR, both of which, again, obfuscate the language specification.

Packrat parsers can avoid these problems while still exhibiting linear-time performance. In particular, because they are recursive descent parsers, they avoid the brittleness of LR parsers. Next, because they backtrack while also memoizing all intermediate results, they do not require explicit management of lookahead. Unlimited lookahead is still available through syntactic predicates, though not to ensure linear-time performance but rather to increase expressiveness. For example, by using syntactic predicates, packrat parsers can recognize $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n > 0\}$, which cannot be expressed by CFGs. Finally, because packrat parsers effectively treat every character in the input as a token, they do not require separate lexical specifications and make the full power of parsing expression grammars available for recognizing lexical syntax.

However, this last property of packrat parsers also represents the biggest challenge to an efficient implementation: The data structure for memoizing intermediate results is a possibly very large table, with the characters in the input defining the horizontal dimension and the nonterminals in the grammar defining the vertical dimension. Keeping this table as small as possible, e.g., by only computing entries as necessary, is the key to efficient packrat parsing. In other words, the challenge of practical packrat parsing is to balance the ease of extensibility with reasonable parser performance, both in terms of memory utilization and parsing latency.

Several techniques used in parsing expression grammars and packrat parsers have also been explored in other parser generation systems. Notably, Parr and Quong have introduced the use of predicates for LL(k) parsing [30] and integrated them into ANTLR [31]. They were later adopted in JavaCC [33]. ANTLR also uses a recursive descent lexer, which provides many of the advantages of packrat parsers when recognizing lexical syntax, though it still separates lexing from parsing. Finally, the `metafront` system [7] builds on a new, linear-time parsing technique that also includes

[1]The name is pronounced with the conviction of a native New Yorker when faced with a troublesome obstacle.

Operator	Type	Prec.	Description
' '	Primary	5	Literal character
" "	Primary	5	Literal string
[ ]	Primary	5	Character class
.	Primary	5	Any character
{ }	Primary	5	Semantic action
(*e*)	Primary	5	Grouping
e?	Unary suffix	4	Option
*e**	Unary suffix	4	Zero-or-more
e+	Unary suffix	4	One-or-more
&*e*	Unary prefix	3	And-predicate
!*e*	Unary prefix	3	Not-predicate
id:*e*	Unary prefix	3	Binding
" ":*e*	Unary prefix	3	String match
e_1 e_2	Binary	2	Sequence
e_1 / e_2	Binary	1	Ordered choice

Table 1: The operators supported by *Rats!*. Note that "Prec." stands for precedence level.

a limited form of syntactic predicates. The advantage of packrat parsers is that they combine all these features into a simple and easily extensible framework with a well-defined formal foundation [15].

3. RATS!

Like most other parser generators, *Rats!* is implemented as a source-to-source transformer that translates a grammar specification into programming language source code. Our parser generator currently targets only Java, though all language-specific aspects have been carefully isolated in two classes to ease future ports to other object-oriented programming languages. A grammar specification starts with a header, which includes the Java package and class name of the corresponding parser, a declaration of top-level nonterminals, and optional code blocks to be included verbatim before, within, and after the parser class. The header is followed by one or more productions, which are of the form:

Type Nonterminal = *e* ;

The *Type* is the Java type of the semantic value, *Nonterminal* is the name of the nonterminal, and *e* is the expression to be parsed.

Table 1 summarizes the expression operators supported by *Rats!*. They directly mirror the operators of parsing expression grammars [15], with straight-forward extensions to create and manipulate semantic values. In particular, semantic actions may appear anywhere in a production and typically define the production's semantic value through an assignment to `yyValue` (so named in deference to Yacc). Bindings assign the semantic value of an expression to a variable, making that value accessible in subsequent actions. Additionally, an and-predicate operator '`&`' directly followed by a semantic action is interpreted as a *semantic predicate*, whose code must evaluate to a boolean value. Finally, a string match `"text"`:*e* is semantically equivalent to:

fresh-id:*e* { `"text".equals(`*fresh-id*`)` }

However, since this is a common idiom for writing parsing expression grammars, it is directly supported by *Rats!* (which also allows it to generate slightly more efficient code).

3.1 Determining Semantic Values

Typically, parsers do not just determine whether an input conforms to a language specification but rather generate an abstract syntax tree (AST) for further processing by a tool. The AST is constructed through semantic actions. As an example, consider this production, which, coincidentally, recognizes productions, from *Rats!*' own grammar:

```
Production Production =
  isTransient:("transient":Id)?
  type:QualifiedName nt:Nonterminal
  Assignment choice:Choice Semicolon
  { yyValue = new
      Production(null != isTransient,
                 type, nt, choice); }
  ;
```

The production binds the results of recognizing the optional `transient` keyword and the mandatory nonterminals `QualifiedName`, `Nonterminal`, and `Choice`, while also ignoring the semantic values of the `Assignment` and `Semicolon` nonterminals. It then constructs the corresponding AST node by using the bound values. Note that the first occurrence of the word `Production` defines the Java type of the semantic value, while the second one represents the nonterminal. The usage of the optional `transient` keyword is explained in Section 5.1.

While semantic actions provide considerable flexibility in creating a production's semantic value, they are not always necessary, thus needlessly cluttering a grammar, and, even worse, can also lead to inefficient or incorrect packrat parsers. For instance, it is often convenient to create productions, which consume keywords or punctuation before or after another nonterminal or which combine several nonterminals into a larger choice. In either case, the referenced nonterminals typically recognize more complex expressions, and the semantic value of the higher-level production is the same as the semantic value of one of the nonterminals. In other words, the higher-level production only passes the semantic value through but does not create a new one.

