Better Extensibility through Modular Syntax

Robert Grimm
New York University
rgrimm@cs.nyu.edu

Abstract

We explore how to make the benefits of modularity available for syntactic specifications and present *Rats!*, a parser generator for Java that supports easily extensible syntax. Our parser generator builds on recent research on parsing expression grammars (PEGs), which, by being closed under composition, prioritizing choices, supporting unlimited lookahead, and integrating lexing and parsing, offer an attractive alternative to context-free grammars. PEGs are implemented by so-called packrat parsers, which are recursive descent parsers that memoize all intermediate results (hence their name). Memoization ensures linear-time performance in the presence of unlimited lookahead, but also results in an essentially lazy, functional parsing technique. In this paper, we explore how to leverage PEGs and packrat parsers as the foundation for extensible syntax. In particular, we show how make packrat parsing more widely applicable by implementing this lazy, functional technique in a strict, imperative language, while also generating better performing parsers through aggressive optimizations. Next, we develop a module system for organizing, modifying, and composing large-scale syntactic specifications. Finally, we describe a new technique for managing (global) parsing state in functional parsers. Our experimental evaluation demonstrates that the resulting parser generator succeeds at providing extensible syntax. In particular, *Rats!* enables other grammar writers to realize real-world language extensions in little time and code, and it generates parsers that consistently outperform parsers created by two GLR parser generators.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Parsing; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules

General Terms design, languages

Keywords parser generator, extensible syntax, parsing expression grammar, packrat parsing, module system, *Rats!*

1. Introduction

In this paper, we explore how to make the benefits of modularity available for syntactic specifications and hence how to make syntax easily extensible. Our research is motivated by the observation that systems and language researchers alike have been exploring how to leverage domain-specific programming languages for simplifying complex systems. Examples include support for accessing hardware in device drivers [29], controlling information flow [31], event-based programming [27, 30], pattern matching for distributed messages [25], and specifying network protocols [24, 35]. While these efforts differ considerably in methodology, design, and implementation, they all build on C, C++, or Java. An important challenge, then, is how to gracefully extend C-like programming languages and, more specifically, how to provide language implementors with the appropriate tools for realizing their domain-specific compilers. While our larger research agenda is to explore the expression, composition, and implementation of fine-grained 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. Furthermore, easily extensible syntax is beneficial to *any* developer realizing complex syntactic specifications, be they significant extensions to an existing programming language, several dialects of the same language, or distinct languages with considerable syntactic overlap, such as C-like languages and their expressions and statements.

A practical solution for extensible syntax can substantially benefit from having three properties. First, the employed syntactic formalism and parsing algorithm should be closed under composition, to enable modularity, and unambiguous, as computer formats rarely have more than one valid interpretation. Additionally, the syntactic formalism should support scannerless parsing [37], i.e., the integration of lexing with parsing, to also provide extensibility at the lexical level [9, 20, 39]. Second, the module system should be sufficiently expressive to manage large-scale syntactic specifications. In particular, it needs to provide encapsulation by grouping related productions into syntactic units, support the concise expression of modifications to existing units, and enable the flexible composition of syntactic units and their extensions with each other. For example, when adding aspects to C, the grammar writer should only need to change the modules affected by the extension and be able to reuse existing modules, even if they now depend on modified syntax. Third, the parsing algorithm and module system should allow for the expression of syntax as code, since *no* syntactic formalism can conveniently and precisely capture all languages. Notably, C and C++ already are context-sensitive and require global parsing state to disambiguate typedef names (i.e., type aliases) from other names.

Unfortunately, context-free grammars (CFGs) and the corresponding LR, LL, and even GLR parsing algorithms, while well understood and widely used, already fall short of the first property. LR, which, for example, is used by Yacc [26], and LL, which is used by ANTLR [34] and JavaCC [16], only recognize a subset of CFGs and consequently are not closed under composition. To make matters worse, they can also be fairly brittle in the face of change [6, 28]. In contrast, GLR [38], which is used by Bison [21], Elkhound [28] and SDF2 [9, 40], and Earley parsing [1, 14] can recognize all CFGs and thus are closed under composition. How-

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

ever, if a CFG is ambiguous, they return either all abstract syntax trees, which is inefficient when only one tree is the correct one, a heuristically chosen one, which may not be the right one, or require explicit disambiguation [39], which adds complexity to a syntactic specification.

In contrast, parsing expression grammars [3, 4, 20] (PEGs) and packrat parsers [18, 19] *do* have the first property and thus provide a more attractive foundation for extensible syntax. Notably, PEGs are not only *closed under composition* but also intersection and complement. Furthermore, they rely on *ordered choices* instead of the unordered choices used in CFGs, thus avoiding unnecessary ambiguities in the first place. Next, they offer additional expressivity through *syntactic predicates*, which match expressions but do not consume the input. Finally, they are *scannerless* by default. PEGs are implemented by so-called packrat parsers, which are recursive descent parsers that may backtrack. To ensure linear-time performance, they also memoize all intermediate results (hence the name). So far, Ford [18, 19] 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, it also raises the issue of how to utilize PEGs and packrat parsers in C-like languages, which dominate among system builders.

In this paper, we address this issue and present *Rats!*,[1] a parser generator for Java that leverages PEGs to provide all three properties of extensible syntax. Our parser generator is implemented within our own framework for building extensible source-to-source transformers [22], includes working grammars for C and Java, and has been released as open source. It provides the three properties by integrating this paper's three contributions.

First, we show how to implement packrat parsers in a strict, imperative programming language through a careful layout of data structures and a common interface to semantic values and parse errors. We also show how to make the corresponding specifications more concise—by automatically deducing semantic values—and the resulting parsers considerably faster—by aggressively optimizing grammars.

Second, we present a module system for organizing, modifying, and composing syntactic specifications. Notably, the module system relies on so-called module modifications to concisely express how to add, override, or remove individual alternatives in a production. Furthermore, it relies on module parameters to compose different syntactic units, including module modifications, with each other. While the main ideas behind our module system are not novel, their application on syntactic specifications is new—and also suitable for other syntactic formalisms that are closed under composition.

Third, we describe a new technique for managing (global) parsing state in functional parsers, which models state modifications as lightweight transactions and supports the recognition of context-sensitive languages.

Our experimental evaluation demonstrates that *Rats!* is indeed practical for easily extending syntactic specifications. In particular, it enables other grammar writers to realize real-world language extensions—including an aspect-enhanced version of C and a combination of Java and C to simplify Java native interface programming—in little time and code. Furthermore, *Rats!*-generated parsers perform reasonably well, out-performing parsers created by two GLR-based parser generators by at least a factor of 1.9, while being at most 2.7 times slower than parsers created by more conventional LL-based parser generators.

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

Operator	Type	Prec.	Description
' '	Primary	6	Literal character
" "	Primary	6	Literal string
[ ]	Primary	6	Character class
_	Primary	6	Any character
{ }	Primary	6	Semantic action
(*e*)	Primary	6	Grouping
e?	Unary suffix	5	Option
*e**	Unary suffix	5	Zero-or-more
e+	Unary suffix	5	One-or-more
&*e*	Unary prefix	4	And-predicate
!*e*	Unary prefix	4	Not-predicate
id:*e*	Unary prefix	4	Binding
" ":*e*	Unary prefix	4	String match
void:*e*	Unary prefix	3	Voided value
<*name*> $e_1 \ldots e_n$	n-ary	2	Sequence
e_1 / ... / e_n	n-ary	1	Ordered choice

Table 1. The operators supported by *Rats!*. Note that "Prec." stands for precedence level. Further note that the name for sequences is optional.

2. Overview of *Rats!*

From a developer's point of view, he or she first writes a grammar specification for the language to be parsed. In doing so, he or she can omit explicit semantic actions and rely on *Rats!*' facilities for automatically deducing productions' values (Section 3). The developer can also organize the grammar into modules and reuse already existing modules (Section 4). Finally, he or she can manage global parser state through lightweight transactions (Section 5). While these features have been designed to integrate with each other, they also do not depend on each other, thus enabling the grammar writer to use only those features he or she needs.

To generate the corresponding parser, the developer invokes the *Rats!* tool itself, which resolves all module dependencies. It starts with the top-level module specified on the command line, loads dependent modules from the file system based on their names, and produces a single global grammar. *Rats!* also adds explicit semantic actions where it can deduce productions' values. Furthermore, it performs extensive optimizations on the grammar to reduce both heap utilization and latency (Section 8). Finally, it emits the parser's source code, which is then compiled to the final parser. While *Rats!* currently targets only Java, all language-specific aspects have been carefully isolated to ease future ports to other languages.