More importantly, productions that recognize lexical syntax either do not need to return semantic values at all—they may, for example, simply consume white space and comments—or they only need to return the text matched in the input as a string. However, explicitly creating such a string within an semantic action is not only tedious, but can also lead to inefficient or incorrect packrat parsers. The underlying problem is that semantic values for packrat parsers *must* be implemented by functional data structures: mutating a data structure after it has been memoized invalidates the parser's state. As a result, the common idiom for efficiently building up Java strings through string buffers must not be used in packrat parsers. To address these issues, *Rats!* adds support for easily passing a semantic value through a production and for simplifying lexical analysis through void and text-only productions. The support for void productions has the added benefit that it further strengthens our parser generator's support for passing a value through. We now discuss these features in turn.

Rats! provides two ways of passing a semantic value through a production; in either case, no semantic actions are required. First, grammar writers can explicitly bind to `yyValue`. This technique is illustrated in the following production, again from *Rats!*' own grammar:

```
Action Header =
  "header":Id yyValue:Action ;
```

The production recognizes the keyword "header" followed by an action; its semantic value simply is the AST node representing the action's code. Second, for many expressions, *Rats!* can automatically deduce the semantic value. As an example, consider this production:

```
Element Primary =
  Nonterminal / Terminal / Action /
  OpeningParenthesis Choice ClosingParenthesis
  ;
```

Since the first three alternatives contain only a single nonterminal each, *Rats!* can easily deduce that each alternative's semantic value is the value of the referenced production. The semantic value of the fourth alternative is the value of the `Choice` production, which *Rats!* deduces with the help of void productions.

A void production is a production with a declared type of `void`; its semantic value is `null`. Void productions are useful for recognizing punctuation elements, such as the operators appearing in *Rats!*' own grammar, or ignored spacing including comments. For example, the following void production recognizes an opening parenthesis followed by spacing:

```
void OpeningParenthesis =
  '(' Spacing ;
```

Void productions also improve the accuracy of *Rats!*' automatic deduction of a compound expression's semantic value: If the compound expression references only a single non-void nonterminal, that nonterminal's semantic value must be the overall expression's value. In the example of the above `Primary` production, both `OpeningParenthesis` (as shown) and `ClosingParenthesis` reference void productions, while `Choice` does not. Consequently, the semantic value of the overall expression is the value of the `Choice` production.

A text-only production is a production with a declared type of `String`. Additionally, it may not contain any actions and may reference only other text-only productions. The semantic value of a text-only production is the text matched in the input. Text-only productions are typically used for recognizing identifiers, keywords, and literals. For example, the following production from *Rats!*' own grammar recognizes string literals:

```
String StringLiteral =
  ["] (EscapeSequence / !["\\] .)* ["] ;
```

The semantic value of this production is the entire string literal, including the opening and closing double quotes. Note that the "`![“\\] .`" expression is read as "any character but a double-quote or backslash."

A final issue that impacts how semantic values are determined is *Rats!*' processing of options, repetitions, and nested choices. To ensure that parser code generation is correct and manageable, our parser generator, similar to Ford's Pappy, lifts options, repetitions, and nested choices into their own productions. Furthermore, it desugars options into choices with an empty, second alternative and repetitions into the corresponding right-recursive expressions. Desugaring repetitions is also necessary so that the component expressions are correctly memoized; otherwise, a packrat parser might not recognize its input in linear time. Note that nested choices that appear as the last element in a sequence need not be lifted. Further note that repeated choices in void and text-only productions are combined during lifting and desugaring (and not lifted into separate productions). The semantic value of the second, empty alternative of a desugared option is `null`. For instance, the "`null != isTransient`" test in the above production for `Production` relies on this to determine the presence of the `transient` keyword. Furthermore, the semantic value of a desugared repetition is a functional list of the component expressions' semantic values; just like the corresponding lists in Scheme or Haskell, *Rats!*' functional lists are implemented as sequences of pairs. To better integrate with Java, functional lists can easily be converted into the corresponding lists in the Java collections framework. Finally, the semantic values of a nested choice must be specified individually in the different alternatives of the choice, unless, of course, *Rats!* can automatically deduce them.

3.2 Error Handling

So far, we have focused on *Rats!*' support for parsing well-formed inputs and generating the corresponding abstract syntax trees through semantic actions. In reality, however, both grammars and language source files are likely to contain errors. Consequently, to assist tool developers in debugging grammars and users in debugging source files, *Rats!* includes a number of error detection and reporting facilities.

Like other recursive descent parsers, packrat parsers cannot support left-recursion. Accordingly, *Rats!*, among several other grammar validity checks, detects both direct and indirect left-recursion and reports a grammar error.[2] Furthermore, like other packrat parsers, *Rats!*-generated parsers collect parse errors even for successful parser steps. To illustrate the need for always tracking parse errors, consider this grammar fragment:

```
Production+ EndOfFile
```

The `Production+` expression succeeds for *any* input that contains at least one valid production. If, however, the input contains an additional production with an embedded syntax error, this grammar fragment fails on the `EndOfFile` expression. If the parser does not track parse errors, it can only generate the not very illuminating error message "end of file expected." However, by tracking parse errors even for successful steps, it can generate a more specific message, such as "assignment expected."