At runtime, the recursive descent parser generated by *Rats!* tries to match the input and generate the corresponding abstract syntax tree. In general, the parser has one method per production in the grammar and memoizes each method's result for a given input position. Consequently, a method's code is executed at most once for an input position, ensuring linear time performance even if the parser needs to backtrack. Upon completion, the parser returns either an error describing the mismatched input (Section 6) or the semantic value corresponding to the well-formed input (Section 7).

3. Grammar Specification

At the core of a grammar specification are the productions relating nonterminals to expressions. *Rats!*' productions are of the form:

Attributes Type Nonterminal = *e* ;

The *Attributes* are a space-separated list of zero or more per-production attributes, *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 mostly mirror the operators of parsing expression grammars [20], with extensions to create and manipulate semantic values. The PEG operators are used to specify a language's syntax and are comparable to the familiar EBNF notation [23, 42], including literals, sequences, choices, repetitions, and options. They differ in that choices, repetitions, and options are greedy and in the inclusion of syntactic predicates, which match expressions without consuming them. As discussed in detail in [20], the greediness of PEG operators helps avoid common shortcomings of CFGs, such as the "dangling else" problem or declarations taking precedence over other constructs in C++, by letting grammar writers express ordering constraints directly in a language's grammar. Where greediness is *not* appropriate, syntactic predicates can limits its effects—with the full expressivity of PEGs. For example, the following (slightly simplified) production from *Rats!*' own grammar recognizes a production's attributes:

```
Pair Attributes =
    &(Type Nonterminal "=":Symbol)
      { yyValue = Pair.EMPTY; }
  / a:Attribute as:Attributes
      { yyValue = new Pair(a, as); } ;
```

The and-predicate operator "`&`" followed by the parenthesized expression in the first alternative denotes a syntactic predicate. It requires that any list of attributes be followed by a type, nonterminal, and "`=`" symbol and thus prevents the production from consuming the type and nonterminal, which, like attributes, can be simple names. It also eliminates the need for treating any names as reserved and thus avoids restricting the Java type names appearing in *Rats!*' productions.

Rats!' additional operators are used to manage semantic values. In particular, semantic actions may appear anywhere in a production and, as shown in the example above, define that production's semantic value through an assignment to `yyValue` (so named in deference to Yacc [26]). Bindings assign the semantic value of a component expression to a variable and, as also shown above, make the value available for creating a production's overall value in subsequent actions. An and-predicate operator "`&`" directly followed by a semantic action is interpreted as a semantic predicate, whose code must evaluate to a boolean value and which can be used to restrict expressions based on their values. A string match `"text":`*e* is semantically equivalent to:

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

However, this is a common idiom for matching specific keywords or symbols in PEGs—see, for example, the above production for attributes—and thus directly supported by *Rats!*. Finally, a voided value ignores an expression's semantic value when automatically deducing a production's overall value.

3.1 Determining Semantic Values

While semantic actions provide considerable flexibility in creating a production's semantic value, they are not always necessary, thus cluttering a grammar and making it harder to modify the grammar. For instance, it is often convenient to create productions that consume keywords or punctuation before or after another nonterminal or that combine several nonterminals into a larger choice. In either case, 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 and 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 a semantic action is not only tedious, but can also lead to inefficient or incorrect packrat parsers. The underlying issue 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.

Finally, many tool writers do not require an optimized representation for an input's abstract syntax tree (AST). Rather, a generic tree representation is sufficient and lets the developer focus on providing a tool's functionality instead of first creating an AST representation. To address these issues, *Rats!*, when compared to Ford's Pappy, adds support for easily passing a semantic value through a production, for simplifying lexical analysis through void and text-only productions, and for automatically creating the AST for hierarchical syntax through generic productions. We now discuss these features in turn.

Passing the Value Through

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 `yyValue`. This technique is illustrated in the following production from our Java grammar:

```
String Identifier = yyValue:Word
  &{! JAVA_KEYWORDS.contains(yyValue)} ;
```

The production recognizes identifiers; its semantic value simply is the word representing the identifier, with the semantic predicate ensuring that the word is not a keyword. Second, for many expressions, *Rats!* can automatically deduce the semantic value. As an example, consider this (slightly simplified) production from our C grammar:

```
GNode PrimaryExpression =
    <Identifier>    PrimaryIdentifier
  / <Constant>      Constant
  / <Parenthesized> void:"(":Symbol Expression
                    void:")":Symbol ;
```

Since the first two 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 third alternative is the value of the `Expression` production, as the values of the string match expressions are explicitly voided.

Void and Text-Only 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 or ignored spacing including comments. For example, the following void production from our C grammar recognizes ignored space characters:

```
transient void Space = ' ' / '\t' / '\f' ;
```

The `transient` attribute disables memoization for a production and is explained in Section 8. 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.

A text-only production is a production with a declared type of `String`. Additionally, it must not contain assignments to `yyValue` and may reference only other text-only productions. The semantic value of a text-only production is the text matched in the input. Consequently, text-only productions eliminate the need for explicitly building up strings through Java's string buffers; on a success-

ful match, the implementation simply creates the string from the buffered input characters. Text-only productions are typically used for recognizing identifiers, keywords, and literals. For example, the following text-only production from our Java 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. The "`![\"\\] _`" expression uses a not-followed-by syntactic predicate, which inspects the input but does *not* consume it. In this example, the corresponding parser looks one character ahead, checks whether the input contains a double quote or backslash, and, if not, continues, recognizing any (other) character. The expression is read as "any character but a double quote or backslash".

Generic Productions

A generic production is a production with a declared type of `generic`. Its semantic value is a generic AST node, `GNode`. The generic node has the same name as the production and the values of the component expressions as its children, with the exception of character terminals, void nonterminals, and voided expressions, which are ignored. For example, the following (slightly simplified) generic production from our C grammar recognizes `return` statements:

```
generic ReturnStatement =
  void:"return":Keyword Expression?
  void:";":Symbol ;
```

It is semantically equivalent to the production:

```
GNode ReturnStatement =
  "return":Keyword e:Expression? ";":Symbol
  { yyValue =
      GNode.create("ReturnStatement", e); } ;
```

To pass a value through a generic production, the corresponding alternative explicitly binds `yyValue`, which instructs *Rats!* not to create a `GNode` for that alternative.

Options, Repetitions, and Nested Choices

A final issue that impacts how semantic values are determined is *Rats!*' processing of options, repetitions, and nested choices. Our parser generator, similar to Ford's Pappy, lifts options, repetitions, and nested choices into their own productions (though, nested choices appearing as the last element of a sequence need not be lifted). Furthermore, it desugars options into choices with an empty, second alternative and repetitions into the corresponding right-recursive expressions. The semantic value of the second, empty alternative of a desugared option is `null`. 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. Processing options, repetitions, and nested choices in this way ensures that parser generation is correct and manageable. However, as discussed in Section 8, our parser generator includes several optimizations that avoid the lifting and desugaring wherever possible. While these optimizations increase parser generator complexity, they also reduce runtime overheads.

4. Module System

Rats!' module system leverages the fact that parsing expression grammars are closed under composition and provides the framework for organizing, modifying, and composing syntactic specifications. More specifically, *Rats!* relies on modules to provide encapsulation by grouping related productions, controlling their visibility, and tracking their dependencies. Next, it relies on module modifications to concisely express syntactic extensions. Module modifications specify how one module differs from another by adding, overriding, or removing individual alternatives and, after application, produce a new module that combines the modifying and modified modules. Finally, *Rats!* relies on module parameters to compose different syntactic units and their extensions with each other. Parameters specify module names and allow for the instantiation of a module with different, actual dependencies. Comparable to functors in ML and templates in C++, module modifications and parameterized modules delay the creation of actual syntactic specifications from grammar development time until parser generation time, i.e., when invoking *Rats!* on a grammar's top-level module. They differ in that module modifications affect a module's contents while module parameters affect a module's dependencies.

In general, we expect a grammar writer to use module parameters for *all* dependencies in a newly developed module. That way, the module can be instantiated with modules unforeseen by the developer, thus maximizing reuse. Only a grammar's top-level module does not have parameters and, instead, instantiates all modules with their actual dependencies. The grammar writer uses module modifications when a new syntactic specification represents a (small) delta on an existing specification. Examples include adding a new operator to an existing programming language's expression syntax or restricting a language's features for educational applications. Of course, the module modification's dependencies, including the modified module's name, are specified by module parameters as well, thus maximizing flexibility in applying the module modification.

In detail, a module starts with a module declaration followed by zero or more dependency declarations. The module declaration specifies the module's name, which exists in a global name space, followed by an optional parameter list. Module parameters are treated as module names and replaced with the actual arguments throughout the module on instantiation. For example, the following declaration introduces a parameterized module defining the symbols common to C and Java:

```
module xtc.util.Symbol(Spacing);
```

Modules are named and organized similarly to Java class files, which avoids name clashes even across projects and organizations and facilitates the automatic loading of modules from the file system.

The dependency declarations specify how a module interacts with other modules. *Rats!*' module system supports three types of dependency declarations. First, `import` declarations make another module's productions referenceable from within the current module. Second, `modify` declarations make another module's productions not only referenceable but also modifiable. Each module can modify at most one other module and the closure of modification dependencies cannot be circular. Finally, `instantiate` declarations instantiate parameterized modules and make their names directly available in other dependency declarations. They can also rename a module, thus allowing for multiple instantiations of the same parameterized module within the same grammar. For programmer convenience, the instantiation syntax can also be used in `import` and `modify` declarations. For example, the module `xtc.util.Symbol` declared above imports the module defining spacing, or layout, as following:

```
import Spacing;
```

Note that the actual imported module is only known after `xtc.util.Symbol` has been instantiated by supplying an argument for the `Spacing` parameter.

In contrast to the global name space for module names and to provide encapsulation, nonterminals are only meaningful in relation to a specific module. Without `import` and `modify` declarations, a module can only reference the nonterminals defined in that module. However, in the presence of `import` and `modify` declarations, a module can also reference the imported or modified modules' nonterminals, which may lead to ambiguous references. To resolve such ambiguities, *Rats!* gives precedence to nonterminals that are defined in either the same module as the reference or in a modified module (modifying modules cannot contain a production with the same name as a production in the modified module). Any remaining ambiguities—for example, when the same nonterminal is defined in several imported modules—can be resolved by using a fully qualified nonterminal in a grammar specification. For example, consider the productions defining the symbols common to Java and C in module `xtc.util.Symbol`:

```
String Symbol = SymbolCharacters Spacing ;

transient String SymbolCharacters =
    <GreaterGreaterEqual> ">>="
  / <LessLessEqual>       "<<="
  / <GreaterGreater>      ">>"
  / <LessLess>            "<<"
    /* and so on... */ ;
```

The production recognizing layout after a symbol can be referenced through the unqualified nonterminal "`Spacing`", as shown, or the fully qualified nonterminal "`Spacing.Spacing`". In the latter case, the nonterminal's qualifier is a module parameter and also renamed during module instantiation. Note that the nonterminal `Spacing` must be defined by the imported module, since it is not defined in `xtc.util.Symbol`. Further note that, if module `Spacing` also defines a nonterminal `SymbolCharacters`, the definition in `xtc.util.Symbol` takes precedence, thus avoiding surprises.

To further control the visibility of productions across modules, each production can either be public, protected, or private, as indicated by the corresponding attribute of the same name. A public production is a top-level production and visible outside the generated parser's class. A protected production is internal to a grammar and visible to any importing or modifying module. A private production is internal to a module and only visible to the defining module and any modifying module. We chose to make private productions visible to modifying modules, since module modifications, as discussed below, can make fine-grained changes to existing productions and thus fundamentally alter the syntax recognized by a module. Protected visibility is the default, thus allowing for the omission of the corresponding attribute in the common case, where a production is neither a top-level nor a "helper" production.

Module dependencies are resolved through a breadth-first search starting with a grammar's top-level module, which, since it is directly instantiated, cannot have any parameters. Because of the breadth-first search, a lower-level module's dependencies are only processed *after* all dependencies of the higher-level module have been resolved. As a result, the order of `import`, `modify`, and `instantiate` declarations in a module header does not matter, and mutually dependent modules can be instantiated within the same module. For example, module `xtc.lang.CSpacing` has a parameter for a module defining constants, and `xtc.lang.CConstant` has a parameter for a module defining spacing (or layout). To instantiate these mutually dependent modules, the top-level module `xtc.lang.C` declares:

#	Syntax
1	*Type Nonterminal* `+=` `<`*Name*$_1$`>` *e* `/` `<`*Name*$_2$`>` `...` `;`
2	*Type Nonterminal* `+=` `<`*Name*$_1$`>` `...` `/` `<`*Name*$_2$`>` *e* `;`
3	*Type Nonterminal* `-=` `<`*Name*`>` `;`
4	*Type Nonterminal* `:=` `...` `/` `<`*Name*`>` *e* `;`
5	*Type Nonterminal* `:=` *e* `;`
6	*Attributes Type Nonterminal* `:=` `...` `;`

Table 2. Overview of *Rats!*' module modification syntax. The different modifications (1) add a new alternative before an existing one, (2) add a new alternative after an existing one, (3) remove an alternative, (4) override an alternative with a new expression, (5) override a production with a new expression, and (6) override a production's attributes, respectively. Note that the ellipses are part of the syntax and indicate unmodified expressions.

```
instantiate
  xtc.lang.CConstant(xtc.lang.CSpacing);
instantiate
  xtc.lang.CSpacing(xtc.lang.CState,
                    xtc.lang.CConstant);
```

Rats! processes these two `instantiate` declarations before processing the corresponding `import` declarations in `xtc.lang.CConstant` and `xtc.lang.CSpacing`, which use the supplied arguments. As a result, *Rats!* correctly instantiates the two mutually dependent modules.

Module modifications concisely capture syntactic extensions by adding, overriding, or removing individual alternatives in a production's main choice; they can also override an entire production's expression or attributes. They are expressed as so-called *partial productions*, which specify how to modify existing, full productions and whose syntax is summarized in Table 2. Since partial productions depend on the different alternatives in a production's main choice having sequence names (see Table 1), it is good practice to always name sequences in a grammar specification. A module containing partial productions must contain a single `modify` declaration, which specifies the module to modify. Furthermore, the modified module must define productions with the same names and types as the partial productions appearing in the modifying module, and these productions, in turn, must contain sequences with the same names as the sequence names appearing in the partial productions. For example, module `xtc.lang.JavaSymbol` is implemented as a modification of `xtc.util.Symbol`:

```
module xtc.lang.JavaSymbol(Symbol);
modify Symbol;

String SymbolCharacters +=
    <TripleGreaterEqual>  ">>>="
  / <GreaterGreaterEqual> ... ;
String SymbolCharacters +=
    <TripleGreater>  ">>>"
  / <GreaterGreater> ... ;
```

The module adds Java's unsigned right shift operators to the `SymbolCharacters` production. It is parameterized so that it can be applied to different modules defining symbols and not just `xtc.util.Symbol`. For example, since `xtc.lang.CSymbol` is defined similarly, both module modifications can be applied after each other, resulting in a single module recognizing the symbols for *both* C and Java. Alternatively, both module modifications can be applied separately, resulting in two distinct modules recognizing the symbols for *either* C or Java. When actually applying the module modification, *Rats!* first creates a new module that has the same name as the modifying module and contains all full productions appearing in either the modifying or modified module. It then

applies all partial productions to the full productions, processing them in the same order as they appear in the modifying module.

5. Managing Parsing Context

Mainly due to syntactic predicates, parsing expression grammars can recognize languages not expressible as context-free grammars, such as $\{\texttt{a}^n\texttt{b}^n\texttt{c}^n \mid n > 0\}$ [20]. At the same time, syntactic predicates are not sufficient for recognizing computer formats, including programming languages such as C, that require possibly global state and thus cannot be precisely captured by a grammar alone. In the case of C, the challenge is to distinguish typedef names, that is, type aliases, from object, function, or `enum` constant names, since the two kinds of names yield considerably different language constructs and therefore ASTs. For example, consider the following code snippet (due to [36]):

```
T(*b)[4];
```

If `T` is a typedef name, the snippet declares `b` to be a pointer to an array containing four `T`'s. Otherwise, the snippet accesses the fifth element of the array pointed to by the result of invoking function `T` on `*b` (since C arrays are zero indexed). To more precisely recognize such languages than possible with parsing expression grammars alone, *Rats!* can optionally provide a global state object, which is accessible through the `yyState` variable.