In addition to these error handling facilities also supported by Ford's Pappy, *Rats!* adds the following four features. First, to aid with the debugging of grammars, *Rats!* can pretty print grammars right after parsing and after performing its optimizations. Grammar writers can also select which optimizations should be performed. Second, *Rats!*-generated parsers enforce the declared type of each production's semantic value, with the result that type errors are detected when compiling a parser. While *Rats!* relies on type erasure [9] to memoize semantic values in instances of a common container class `SemanticValue`, it is sufficient for type safety to declare `yyValue` and all bound variables with the correct type. Third, parse error messages are automatically

[2]Ford's Pappy automatically converts direct left-recursions into the corresponding right-recursions; we have not yet implemented this feature in *Rats!*.

```
public abstract class Result {
  // The parser object.
  public final PackratParser parser;

  // Create a new result.
  public Result(PackratParser parser) {
    this.parser = parser;
  }

  // Determine if the instance has a value.
  public abstract boolean hasValue();

  // Get the actual value.
  public abstract Object semanticValue();

  // Get the (embedded) parse error.
  public abstract ParseError parseError();

  // Create a new semantic value, using this result's
  // parser.
  public abstract SemanticValue
    createValue(Object value, ParseError error);
}
```

Figure 1: The common base class for semantic values and parse errors.

deduced from nonterminal names. For example, a parse error within the production for `StringLiteral` results in the error message "string literal expected." The reported position in the input is the start of the production. However, for string literals and string matches, which are typically used for recognizing keywords or punctuation, the error message specifies the string and the beginning of the corresponding expression. Finally, parsers automatically track file names, line numbers, and column numbers. Furthermore, if semantic values are instances of our source-to-source transformer's AST nodes, *Rats!*-generated parsers can optionally annotate these nodes with the corresponding information. That way, later tool phases can easily report the location of semantic errors. Overall, *Rats!*' error handling facilities have been designed so that tool implementors can focus on the functionality of their tools and need not worry about the details of error detection and reporting.

4. PARSER IMPLEMENTATION

Each parser generated by *Rats!* has a main class, which is instantiated once for each character in the input. This parser class thus represents the columns of a packrat parser's memoization table. For each production, the parser class has a field to store the memoized intermediate result and a corresponding accessor method. On invocation, the accessor method tests whether the field's value is `null`. If so, the accessor calculates the result, stores it, and then returns it. If the field is not `null`, the accessor method simply returns the stored value.

All parser classes have a common parent, named `PackratParser`, which provides access to the characters in the input and tracks file names as well as line and column numbers. Characters are read in, on demand, from regular Java character streams. While seemingly trivial, this implementation detail avoids an important restriction when compared to

```
yyResult = this.pId();
yyError  = yyError.select(yyResult.parseError());
if (yyResult.hasValue()) {
  String att = (String)yyResult.semanticValue();
  ...
}
```

Figure 2: An example code snippet corresponding to the expression `att:Id`, with `Id` having a semantic value of type `String`. The code snippet first attempts to match the `Id` production in the input. It then records any parse errors in `yyError`. Next, if the match has been successful, it binds the corresponding semantic value to `att`.

Ford's packrat parsers: Ford states [13, 14] that his parsers need to have the *entire* input available up-front, thus making them unusable for interactive applications. Because they use Java's character streams, this is not the case for *Rats!*-generated parsers.

When receiving a result from an accessor method, parsers need to easily distinguish between semantic values and parse errors, as they need to execute different code depending on the type of result. To this end, we leverage Java's object-oriented features and represent them through two separate container classes. The container class for semantic values, named `SemanticValue`, stores the actual value, a reference to the parser object representing the input after the parsed expression, and possibly an embedded parse error. As already hinted at in Section 3.2, the field for the actual value is declared to be a Java `Object`, which is an application of type erasure [9] and allows us to use the same container class for all types of semantic values. The container class for parse errors, named `ParseError`, stores the error message and a reference to the parser object representing the location of the error.

Both container classes have a common base class, which is shown in Figure 1. The implementation of the concrete methods is trivial—between one and two lines of code per method. At the same time, the use of this common base class significantly simplifies the implementation of memoizing parsers, as illustrated in Figure 2. In particular, *no* `instanceof` tests are necessary to distinguish between semantic values and parse errors, and *no* type casts are required to access each container class. Due to our use of type erasure, type casts are still necessary for accessing the actual semantic values. However, these casts cannot fail, as the corresponding `yyValue` declarations in the productions that create the values have the same type.

5. OPTIMIZATIONS

With the overview of our parser implementation in place, we can now turn to the optimizations performed by *Rats!*. The goals for optimizing packrat parsers are two-fold. First, the optimizations should reduce the size of the table memoizing intermediate results. Decreasing the size of this table is important not only for keeping heap utilization as low as possible but also for improving parser performance. After all, a smaller memoization table decreases the frequency of memory allocator and garbage collector invocations and also

Name	Description	*Rats!*
Chunks	Break memoizing fields into chunks; do not memoize productions referenced only once.	√
Grammar	Fold duplicate productions and eliminate dead productions.	√
Transient	Do not memoize transient productions.	New
Choices	Inline transient productions into choices.	New
Terminals	Optimize recognition of terminals, notably by using switch statements.	Improved
Prefixes	Fold common prefixes.	New
Errors	Avoid the creation of unnecessary parsing errors.	New
Values	Avoid the creation of duplicate semantic values.	New
Repeated	Do not desugar transient repetitions.	New
Cost	Perform cost-based inlining.	√

Table 2: Overview of optimizations.

increases the table fraction that fits into a processor's caches. Second, the optimizations should improve the performance of productions that recognize lexical syntax. This is important, because packrat parsers integrate lexical analysis with parsing and thus cannot utilize well-performing techniques, such as DFAs [1], for recognizing tokens.

Table 2 summarizes the optimizations performed by *Rats!*; it also identifies which optimizations are new or improved in comparison to Ford's Pappy. The chunks, grammar, and cost optimizations, which are also performed by Pappy, work as follows. First, the *chunks* optimization is based on the observation that most table fields for a given parser object never memoize a result, i.e., remain `null`. Consequently, to reduce the memory overhead of allocating the fields, the chunks optimization introduces a level of indirection and allocates fields in chunks. The parser object, in turn, references the chunks, instead of referencing results directly. Additionally, if a nonterminal is referenced only once within a grammar, the parser cannot backtrack on the corresponding production, and the production is not memoized at all.

Next, the *grammar* optimization is based on the observation that the lifting and desugaring of expressions described in Section 3.1 can result in duplicate productions, which might even *increase* the size of the memoization table. Consequently, to minimize the impact of these transformations, the grammar optimization folds equivalent productions into a single one. Comparable to dead code elimination, it also removes non-top-level productions that are never referenced from the grammar. Finally, the *cost* optimization inlines productions, with the goal of avoiding the overhead of invoking accessor methods and performing memoization. However, since indiscriminate inlining can invalidate the linear-time performance guarantee of packrat parsers, the cost optimization only inlines very small productions.

We now describe the optimizations new to or improved by *Rats!*. More specifically, in Section 5.1, we show how to further reduce the number of productions that need to be memoized. Next, in Section 5.2, we explore how to improve the recognition speed for lexical syntax. In Section 5.3, we describe how to reduce the memory overhead for productions that need to be memoized. Finally, in Section 5.4, we explore an optimization that avoids the possibly deep recursion of desugared repetitions when recognizing long inputs.

5.1 Transient Productions

The *transient* optimization is based on the observation that packrat parsers typically do not need to backtrack for productions recognizing lexical syntax. In particular, identifiers, keywords, operators (such as "`<=`"), and punctuation (such as "`;`") have straight-forward productions that do not backtrack. More importantly, spacing, which includes all white space and comments, makes up large parts of most programming language source files but also does not require backtracking. At the same time, productions that recognize numeric literals often *do* need to backtrack. However, if a production does not need to backtrack, there is no need to memoize the intermediate results.

Consequently, the transient optimization gives grammar writers control over which productions are memoized through the `transient` keyword. If a production is declared to be `transient`, *Rats!* does not allocate a field for memoizing the production's result; rather, the corresponding parsing code is always executed. The use of this keyword is illustrated in the following production, which consumes optional white space and comments:

```
transient void Spacing =
    ( WhiteSpace
    / TraditionalComment
    / EndOfLineComment )*
    ;
```

Obviously, the indiscriminate use of the `transient` keyword can negate the benefits of memoization and result in parsers that perform in time (considerably) worse than linear. As a result, grammar writers need to use the `transient` keyword with care. They should either verify that a production cannot backtrack or measure the effects over a set of representative inputs. In practice, we expect that grammar writers mostly reuse the corresponding productions from the grammars distributed with our source release of *Rats!*.

5.2 Improved Terminal Recognition

Like the corresponding optimization in Ford's Pappy, the *terminals* optimization is based on the observation that many productions for recognizing lexical syntax have alternatives that start with different characters. To improve recognition speed, the terminals optimization replaces successive `if` statements that parse disjoint lexical alternatives with a single `switch` statement, while also folding alternatives that start with the same characters into one alternative with a common prefix.[3] Furthermore, to avoid the dynamic instantiation of text matched in the input, our version also uses literal Java strings, if the recognized text can be statically determined, and converts text-only productions, whose semantic value is never used, i.e., bound, into the corresponding void productions. The *prefixes* optimization generalizes

[3]Note that the resulting nested choice does not need to be lifted because it always appears as the last element in a sequence.

the folding of alternatives with common prefixes to nonterminals, with the goal of avoiding repeated calls to accessor methods and possibly even memoization.

However, the effectiveness of the terminals optimization depends to some degree on how a grammar has been written. For instance, the `Spacing` production shown in Section 5.1 references three nonterminals instead of directly specifying the expressions for recognizing white space and comments. As a result, the terminals optimization can use a `switch` statement to recognize white space, but not for spacing overall. The *choices* optimization addresses this problem and creates further opportunities for the terminals optimization. It is based on two observations. First, most productions for recognizing lexical syntax never backtrack and can safely be declared as transient. Second, if a transient production references another transient production, the referenced production can safely be inlined into the first, as no intermediate results need to be memoized. Consequently, if a nonterminal appears as the only expression in an alternative, the nonterminal references a void or text-only production, and both the referencing and the referenced productions are transient, the choices optimization simply inlines the referenced production. In the case of the `Spacing` production, the overall result of the combined terminals and choice optimizations is a single `switch` statement for recognizing white space *and* comments instead of just white space alone.

5.3 Avoiding the Creation of Results

The errors and values optimizations reduce the number of `ParseError` and `SemanticValue` container objects allocated by packrat parsers. The *errors* optimization is based on the observation that most alternatives in a production's top-level choice fail on the first expression. For example, the statement production for any C-like language has a large number of alternatives. Only one of these alternatives can succeed on a given input and most alternatives are completely distinct, starting with a different keyword. Consequently, the errors optimization suppresses the generation of a parse error when the first expression in an alternative fails. At the same time, parse errors are still generated when a subsequent expression fails or when *all* alternatives fail. Generating a single parse error when all of a production's alternatives fail on their respective first expressions has one added benefit: the corresponding error message is more meaningful, as it indicates that the overall production has failed. For example, instead of "if expected" (assuming that the last alternative parses the if statement) the error message would read "statement expected."

The *values* optimization is based on the observation that many productions simply pass the semantic value through. For example, Java has 17 expression precedence levels, which are implemented by separate productions. All of these productions must be invoked to recognize a primary expression, such as a literal or identifier, with productions of lower precedence levels simply passing the corresponding value through. As discussed in Section 3.1, similar observations have motivated us to simplify the specification of such productions by eliminating the need for explicit semantic actions. The values optimization, however, does not depend on how a semantic value is calculated, be it through an explicit action, an assignment to `yyValue`, or through *Rats!*' value deduction facilities. Rather, where possible, new instances of `SemanticValue` are created by invoking the `createValue()` method shown in Figure 1 on the last result accessed while parsing an expression. The method's implementation for semantic values uses the reference equality test `==` to compare the specified value and error with its own and, if they are identical, returns `this` instead of allocating a new container object. This simple and dynamic delegation of object creation makes the values optimization available to a much larger class of parsing expressions than would be possible with using only static analysis in the parser generator.