The key issue in supporting such a state object is how to manage state modifications without violating the functional nature of packrat parsers. The obvious solution of invalidating all memoized intermediate results on a state change is impractical: Invalidation requires re-parsing any input already consumed and thus violates the linear-time performance guarantees of packrat parsers. Alternatively, monads provide a general solution for managing (global) state in functional programming languages [41]. But monads are not supported by C-like programming languages and, likely, not familiar to users of *Rats!*. Furthermore, Ford's evaluation of packrat parsers written in Haskell suggests that the monadic version has noticeable performance overheads when compared to those parsers not using monads [18].

Instead, *Rats!* takes a different approach, which preserves linear time performance. Our approach requires that all state modifications are performed within possibly nested, lightweight transactions. Furthermore, if an ordered choice's alternatives may reference the same nonterminals for the same input positions, notably by having a common prefix, the state must be modified in the same way across all alternatives through the common nonterminals. Although not as general as monads, our approach works for programming languages and similarly structured formats for two reasons. First, many programming languages declare constructs that might cause state changes before use. Consequently, the effects of state modifications always flow forward through the input, but never backwards, and previously parsed and memoized expressions need not be invalidated. Second, most programming languages are statically scoped. Consequently, state modifications also have well-defined scopes and can be effectively modeled by nested transactions.

To illustrate the use of lightweight transactions, consider our C grammar. Using lightweight transactions, the production for external declarations can be written as:

```
GNode ExternalDeclaration = {yyState.start(); }
    Declaration             {yyState.commit();}
  / FunctionDefinition      {yyState.commit();}
  / {yyState.abort();} &{ false } ;
```

Since function definitions introduce a new scope, the parser needs to start a new transaction before attempting to parse the alternatives of an external declaration. The corresponding `yyState.start();` statement at the beginning of the production is always executed and, consequently, the transaction covers all three alternatives of the production. The production commits the transaction on a successful recognition of a declaration or function definition and otherwise aborts the transaction while also causing a parse error through the explicit semantic predicate in the third alternative. The transaction must cover the recognition of both declarations and function definitions because they share a common prefix, a sequence of declaration specifiers followed by a declarator. However, the transaction's nested scope for disambiguating names is only "activated" by setting a flag when parsing a function's parameters and body. Names introduced by declarations continue to be added to the outer, top-level scope.

To eliminate the clutter of explicit transactional operations, *Rats!* grammars can utilize a per-production `stateful` attribute, which instructs the parser generator to automatically emit the corresponding code. As an added benefit, the automatically generated code is more efficient because the parser generator can avoid emitting a final alternative that always fails. In general, this attribute should be specified for all productions that might recognize a new language scope. For example, the above production is semantically equivalent to the following production, which represents a simplified version of the one in our C grammar:

```
stateful GNode ExternalDeclaration =
    <Declaration> Declaration
  / <Function>    FunctionDefinition ;
```

The three transactional operations—`start()`, `commit()`, and `abort()`—can easily and efficiently be implemented by pushing and popping pre-allocated context records to and from a stack. For our C grammar's state object, each record contains a hash table recording the different types of names and several flags indicating previously parsed expressions, including `typedef` specifiers. The state object's class exposes methods to access and update this contextual state within a transaction. For example, the (slightly simplified) production for recognizing typedef specifiers in our C grammar reads:

```
generic TypedefSpecifier =
  void:"typedef":Keyword {yyState.typedef();} ;
```

Next, the production for recognizing simple declarators reads:

```
generic SimpleDeclarator =
  id:Identifier { yyState.bind(id); } ;
```

Finally, the production for recognizing typedef names reads:

```
generic TypedefName =
  id:Identifier &{ yyState.isType(id) } ;
```

The `typedef()` method sets a flag in the current context record, indicating that a typedef specifier has been recognized. The `bind()` method updates the context record's hash table with a binding for an identifier. The identifier is marked as a typedef name, if the corresponding flag is set, and, otherwise, as an object, function, or `enum` constant name. The `isType()` method tests whether the identifier represents a typedef name by looking up the binding, starting with the current context record's hash table. Note that a parse time binding is considerably simpler than the corresponding binding created during semantic analysis: it only contains enough information to distinguish between typedef and other names, i.e., the name and a boolean.

In our experience, modifying a global state object through lightweight transactions is sufficiently powerful to recognize the ISO, Kernighan and Ritchie, and GCC dialects of C, including several subtle corner cases. It also allows for a cleaner C grammar than the corresponding LR and LL grammars. In particular, we do

not require separate sets of productions for tracking type specifiers in a sequence of declaration specifiers. This technique is used, for example, by GCC's grammar, noticeably complicates the syntactic specification, and represents an incomplete solution. Additionally, scannerless parsing helps avoid the "lexer hack", which shares a symbol table between a separate lexer and parser and which can lead to subtle bugs in the presence of lexer lookahead [36]. At the same time, global parser state clearly is not a panacea; symbol resolution in general is best left to later compiler phases [15].

6. Error Handling

So far, we have focused on *Rats!*' support for expressing extensible syntax and recognizing well-formed inputs. 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 other grammar and module validity checks, detects and reports left-recursive productions. However, it does automatically convert direct left-recursions in void, text-only, and generic productions into the corresponding right-recursions and includes support for promises to create left-recursive data structures with right-recursive productions. Like Ford's packrat parsers, *Rats!*-generated parsers collect parse errors even for successful parser steps. To illustrate the need for this, consider the Java grammar fragment:

```
Declaration* EndOfFile
```

The `Declaration*` expression succeeds for *any* input. If, however, the input contains a declaration with an embedded syntax error, the grammar fragment fails on the `EndOfFile` expression. If the parser does not track embedded parse errors, it can only generate the not very illuminating error message "end of file expected". 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—either as plain text or hyperlinked HTML—after loading and instantiating all modules, after applying all module modifications, or after applying all optimizations. Second, *Rats!*-generated parsers enforce the type of each production's semantic value by declaring `yyValue` and all bound variables with the corresponding type. As a result, type errors are detected when compiling a parser. Delegating type checking to the compiler represents a practical compromise between safety and parser generator complexity; we believe it reasonable because *Rats!*-generated parsers are carefully formatted for human readability. Third, error messages are automatically deduced from nonterminal names. For example, a parse error within the production for `ReturnStatement` results in the error message "return statement expected". The reported position in the input is the start of the production. For string literals and string matches, which are typically used for recognizing keywords or punctuation, the error message specifies the string and the position of the corresponding expression. Fourth, parsers automatically track file names as well as line and column numbers. If semantic values are instances of our source-to-source transformer framework's AST nodes, *Rats!*-generated parsers can automatically 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.

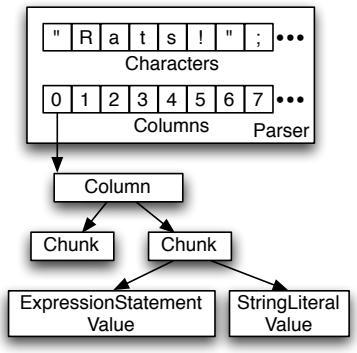

Figure 1. The data structures of a *Rats!*-generated parser. The parser object stores the memoization table in two arrays, one for read-in characters and the other for column objects. Each column object references several chunks, which, in turn, reference memoized results. Each result represents either a semantic value or a parse error.

7. Parser Implementation

The implementation of *Rats!*-generated parsers is guided by two constraints. First, *Rats!* has to correctly implement an essentially functional parsing technique in an imperative, stateful programming language. Second, it has to enable optimizations that reduce the overhead of memoizing intermediate results and thus the runtime overhead of packrat parsing. Our implementation meets both constraints through a careful layout of data structures and a common interface to both semantic values and parse errors. We present the implementation of *Rats!*-generated parsers in this section, while we discuss the corresponding optimizations in the next section.

Each parser generated by *Rats!* has a main class, which inherits from a base class named `PackratParser`. The base class stores the memoization table in two arrays: one array of read-in characters and one of column objects, which store memoized results and track the file name as well as line and column numbers. Compared to Ford's parsers, which rely on a linked list of records representing table columns, and as discussed in Section 8, this layout lets parsers avoid allocating column objects for input positions that do not require memoization. Characters are read, on demand, from a regular Java character stream. While seemingly trivial, this implementation detail avoids an important restriction when compared to Ford's packrat parsers: Ford states [18, 19] 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.