5.4 Repetitions in Transient Productions

Our final optimization is motivated by the observation that desugared repetitions may recurse quite deeply when matching a large number of repeated expressions. As a result, we have observed stack overflow errors on some Java virtual machines when parsing source files with very long comments (on the order of several printed pages). In general, repetitions need to be desugared into the corresponding right-recursive expressions to ensure that the component expressions are correctly memoized. However, as discussed in Section 5.1, spacing, including comments, typically does not require memoization, as a packrat parser does not backtrack for the corresponding productions. Consequently, the *repeated* optimization preserves repetitions in transient void or text-only productions. While this optimization has complicated parser code generation considerably, it also ensures that *Rats!*-generated parsers can scale over longer inputs.

6. EXPERIMENTAL EVALUATION

In this section, we present the results of our experimental evaluation. The goal is to understand the costs associated with packrat parsing and thus the costs of making parsers more easily extensible. We focus on quantifying the overall performance of packrat parsers, notably their memory overhead and latency, and the effects of the optimizations presented in Section 5. More specifically, we compare the performance of a Java parser generated by *Rats!* with the corresponding parsers generated by ANTLR and JavaCC over a sampling of Java source files. We also analyze the impact of the different optimizations when parsing an example source file.

To summarize our results, we show that *relative* to ANTLR and JavaCC our packrat parser has noticeable overheads, while its *absolute* performance is reasonable on modern computer hardware. For instance, on a 2002 consumer-level computer, the *Rats!*-generated parser processes a 67 KB Java source file in only 0.3 seconds, which is acceptable given the complexity of subsequent language processor phases. Furthermore, while our experimental setup is not directly comparable with Ford's, packrat parsers generated by *Rats!* perform considerable better than those generated by Ford's parser generator, Pappy, requiring only a quarter as much memory and parsing 4.6 times faster. Finally, we show that all optimizations besides prefix folding and cost-based inlining result in measurable improvements in parser performance. Chunking and transient productions are particularly effective at reducing the memory overheads of packrat parsers, while the other optimizations improve latency. From these results, we conclude that packrat parsers generated by *Rats!* are a realistic building block for making compilers and other programming language processors more easily extensible.

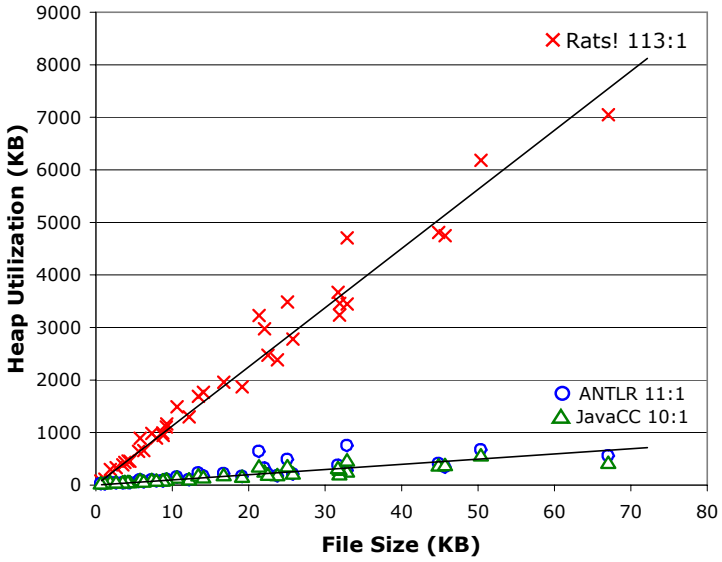

Figure 3: Heap utilization.

6.1 Experimental Setup

All measurements reported in this section were performed on both a consumer-level and a top-of-the-line computer from the fall of 2002. The consumer-level computer is an Apple iMac, with an 800 MHz PowerPC G4 processor and 1 GB of RAM, running Mac OS X 10.3.2 and Apple's port of the Java Development Kit version 1.4.1. Spotchecks with the more recent version 1.4.2 show the same results. The top-of-the-line computer is a Dell Dimension 8250, with a 3 GHz Pentium IV processor and 1 GB of RAM, running Windows XP Professional (service pack 1) and Sun's Java Development Kit version 1.4.2.

Our experiments compare parsers created with three different parser generators. The first parser is a packrat parser generated by *Rats!* from our own Java grammar. Unless otherwise noted, our packrat parser has been generated with all optimizations *besides* prefix folding and cost-based inlining. The second parser has been generated by ANTLR version 2.7.2, using a modified version of the Java grammar (v. 1.20) distributed by the ANTLR project. Finally, the third parser has been generated by JavaCC version 3.2, using the Java grammar dated 5/5/2002 and distributed by the JavaCC project. To measure only parser performance, all three grammars do not contain any semantic actions and do not create an abstract syntax tree. In fact, our modifications to ANTLR's grammar only remove the instructions for creating an AST. As inputs, we use a sampling of 38 Java source files taken from the Cryptix open source cryptographic libraries [34], from ANTLR's sources, and from *Rats!*' sources. The files are between 766 bytes and 67 KB large, represent a variety of programming and commenting styles, and contain a total of 714 methods with 8,058 non-commenting source statements.

All measurements represent the average of 20 iterations over the same input. To ensure that the Java virtual machine could load all classes and perform just-in-time compilation of commonly executed code, we perform two additional iterations before the 20 instrumented ones. We believe that this setup is consistent with a compiler processing several source files in a single invocation. To exclude the overhead of accessing the file system, each experiment also reads the input file into memory before parsing. Furthermore, to exclude the overhead of automatic memory management, we allocate an initial Java heap sufficiently large to avoid garbage collection during each iteration; though, we do force GC before each iteration. As a result, our heap utilization numbers reflect total memory pressure for each iteration and may include objects that are not reachable by the end of the iteration.