The parser's main class defines the actual column object, which, conceptually, has one field per production to store the memoized intermediate result. In practice and as also discussed in Section 8, *Rats!* uses a level of indirection by allocating fields in chunks. Chunks reduce heap utilization because most nonterminals are not visited for a given input position and, consequently, do not require a field to store the memoized result. Figure 1 illustrates the data structures used by *Rats!*-generated parsers.

The parser's main class also defines a per-production accessor method, which takes the index representing the current input position as its only argument and returns the corresponding result. Since the input to be parsed is implicitly available through the array of characters, accessor methods represent functions from strings to indices to results: `String -> Index -> Result`. On invocation, an accessor method tests whether the column object for that index has been allocated, creating it if necessary, and then tests whether the appropriate chunk has been allocated, again creating it if nec-

```
public abstract class Result {
  // The index into the input.
  public final int index;

  // Create a new result.
  public Result(int idx) { index = idx; }

  // 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();

  // Select the more specific parse error.
  public abstract ParseError select(ParseError e);

  // Create a new value, using this result's index.
  public abstract SemanticValue
    createValue(Object val, ParseError e);

  // Determine whether this result has the value.
  public abstract boolean hasValue(Object val);
}
```

Figure 2. The common base class for semantic values and parse errors. The last three methods are used to optimize parser performance and are explained in Section 8.

essary. Next, it tests whether the memoization field is `null`. If so, the method calculates the result, stores it, and returns it. If not, the method simply returns the stored value.

When receiving results from accessor methods, 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 semantic values and parse errors through two separate container classes. The container class for semantic values, named `SemanticValue`, stores the actual value, the index representing the input after the parsed expression, and possibly an embedded parse error. The field for the actual value is declared to be a Java `Object`, which is an application of type erasure [7] 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 the index representing the location of the error.

Both container classes have a common base class, which provides a uniform interface for distinguishing between values and errors as well as for accessing the contained information. It is shown in Figure 2. The implementation of the concrete methods is very simple—between one and two lines of code per method. At the same time, the use of this common base class significantly simplifies parser implementation, as illustrated in Figure 3 for the production recognizing pointer declarators in our C grammar:

```
generic PointerDeclarator =
  Pointer DirectDeclarator ;
```

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 (which also ensures the type safety of the semantic

```
private Result pPointerDeclarator(final int yyStart)
  throws IOException {
  Result     yyResult;
  GNode      yyValue;
  ParseError yyError = ParseError.DUMMY;

  yyResult = pPointer(yyStart);
  yyError  = yyResult.select(yyError);
  if (yyResult.hasValue()) {
    GNode v$g$1 = (GNode)yyResult.semanticValue();

    yyResult = pDirectDeclarator(yyResult.index);
    yyError  = yyResult.select(yyError);
    if (yyResult.hasValue()) {
      GNode v$g$2 = (GNode)yyResult.semanticValue();

      yyValue = GNode.create("PointerDeclarator",
                             v$g$1, v$g$2);
      setLocation(yyValue, yyStart);
      return yyResult.createValue(yyValue, yyError);
    }
  }
  return yyError;
}
```

Figure 3. Example code for recognizing pointer declarators in our C parser. The method body attempts to match a `Pointer` followed by a `DirectDeclarator`. If the matches are successful, it creates an AST node with the two productions' values as its children and returns a new value container with the node and any embedded parse error. The method implicitly tracks the current input position through `yyResult`, while explicitly tracking any parse error through `yyError`. Note that the method's result is not memoized, since the corresponding production is only referenced once in the C grammar and thus cannot be visited more than once for a given input position.

actions). Consequently, they could be compiled away in languages, such as C++, that support static casts.

8. Optimizations

We now turn to the optimizations performed by *Rats!*. The primary 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 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 are scannerless, integrating lexical analysis with parsing, and thus cannot utilize well-performing techniques, such as DFAs, for recognizing tokens.

Table 3 summarizes the optimizations performed by *Rats!*. The optimizations include all those performed by Ford's Pappy, while also introducing significant new ones. The chunks, grammar, terminals, and cost optimizations, which are also performed by Pappy, work as following. First, the *chunks* optimization is based on the observation that most nonterminals are not visited for a given input position, with the table field memoizing the production's result remaining `null`. Consequently, as illustrated in Figure 1, the chunks optimization introduces a level of indirection and allocates fields in chunks, thus reducing heap utilization. Second, the *grammar* optimization is based on the observation that the lifting and desugar-

Name	Description	*Rats!*
Chunks	Organize memoized fields into chunks.	Same as [18]
Grammar	Fold duplicate productions and eliminate dead productions.	Same
Terminals	Optimize recognition of terminals, incl. using switch statements.	Improved
Cost	Perform cost-based inlining.	Same
Transient	Do not memoize transient productions.	New
Nontransient	Automatically recognize productions as transient.	New
Repeated	Do not desugar transient repetitions.	New
Left	Implement direct left-recursions as repetitions, not recursions.	New
Optional	Do not desugar options.	New
Choices1	Inline transient void and text-only productions into choices.	New
Choices2	Inline productions that are marked `inline` into choices.	New
Errors	Avoid creating parse errors for embedded expressions.	New
Select	Avoid accessor for tracking most specific parse error.	New
Values	Avoid creating duplicate semantic values.	New
Matches	Avoid accessor for string matches.	New
Prefixes	Fold common prefixes.	New
GNodes	Specialize generic nodes with a small number of children.	New

Table 3. Overview of optimizations. *Rats!* introduces significant new optimizations when compared to Ford's previous work.

ing of expressions described in Section 3.1 can result in duplicate productions, which might even *increase* the size of the memoization table. Consequently, the grammar optimization folds equivalent productions into a single one and, comparable to dead code elimination, eliminates non-top-level productions that are never referenced. Third, the *terminals* optimization is based on the observation that many productions for recognizing lexical syntax have alternatives that start with different characters. Consequently, the terminals optimization replaces successive `if` statements that parse disjoint lexical alternatives with a single `switch` statement. It also folds alternatives that start with the same literals into one alternative with a common prefix and, if the matched text can be determined statically, uses literal Java strings instead of dynamically instantiating strings. 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.

Of the new optimizations introduced by *Rats!*, the *transient* optimization is the most important. It is based on the observation that packrat parsers never backtrack over many productions. For example, many helper productions for recognizing lexical syntax, such as the `SymbolCharacters` production shown in Section 4 and including those for identifiers, keywords, operators, and punctuation, do not require backtracking. More importantly, spacing, which includes all white space and comments, makes up large parts of most source files but does not require backtracking. Finally, many productions for hierarchical syntax start with distinct keywords or symbols and do not require backtracking either. However, if the parser can never backtrack over a production, there also is no need to memoize the intermediate result. Consequently, the transient optimization gives grammar writers control over which productions are memoized. If a production is declared to be `transient`, *Rats!* does not allocate a field for memoizing the production's result; rather, as illustrated in Figure 3, the corresponding parsing code is always executed. As a result, transient productions reduce the size of a parser's column objects and chunks. Furthermore, for input positions that are only touched by transient productions (such as those in the middle of a token), no results need to be memoized at all and the corresponding column object is never allocated, which further reduces heap utilization.

Several other optimizations build on the transient optimization and its motivating observation. In particular, the *nontransient* optimization automatically marks productions as transient, thus eliminating the need for grammar writer intervention. Currently, it recognizes productions that are only referenced once in a grammar, but on-going work is trying to extend the scope of this optimization. The *repeated* optimization is based on the observation that a repetition's component expressions do not require memoization if the repetition appears in a transient production—after all, the transient attribute already indicates that the parser will not backtrack over the production. Consequently, the repeated optimization preserves repetitions in transient productions and directly implements them in the generated parser; it improves not only performance but also avoids stack overflow errors for some Java virtual machines when recursing over very long repetitions. The *left* optimization is based on the observation that direct left-recursions are typically used for recognizing expressions. Since each type of expression has a unique operator, the parser does not backtrack for the different alternatives, and the left optimization converts direct left-recursions into equivalent repetitions instead of right-recursions. The *optional* optimization is based on the observation that a packrat parser never backtracks over the alternatives of a desugared option and thus preserves options for all productions, whether they are transient or not.

The *choices1* optimization is based on two observations. First, the effectiveness of the terminals optimization depends to some degree on how a grammar has been written: If an alternative references a nonterminal instead of the corresponding literals, it cannot include the alternative in a `switch` statement. Second, transient productions can be safely inlined into other productions, since they do not require memoization. Consequently, if a nonterminal appears as the only expression in an alternative, the nonterminal references a void or text-only production (which is typically used for lexical syntax), and the referenced production is transient, the choices1 optimization inlines the production. The *choices2* optimization generalizes the choices1 optimization to inlining all transient productions, not just void or text-only productions, with the goals of (1) avoiding the overhead of invoking accessor methods and (2) creating opportunities for other optimizations. However, experiences with a first implementation showed that it can be too aggressive. Notably, for several productions recognizing expressions in our C and Java grammars, it inlined the same production multiple times into another production. The corresponding parser methods are very large and thus hard to compile by the just-in-time compiler of the Java virtual machine, resulting in a noticeable *decrease* in parser performance. Consequently, to better control this optimization, we introduce the `inline` attribute, which makes a production available for inlining through the choices2 optimization and is otherwise equivalent to the `transient` attribute. It is typically used to mark productions that are referenced through a lone nonterminal in a larger choice, such as a primary identifier appearing in only the production recognizing all primary expressions or a return statement appearing only in the production recognizing all statements.

The *errors* optimization is based on the observation that most alternatives in a production's main choice fail on the first expression. For example, the statement production for any C-like language has a large number of alternatives, but only one of these alternatives can succeed on a given input. 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 *all* alternatives fail. The *select* optimization is based on the observation that tracking the most specific parser error requires two method invocations, one to access a result's parse error and one to compare that error with the most specific one. To reduce the overhead of this common idiom, the select optimization performs access and comparison with a single invocation of the `select()` method defined in Figure 2 and used in Figure 3.

The *values* optimization is based on the observation that many productions simply pass the semantic value through. For example, both our C and Java grammars have 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. Consequently, the values optimization delegates the creation of new instances of `SemanticValue` to the last result accessed while parsing an expression, i.e., to the `createValue()` method defined in Figure 2 and used in Figure 3. This dynamic delegation of object creation works independently of how a semantic value has been created, be it through an explicit action, an assignment to `yyValue`, or through *Rats!*' value deduction facilities. As a result, this optimization applies to a much larger class of parsing expressions than would be possible with a static analysis in the parser generator. The *matches* optimization, comparable to the select optimization, replaces two method invocations for implementing string matches with a single method invocation of the `hasValue(Object)` method defined in Figure 2.

The *prefixes* optimization generalizes the terminals optimization by extending the folding of alternatives with common prefixes to nonterminals, and not just literals; it is comparable to left factoring for context-free grammars. The goals are (1) to avoid repeated calls to accessor methods when trying the alternatives with a common prefix and (2) to avoid memoization altogether by reducing the number of nonterminal references.

Finally, the *gnodes* optimization is based on the observation that most AST nodes only have a small number of children. In particular, all productions in our C and Java grammars that do not contain repetitions and thus have a fixed number of component expressions result in AST nodes with at most seven children. Furthermore, many productions that do contain repetitions, such as those recognizing declaration specifiers in C or modifiers in Java, result in AST nodes with few children in practice. Consequently, the gnodes optimization introduces several versions of our generic node abstraction that are specialized for a particular number of children instead of relying on a variable length list. If the number of children is known at parser generation time, the parser invokes the corresponding factory method directly (see Figure 3 for an example). If the number is not fixed, the parser invokes another factory method, which dynamically selects either a specialized or the general version.

9. Experimental Evaluation

In this section, we present the results of our experimental evaluation. First, we evaluate *Rats!*' support for syntactic extensibility in Section 9.1 by determining the effort involved in creating three real-world language extensions. Second, we evaluate parser performance in Section 9.2 by comparing *Rats!* with four other parser generators. In summary, our results show that *Rats!* does, in fact, provide concise syntactic specifications, with grammars being considerably shorter than other grammars, and easy extensibility, with extensions being realizable by others with little code and effort. *Rats!* also produces parsers that perform reasonably well, out-performing two GLR-based parsers by at least a factor of 1.9, while being at most 2.7 times slower than more conventional LL parsers.

9.1 Extensions

To evaluate *Rats!*' support for extensible syntax, we present three language extensions that have been implemented on top of the C and Java grammars distributed with our parser generator. The first extension is C4 [17] (for CrossCutting C Compiler), which enhances C with support for aspect-oriented programming techniques. C4's goal is to simplify the development of system software variants, and it has already been used for implementing Linux kernel extensions. C4's syntax specification comprises four modules—in addition to the 11 modules of the C grammar—with 150 lines of code. It also requires a subclass of the C parser's global state class to support aspect scopes spanning several disjoint aspect blocks; the class adds 130 lines of Java code. A novice *Rats!* user took 4 hours to learn how to use our parser generator, 11 hours to write and debug the original grammar, and 2 hours to port the grammar over to the newly added module system.

The second extension is Jeannie, which significantly simplifies the development of Java programs that also include native C methods. Basically, Jeannie enables the direct embedding of C code in Java source files, while also providing sugar for conveniently accessing Java objects from the embedded C code. For example, using Java's native interface, a developer needs to write the following code to call a Java method from C:

```
jclass cls = (*env)->GetObjectClass(env, obj);
jmethodId mid = (*env)->GetMethodId(env, cls,
  "method", "signature");
jobject result = (*env)->CallObjectMethod(env,
  cls, mid, ...);
```

However, using Jeannie, the developer simply writes:

```
jobject result = `obj.method(...);
```

The backtick "`" is a new unary operator that switches between C and Java for the following expression. Jeannie's grammar combines the complete C and Java grammars with the syntax for switching between the two languages and the sugar for calling back to Java. The syntax specification comprises four modules with 230 lines of code. Another novice *Rats!* user took 20 hours to familiarize himself with packrat parsers in general and *Rats!* in particular, 5 hours to understand the existing C and Java grammars, and 20 hours to write and debug the Jeannie grammar. We believe that the difference in development time when compared to C4 stems mostly from the fact that combining two entire languages is considerably more difficult than adding select new constructs to a language.

Our third syntactic extension removes pointers from C and thus exposes a simplified language that is more suitable for introductory programming courses. It also illustrates *Rats!*' support for disabling syntax. Barring the single module's header, the syntactic specification consists of only *four* partial productions:

```
GNode Declarator -= <Pointer> ;
generic AbstractDeclarator :=
  DirectAbstractDeclarator ;
GNode UnaryExpression -= <Address>,
  <LabelAddress>, <Indirection> ;
Action ComponentSelectionExpression -=
  <Indirect> ;
```

System	Algorithm	Modules	Lex	AST	LoC
Rats!	PEG	9	—	—	790
SDF2	GLR	57	—	—	1,680
Elkhound	LALR/GLR	1	1	1	2,370
ANTLR	LL	1	1	—	1,280
JavaCC	LL	1	1	—	1,240

Table 4. Comparison between Java grammars. "Algorithm" indicates the underlying formalism, "Modules" indicates the number of grammar units, "Lex" indicates a separate lexer, "AST" indicates a separate AST definition, and "LoC" stands for "lines of code", which, for Elkhound, also includes several necessary C++ files.

These productions remove support for pointers from (abstract) declarators and support for address and indirection operators from expressions. No other changes are necessary, and the corresponding source-to-source transformer can simply pretty print the resulting AST with the existing C pretty printer, handing off the resulting code to a conventional C compiler.

9.2 Performance

Our performance evaluation compares Java 1.4 recognizers and parsers generated by *Rats!* 1.8.0, SDF2 2.3.3 [9, 40], Elkhound 2005.08.22b [28], ANTLR 2.7.5 [34], and JavaCC 4.0 [16]. We also use the `sglri` tool from pre-release 14567 of Stratego/XT [8], as SDF2's `sglr` tool has known inefficiencies—though both rely on the same parser engine. SDF2 is a modular GLR parser generator written in and targeted at C, Elkhound is a hybrid GLR and LALR(1) parser generator written in and targeted at C++, and ANTLR and JavaCC are LL(k) parser generators written in and targeted at Java. We utilize our own grammar for *Rats!*, the java-front 0.8 grammar for SDF2, but with all Java 1.5 features removed, our own grammar for Elkhound, which largely mirrors our *Rats!* grammar for syntax and Elkhound's C++ grammar for AST structure and supporting code, the version 1.21 grammar for ANTLR, and the grammar dated 5/5/2002 for JavaCC. Table 4 summarizes the five grammars.