6.2 Overall Performance

Figure 3 graphs the heap utilization in KB against file size for the 38 Java source files as measured on the iMac. Figure 4 graphs the corresponding latency in milliseconds, also measured on the iMac. The graphs clearly show that memoization is an effective technique for packrat parsers, as both heap utilization and latency grow only linearly with input size (with latency showing larger variations). However, they also illustrate the costs of memoization, as our packrat parser requires 113 bytes for each byte in the input, which is an order of magnitude more memory than for both the parsers generated by ANTLR and JavaCC. Furthermore, our packrat parser performs 2.94 times slower than the parser generated by ANTLR, which, in turn, performs 2.24 times slower than the parser generated by JavaCC. We believe that ANTLR's inferior performance when compared to JavaCC is mostly due to its lexer, which, like the parser, uses a recursive descent algorithm instead of JavaCC's table-driven lexer.

When performing the same experiments on the Dell PC, the measured heap utilization is the same as on the iMac: 113:1 for *Rats!*, 11:1 for ANTLR, and 10:1 for JavaCC. Due to the limited resolution of Java's `System.currentTimeMillis()`, however, we could only measure the parsing latency for the 13 largest input files. All three parsers still perform linearly relative to input size, though at different rates: 625 KB/second for *Rats!*, 1375 KB/second for ANTLR, and

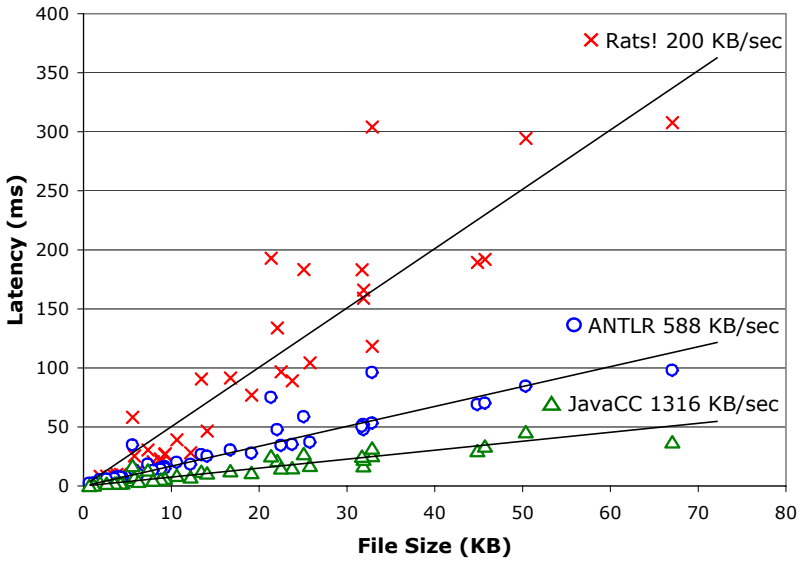

Figure 4: Latency.

3125 KB/second for JavaCC. While the latency ratio between ANTLR and JavaCC is almost the same on the Dell PC—the ANTLR parser performs 2.27 times slower than the JavaCC parser, the latency ratio between our packrat parser and the ANTLR parser is only 2.20. We believe that this improvement reflects the Dell PC's superior memory subsystem: Not only has the Dell PC double the level-2 processor cache at 512 KB instead of 256 KB, but it also has a considerably faster system bus at 533 MHz instead of 100 MHz and correspondingly faster memory. In other words and not surprisingly due to their relatively large memory requirements, packrat parsers can clearly benefit from large processor caches.

6.3 Effects of the Optimizations

Figure 5 illustrates the effects of the different optimizations when parsing *Rats!*' own code generator, showing both heap utilization—the lower curve—and latency—the upper curve. Note that the names of the individual optimizations are defined in Table 2. Further note that optimizations are cumulative from left to right. For example, the data points labelled "transient" include the chunks, grammar, and transient optimizations.

The figure shows that the largest savings in heap utilization are due to the use of chunks for storing memoized results. The use of transient productions reduces heap utilization by another 43%. While the other optimizations reduce heap utilization by only a further 13%, they have a considerable impact on latency. In particular, the grammar optimization reduces latency by 15% when compared to only performing the chunks optimization. Furthermore, the choices, terminals, errors, values, and repeated optimizations together reduce latency by another 53% when compared to only using the chunks, grammar, and transient optimizations.

Cost-based inlining, however, has almost no impact on heap utilization and slightly increases latency. Even worse, prefix folding, which is not shown in Figure 5 and has been measured separately for all source files, increases heap utilization by 7% to 121 bytes for each byte in the input. This increase is due to the fact that prefix folding changes which fields are part of which chunk, with the result that more chunks are allocated. Based on these results, both cost-based inlining and prefix folding are disabled by default. However, grammar writers can use command line flags to control which optimizations are performed by *Rats!*, including cost-based inlining and prefix folding.

6.4 Discussion

As shown above, our packrat parser for Java has considerably higher memory requirements than the corresponding parsers generated by ANTLR and JavaCC. It is also slower, between 2.20 and 2.94 times in our experiments when compared to the corresponding ANTLR parser. At the same time, grammars for *Rats!* are easier to modify and extend. They also are more concise: our Java grammar has 530 lines, while both the ANTLR and JavaCC grammars are more than twice as large, each comprising about 1,200 lines. Furthermore, our packrat parser has reasonable *absolute* performance, for example, parsing a 67 KB source file in only 0.3 seconds and requiring 7,571 KB of memory. Given that source files in many object-oriented languages tend to be small, we believe that these overheads are acceptable, especially for modern computer hardware. We also note that the high heap utilization of packrat parsers does not impact later language processor phases. Any memoized intermediate results only need to be available during parsing and can be safely discarded after the AST has been built.