We present results for both the performance of Java recognizers, i.e., parsers that do not generate ASTs, and full Java parsers, thus enabling us to identify the overheads associated with each parser generator's AST framework. The *Rats!*-, SDF2-, and ANTLR-generated parsers rely on generic AST nodes, while the Elkhound- and JavaCC-generated parsers rely on specialized classes generated for each type of node. The recognizer for *Rats!* is generated from our Java grammar based on a parser generation time flag. The recognizers for SDF2 and Elkhound simply do not execute semantic actions, as specified by a runtime flag. The recognizer for ANTLR is generated from a separate grammar that omits the AST annotations. Finally, the grammar and AST classes for the JavaCC-based parser are automatically generated from the recognizer grammar with Java Tree Builder 1.2.2 [33].

As inputs, we use a sampling of 38 Java source files taken from the Cryptix libraries [12], ANTLR's sources, and *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 730 methods with 8,058 non-commenting source statements. All measurements were performed on a consumer-level Apple iMac from the fall of 2002, with an 800 MHz PowerPC G4 processor and 1 GB of RAM, running Mac OS X 10.4.5 and Apple's port of Java 1.4.2 update 2. For each input file, we measured 10 iterations over that file after 2 warm-up runs. We then derived overall throughput and heap utilization statistics by performing a least-squares-fit over the averages for each file. Note that heap utilization reflects total memory pressure, as we allocate a heap large enough to avoid garbage collection while parsing. Further note that all input files and benchmark code are part of *Rats!*' distribution, so that experiments are readily repeatable.

	Recognizer		Parser	
System	T-put	Heap Util.	T-put	Heap Util.
Rats!	518.0	51.5	317.0	58.0
SDF2	136.1	—	21.4	—
Elkhound	141.5	—	139.4	—
ANTLR	538.6	11.5	393.6	28.0
JavaCC	1,114.3	10.6	382.9	63.2

Table 5. Comparison between Java recognizers and parsers, i.e. parsers that do not and that do generate an AST. Throughput ("T-put") is measured in KB/s and heap utilization is measured in bytes of heap per byte in the input. The recognizers and parsers generated by SDF2 and Elkhound do not report heap utilization. The corresponding grammars are summarized in Table 4.

Table 5 shows the results for our experiments. The performance numbers illustrate the cost of memoization and therefore the cost of relying on PEGs to enable modularity, as the *Rats!*-generated recognizer and parser are up to 2.2 times slower and consume up to 4.8 times more heap than the corresponding LL-based recognizers and parsers. But they also illustrate that our optimizations are effective, as the *Rats!*-generated recognizer and parser consistently out-perform the GLR-based recognizers and parsers by at least a factor of 2.3.

The relatively poor performance of the SDF2- and JavaCC-generated parsers relative to the corresponding recognizers reflects the costs of effectively creating a parse tree instead of just creating an abstract syntax tree. In the case of SDF2, the parser engine always creates a parse tree, from which the AST is extracted based on embedded annotations. In the case of JavaCC, Java Tree Builder is too aggressive in automatically including all of a production's component expressions. In contrast, Elkhound's AST generation is seemingly for free. However, our Elkhound-generated parser, just like Elkhound's C++ parser, does not free memory allocated for AST nodes and thus leaks memory when rejecting unsuccessful alternatives. A more careful implementation would incur additional overhead for properly managing memory.

Additional experiments show that performance is somewhat dependent on inputs. Notably, when adding three very large source files, ranging from 123.7 KB to 134.5 KB, to our input set, the *Rats!*-generated parser's throughput drops to 269.2 KB/s while the ANTLR-generated parser's throughput increases to 419.6 KB/s. Overall, the *Rats!*-generated recognizer and parser are up to 2.7 times slower than the LL-based recognizers and parsers and outperform the GLR-based recognizers and parsers by at least a factor of 1.9. The additional results suggest that ANTLR-generated parsers have a relatively high startup cost, while *Rats!*-generated parsers become increasingly memory-bound due to ever larger memoization tables. In fact, our *Rats!*-generated C parser avoids this limitation by incrementally parsing each external declaration and then discarding the memoization table. However, this technique does not apply to Java sources, which typically contain exactly one top-level class or interface declaration.

At 790 lines, our Java grammar is concise and largely follows the language specification, while the ANTLR and JavaCC grammars are more than 50% larger, the SDF2 grammar is more than 2 times larger, and the complete Elkhound specification is almost 3 times larger. Furthermore, our grammar requires neither explicit lookahead specifications—as required for the ANTLR and JavaCC grammars—nor disambiguation filters [39]—as required for the SDF2 grammar—nor explicit merge functions—as required for the Elkhound grammar. In fact, Elkhound's hybrid algorithm results in the reporting of 18 shift/reduce and 3 reduce/reduce conflicts, even

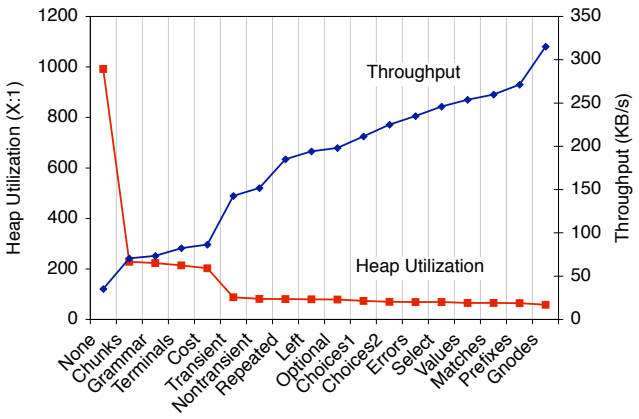

Figure 4. Effects of individual optimizations on throughput and heap utilization of *Rats!*-generated Java parser.

though only two conflicts can yield multiple results and require explicit disambiguation.

Figure 4 illustrates the effects of *Rats!* optimizations on parser throughput and heap utilization. The individual optimizations listed on the x-axis are summarized in Table 3 and are cumulative from left to right. For example, the data points labelled "Terminals" include the chunks, grammar, and terminals optimizations. The figure shows that, of the factor 8.9 improvement in throughput between no and all optimizations, the optimizations also performed by Ford's Pappy contribute a factor 2.44 improvement and the newly added optimizations a factor 3.64 improvement. Of the factor 17.1 improvement in heap utilization, the optimizations also performed by Ford's Pappy contribute a factor 4.9 improvement and the newly added optimizations a factor 3.5 improvement. These results illustrate that the newly added optimizations have a significant impact on resource utilization, especially for parser throughput, and represent a clear improvement over Ford's work. At the same time, the chunks optimization remains the most important optimization for reducing heap utilization.

10. Related Work

Motivated by the observation that traditional LR and LL grammars are too hard to extend, several previous efforts have explored how to make syntactic specifications more easily modifiable. Notably, Cardelli et al. [11] introduce the idea of incremental grammar definitions, which can add, override, or remove individual alternatives. They also define a formal type system for their grammars and provide an LL-based implementation. PPG [10], which is the parser generator for the Polyglot extensible compiler [32], provides similar functionality but is implemented as a frontend to an LALR(1) parser generator. Next, ANTLR [34] is an LL(k) parser generator, models grammar modifications as inheritance, and provides the ability to add or override alternatives. Furthermore, to improve expressivity, it is the first system to utilize syntactic predicates. However, all three systems are fundamentally limited by their choice of parsing algorithm and cannot support the composition of arbitrary syntactic units.

To avoid this limitation, Bison [21], a direct Yacc replacement, now supports GLR parsing as an alternative to LALR(1). Elkhound [28] also supports GLR parsing, but, to improve recognition speed, generates hybrid parsers that fall back on LALR(1) for unambiguous productions. However, parsers generated by either system also preserve ambiguities and can produce several parse or abstract syntax trees, which then need to be explicitly disambiguated through code. In fact, Elkhound's parsers need to treat semantic values as functional, as they may be shared between trees, and could thus benefit from *Rats!*' technique for safely updating contextual state. In contrast, `metafront` [6] employs a new parsing algorithm called specificity parsing. The algorithm recognizes all CFGs (modulo left-recursions) and always returns a single tree by (1) giving precedence to the more "specific" nonterminal, with specificity being derived from a grammar, and (2) relying on explicit syntactic predicates for any remaining ambiguities. While effective, we prefer to build on PEGs, as they avoid ambiguities in the first place and also have a stronger formal foundation [20].

SDF2 [9, 40] shares many of the same goals as our work and also provides a module system for syntactic specifications. However, its module system is less expressive, supporting only the addition of new alternatives but not their overriding or removal. It also lacks module parameters, which fixes module dependencies at module definition time and not module instantiation time, thus limiting module reuse. Furthermore, because SDF2 builds on a GLR parser generator, grammars require disambiguation, which is provided by separately specified disambiguation filters [39]. In contrast, *Rats!*' syntactic predicates are an integral part of the underlying syntactic formalism and also more expressive than the follow restrictions and reject productions utilized by SDF2. Additionally, priority and associativity can be directly encoded in a grammar; though *Rats!* does not support left-recursions in general and instead requires explicit semantic actions to construct the corresponding left-recursive data structures through promises.

Many programming languages with macro support include at least some facilities for extending a language's syntax—[5] surveys several representative systems. However, these facilities usually are tightly integrated with the language itself, interleaving parsing, macro expansion, and possibly type checking, and are thus unsuitable for extensible syntax in general. Furthermore, modules have been studied extensively in the context of programming languages [2]. We build on this body of work, notably the basic idea behind ML's functors, and adapt modules to a new domain, syntactic specifications. Finally, Dijkstra and Swierstra [13] explore how to implement lazy, functional parser combinators in Java. However, their direct (and manual) mapping of Haskell's parser combinators onto Java objects is relatively verbose and slow, while also lacking *Rats!*' support for global state.

11. Conclusions

In this paper, we have presented *Rats!*, a parser generator for Java that supports easily extensible grammars by making the benefits of modularity available for syntactic specifications. To this end, *Rats!* eschews context-free grammars and instead builds on parsing expression grammars. PEGs are attractive because they are closed under composition—thus enabling the composition of syntactic units, rely on ordered rather than unordered choices—thus avoiding ambiguities, provide syntactic predicates—thus increasing expressivity, and are scannerless by default—thus integrating lexical analysis. *Rats!* implements the corresponding packrat parsers, which are lazy and functional, in a strict, imperative language through a careful layout of data structures and a common interface to semantic values and parse errors. It also improves on Ford's previous work by exposing more concise grammars—through the automatic deduction of semantic values, supporting global state—which is safely modified through lightweight transactions, and producing better performing parsers—through extensive optimizations.

On top of this syntactic foundation, *Rats!* builds a module system for grammar specifications. The module system organizes grammars into modules, which encapsulate related productions and explicitly track dependencies on other modules. It also supports so-called module modifications, which can add, override, or remove individual alternatives in productions and thus concisely

express syntactic extensions. Finally, it relies on module parameters to compose different modules with each other, notably by delaying the specification of actual module dependencies until module instantiation time. Our experimental evaluation validates *Rats!* as a substrate for extensible syntax. It enables others to realize real-world programming language extensions in little time and code, and its parsers perform reasonably well, out-performing GLR-based parsers by at least a factor of 1.9, while being at most 2.7 times slower than more conventional LL parsers.

Besides further developing the C4 and Jeannie systems discussed in Section 9.1, our future work will focus on two issues. First, we will investigate how to improve the scope of the nontransient optimization, so that it can automatically recognize more productions as `transient` and, where possible, also as `inline`, thus reducing the need for manual annotation and increasing opportunities for *Rats!*' other optimizations. Second, we will explore how to leverage essentially the same syntactic specification for generating parsers that create different ASTs, thus further improving grammar reuse. For example, while a parser used in a compiler can safely discard all layout including comments and thus create a relatively compact AST, a parser used in a software refactoring tool must preserve layout, so that the tool can emit properly formatted sources again. At the same time, both parsers should be based on the same language specification. The open source release of our parser generator is available at `http://cs.nyu.edu/rgrimm/xtc/`.

Acknowledgments

Martin Hirzel implemented the Jeannie grammar and Marco Yuen implemented the C4 grammar. We thank Martin Bravenboer and Scott McPeak for helping to evaluate SDF2 and Elkhound, respectively. We also thank Benjamin Goldberg, Martin Hirzel, Bryan Ford, Trevor Jim, and the anonymous reviewers for their feedback and discussions. This material is based in part upon work supported by the National Science Foundation under Grant No. 0448349.

References

[1] J. Aycock and R. N. Horspool. Practical Earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

[2] J. Bender. Mini-bibliography on modules for functional programming languages. `http://readscheme.org/modules/`, Mar. 2004.

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

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

[5] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 31–40, Jan. 2002.

[6] C. Brabrand, M. I. Schwartzbach, and M. Vanggaard. The `metafront` system: Extensible parsing and transformation. *Electronic Notes in Theoretical Computer Science*, 82(3):592–611, Dec. 2003.

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

[8] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2005.

[9] M. Bravenboer and E. Visser. Concrete syntax for objects. In *Proc. 2004 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 365–383, Oct. 2004.

[10] M. Brukman and A. C. Myers. PPG: A parser generator for extensible grammars. `http://www.cs.cornell.edu/Projects/polyglot/ppg.html`.

[11] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Tech. Report 121, DEC SRC, Feb. 1994.

[12] Cryptix Foundation. Cryptix JCE. `http://www.cryptix.org/`.

[13] A. Dikjstra and D. S. Swierstra. Lazy functional parser combinators in Java. In *Proc. 1st Workshop on Multiparadigm Programming with Object-Oriented Languages*, pp. 11–42. John von Neumann Institute for Computing, May 2001.

[14] J. C. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, Feb. 1970.

[15] T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Proc. 18th European Conference on Object-Oriented Programming*, vol. 3086 of *LNCS*, pp. 147–171. Springer, June 2004.

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

[17] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. `patch (1)` considered harmful. In *Proc. 10th Workshop on Hot Topics in Operating Systems*, pp. 91–96, June 2005.

[18] B. Ford. Packrat parsing: A practical linear-time algorithm with backtracking. Master's thesis, MIT, Sept. 2002.

[19] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proc. 2002 ACM International Conference on Functional Programming*, pp. 36–47, Oct. 2002.

[20] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proc. 31st ACM Symposium on Principles of Programming Languages*, pp. 111–122, Jan. 2004.

[21] Free Software Foundation. Bison. `http://www.gnu.org/software/bison/`.

[22] R. Grimm. Systems need languages need systems! In *Proc. 2nd ECOOP Workshop on Programming Languages and Operating Systems*, July 2005.

[23] ISO. Information technology—syntactic metalanguage—extended BNF. ISO/IEC Standard 14977, Aug. 1996.

[24] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the Prolac protocol language. In *Proc. 1999 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 3–13, Aug. 1999.

[25] K. Lee, A. LaMarca, and C. Chambers. HydroJ: Object-oriented pattern matching for evolvable distributed systems. In *Proc. 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 205–223, Oct. 2003.

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

[27] D. Mazières. A toolkit for user-level file systems. In *Proc. 2001 USENIX Annual Technical Conference*, pp. 261–274, June 2001.

[28] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proc. 13th International Conference on Compiler Construction*, vol. 2985 of *LNCS*, pp. 73–88. Springer, Mar. 2004.

[29] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, pp. 17–30, Oct. 2000.

[30] T. Millstein. Practical predicate dispatch. In *Proc. 2004 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 345–364, Oct. 2004.

[31] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, pp. 228–241, Jan. 1999.

[32] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International*

Conference on Compiler Construction, vol. 2622 of *LNCS*, pp. 138–152. Springer, Apr. 2003.

[33] J. Palsberg, K. Tao, and W. Wang. Java tree builder. `http://compilers.cs.ucla.edu/jtb/`.

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

[35] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proc. 1st ACM/USENIX Symposium on Networked Systems Design and Implementation*, pp. 267–280, Mar. 2004.

[36] J. Roskind. Parsing C, the last word. `http://groups-beta.google.com/group/comp.compilers/msg/c0797b5b668605b4`, Jan. 1992.

[37] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Proc. 1989 ACM Conference on Programming Language Design and Implementation*, pp. 170–178, June 1989.

[38] M. Tomita, ed. *Generalized LR Parsing*. Kluwer, 1991.

[39] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Proc. 11th International Conference on Compiler Construction*, vol. 2304 of *LNCS*, pp. 143–158. Springer, Apr. 2004.

[40] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sept. 1997.

[41] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, vol. 925 of *LNCS*, pp. 24–52. Springer, 1995.

[42] 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.