In comparison to Ford's work, *Rats!* represents a clear improvement in resource utilization. His packrat parsers are written in Haskell, do construct an AST, and were measured on a 1.3 GHz Athlon PC running Linux. The Java parser generated by Ford's parser generator, Pappy, has a heap utilization of 441:1 and parses 43.4 KB/second. A handwritten packrat parser performs better, showing a heap utilization of 297:1 and parsing 52.1 KB/second. While the two experimental setups are not directly comparable, the Java parser generated by *Rats!* performs better in absolute terms, expos-

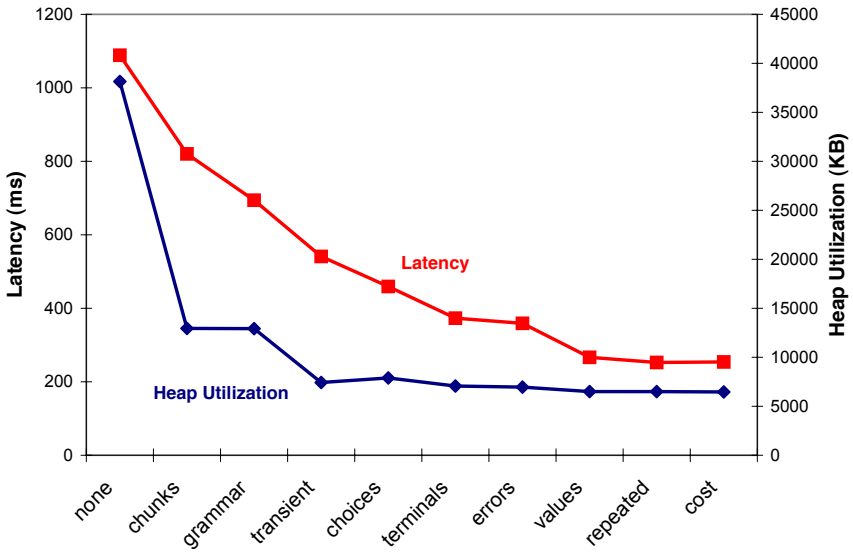

Figure 5: Effects of individual optimizations.

ing a heap utilization of 113:1 and parsing 200 KB/second on roughly comparable hardware. Furthermore, as discussed in Section 6.3, our additional optimizations are also effective at reducing heap utilization and improving parsing speed in relative terms. Overall, we conclude that, while traditional parsers are still an appropriate substrate for compilers that target static languages, *Rats!* represents a realistic building block for making compilers and other language processing tools more easily extensible.

7. OUR TRANSFORMER FRAMEWORK

In parallel to developing *Rats!*, we have also been creating a framework for extensible source-to-source transformers. *Rats!* is implemented within this framework, and, as mentioned in Section 3.2, *Rats!*-generated parsers can automatically annotate AST nodes from our framework with location information to simplify error reporting. Additionally, both *Rats!* and our framework are being used as the basis for term projects in our department's PhD-level compilers course.

Our source-to-source transformer framework is centered around three main abstractions, which are implemented by extending the corresponding abstract base classes:

Nodes. Abstract syntax tree nodes represent the structure of programs, such as declarations, statements, and expressions. Nodes can be annotated with metadata through properties, which map names to the corresponding values.

Visitors. Visitors represent the different phases of a language processor, such as semantic analysis, optimizations, and code generation. They can either walk or modify a program's AST.

Utilities. Utilities provide state and functionality shared between several visitors. So far, we have implemented a concrete utility for analyzing *Rats!*' grammars and one for pretty printing grammar as well as program sources.

To provide scalable extensibility [28], visitors do not rely on statically declared interfaces, as in the original visitor design pattern [16]. Rather, the appropriate `visit()` methods are selected through reflection [6] based on an AST node's type. Lookups are cached to improve dynamic dispatch performance. Furthermore, language processor functionality can also be implemented by AST nodes: instead of accepting an AST node in a `visit()` method, the corresponding AST node methods accept a visitor in a `process()` method. The appropriate `visit()` or `process()` method is transparently selected through our reflection-based dynamic dispatch facility. As a result, both transformer and language extensions have straight-forward implementation strategies: New transformer phases can be structured as complete visitors, while new language constructs can be structured as AST nodes that also specify the corresponding analysis and transformation methods. In either case, the extension requires a single main class that concisely defines the added functionality.

While our framework clearly is not as mature as Polyglot [28], which provides a toolkit for extending Java and has been used for several language extensions [24, 27, 26], we still believe that our use of reflection-based dynamic dispatch represents a useful alternative to Polyglot's mixin-based extension model. In particular, Polyglot's mixin-based model requires that all compiler functionality is associated with AST nodes. This clearly simplifies the addition of new language features to a compiler—which, after all, is the the goal behind Polyglot's design. However, it also makes the addition of new compiler phases rather cumbersome. In contrast, reflection-based dynamic dispatch allows us to easily support both language and compiler extensions, albeit at some performance overhead.

8. FUTURE WORK

For future work, we plan to focus on two issues. First, our optimizations are effective at eliminating entire rows—which correspond to nonterminals in a grammar—from a packrat parser's memoization table and at also reducing the memory overhead of the remaining rows. However, we believe that it may be possible to also avoid the instantiation of entire columns—which correspond to characters in the input. The key observation is that programming language source files typically contain considerable amounts of spacing, which are also ignored by most language processing tools. Since spacing is recognized by transient productions, the corresponding productions are not memoized. At the same time, our parsers still allocate a parser object for each spacing character in the input, even though *no* memoization fields are used. The new optimization would thus avoid the allocation of parser objects for spacing and similar productions.

A second issue is that our parser generator still requires tool implementors to explicitly define AST nodes, which unnecessarily complicates the initial development of language processors. ANTLR, for example, addresses this problem by including support for easily creating abstract syntax trees that are based on a generic tree node class [29]. However, it also lacks the integration with a framework for building tools that process the generated trees. We believe that we can do better by integrating such a generic tree node facility not only with our parser generator but also with our framework for building source-to-source transformers.

9. CONCLUSIONS

As object-oriented programming languages are evolving at a rather rapid pace, language implementors need appropriate tools, such as easily extensible parser generators, to keep up with these changes. However, context-free grammars and the corresponding LR and LL parsers, while well understood and widely used, are also unnecessarily hard to extend. To address this need, we have introduced *Rats!*, a parser generator for Java, that supports easily modifiable grammars and avoids the complexities associated with altering LR and LL grammars. Our parser generator builds on recent research by Ford on packrat parsers, which are recursive descent parsers that perform backtracking but also memoize each intermediate result, thus ensuring linear-time performance.

Compared to Ford's packrat parser generator, *Rats!* supports simpler grammar specifications by automatically deducing a production's semantic value and through void and text-only productions. Additionally, it features improved error detection and reporting facilities. Finally, it performs more aggressive optimizations, which not only reduce the memory requirements for a packrat parser's memoization table but also improve its recognition speed, especially for productions that perform lexical analysis. Our performance evaluation shows that these optimizations are effective. It also illustrates that, while packrat parsers have higher resource requirements than more conventional parsers, they have acceptable absolute performance on modern computer hardware. For example, a *Rats!*-generated Java parser processes a 67 KB source file in only 0.3 seconds on a consumer-level computer, while also requiring 7,571 KB of memory. Based on these results, we conclude that *Rats!* provides a practical building block for building more easily extensible language processing tools. The open source release for our parser generator and the corresponding source-to-source transformer framework is available at `http://www.cs.nyu.edu/rgrimm/xtc/`.

10. ACKNOWLEDGMENTS

We thank Benjamin Goldberg and Brian Ford for their feedback and discussions.

11. REFERENCES

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice Hall, June 1972.

[2] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the 2001 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 31–42, Tampa Bay, Florida, Oct. 2001.

[3] J. Baker and W. C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation*, pages 270–281, Berlin, Germany, June 2002.

[4] A. Birman. *The TMG Recognition Schema*. PhD thesis, Princeton University, Feb. 1970.

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

[6] J. Blosser. Java tip 98: Reflect on the Visitor design pattern. *JavaWorld*, July 2000. Available at `http://www.javaworld.com/javaworld/javatips/jw-javatip98.html`.

[7] C. Braband, M. I. Schwartzbach, and M. Vanggaard. The METAFRONT system: Extensible parsing and transformation. Technical Report BRICS RS-03-7, BRICS, Aarhus, Denmark, Feb. 2003.

[8] G. Bracha. Generics in the Java programming language. Tutorial, Sun Microsystems, Mar. 2004. Available at `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.

[9] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 183–200, Vancouver, Canada, Oct. 1998.

[10] R. Cartwright and G. L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In *Proceedings of the 1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 201–215, Vancouver, Canada, Oct. 1998.

[11] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 2000 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 130–145, Minneapolis, Minnesota, Oct. 2000.

[12] O. Enseling. Build your own languages with JavaCC. *JavaWorld*, Dec. 2000. Available at `http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html`.

[13] B. Ford. Packrat parsing: A practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 2002.

[14] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 ACM International Conference on Functional Programming*, pages 36–47, Pittsburgh, Pennsylvania, Oct. 2002.

[15] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pages 111–122, Venice, Italy, Jan. 2003.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Jan. 1995.

[17] J. J. Heiss. New language features for ease of development in the Java 2 platform, standard edition 1.5: A conversation with Joshua Bloch. Article, Sun Microsystems, May 2003. Available at `http://java.sun.com/features/2003/05/bloch_qa.html`.

[18] ISO/IEC. Information technology—syntactic metalanguage—extended BNF. ISO/IEC Standard 14977, International Standards Organization/International Electrotechnical Commission, Geneva, Switzerland, 1996.

[19] A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, Snowbird, Utah, June 2001.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[21] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–12, Anaheim, California, Oct. 2003.

[22] K. Lee, A. LaMarca, and C. Chambers. HydroJ: Object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 205–223, Anaheim, California, Oct. 2003.

[23] J. R. Levine. *lex & yacc*. O'Reilly, Oct. 1992.

[24] J. Liu and A. C. Myers. JMatch: Iterable abstract pattern matching for Java. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 110–127, New Orleans, Louisiana, Jan. 2003.

[25] T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 224–240, Anaheim, California, Oct. 2003.

[26] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, Jan. 1999.

[27] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 132–145, Paris, France, Jan. 1997.

[28] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, Warsaw, Poland, Apr. 2003. Springer-Verlag.

[29] T. J. Parr. ANTLR reference manual. Available at `http://www.antlr.org/doc/index.html`, Jan. 2003.

[30] T. J. Parr and R. W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *Proceedings of the 5th International Conference on Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 263–277, Edinburgh, Scotland, Apr. 1994. Springer-Verlag.

[31] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.

[32] G. L. Steele, Jr. Growing a language, Oct. 1998. Invited talk at the *1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*.

[33] Sun Microsystems. JavaCC: LOOKAHEAD MiniTutorial. Technical report, 2004. Available at `https://javacc.dev.java.net/doc/lookahead.html`.

[34] The Cryptix Foundation. Cryptix JCE. Available at `http://www.cryptix.org/`.

[35] M. Viroli and A. Natali. Parametric polymorphism in Java: An approch to translation based on reflective features. In *Proceedings of the 2000 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–165, Minneapolis, Minnesota, Oct. 2000.

[36] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, Nov. 1977.