Declarative Syntax Tree Engineering*

Or, One Grammar to Rule Them All

Robert Grimm

Technical Report TR2007-905
New York University
rgrimm@cs.nyu.edu

Abstract. Grammars for many parser generators not only specify a language's syntax but also the corresponding syntax tree. Unfortunately, most parser generators pick a somewhat arbitrary combination of features from the design space for syntax trees and thus lock in specific trade-offs between expressivity, safety, and performance. This paper discusses the three major axes of the design space—specification within or outside a grammar, concrete or abstract syntax trees, and dynamically or statically typed trees—and their impact. It then presents algorithms for automatically realizing all major choices from the same, unmodified grammar with inline syntax tree declarations. In particular, this paper shows how to automatically (1) extract a separate syntax tree specification, (2) embed an abstract syntax tree within a concrete one, and (3) infer a strongly typed view on a dynamically typed tree. All techniques are implemented in the *Rats!* parser generator and have been applied to real-world C and Java grammars and their syntax trees.

1 Introduction

Parser generators convert a concise specification of a language's syntax, the grammar, into executable code, the parser. As a result, they automate much of the process of parser writing and thus simplify the development of language processing tools. However, just recognizing a language's syntax is not sufficient, and parsers usually generate some syntax tree representation, which is consumed by later phases of a language processing tool. Consequently, many parser generators—including ANTLR [1], Elkhound [2], JavaCC with the JJTree [3] or JTB [4] preprocessors, SableCC [5, 6], SDF2 [7, 8], and my own *Rats!* [9]—go beyond the semantic actions familiar from Yacc [10], support the declarative specification of syntax trees, and automatically include the corresponding action code in generated parsers. Again, this automates much of the process of implementing syntax trees and thus simplifies tool development.

The design space for declarative syntax tree specifications spans three major axes: The syntax tree can be declared inline within the grammar or separately;

* This material is based upon work supported by the National Science Foundation under Grants No. CNS-0448349 and CNS-0615129. Thanks to Martin Hirzel for his feedback on earlier versions of this paper.

it can be concrete (i.e., a parse tree) or abstract; and it can be dynamically typed or statically typed. Each choice in this design space reflects a particular trade-off between expressivity, safety, and performance. However, none of the above parser generators (besides *Rats!*) supports all options in the design space. And, if a parser generator supports a choice along an axis, it typically requires modifications to the grammar. As a result, the re-use of grammars is curtailed: depending on a project's requirements, developers either need to use a different parser generator or a differently written grammar. At the same time, grammar re-use is highly desirable, since developing and maintaining the grammar for a real-world language requires a significant effort.

To improve the flexibility of parser generators and facilitate the re-use of grammars, this paper shows how to automatically realize all options in the design space from the same unmodified grammar and syntax tree specification. The starting point is an inline, abstract, and dynamically typed syntax tree. The paper then presents algorithms for automatically (1) extracting a separate syntax tree specification to serve as documentation for tool developers, (2) embed an abstract syntax tree within a concrete one to facilitate source refactorings, and (3) infer a strongly typed view on a dynamically typed tree to improve static safety. All techniques are implemented in the *Rats!* parser generator and integrated with the xtc toolkit for source-to-source transformers [11]. They have also been applied to real-world grammars for C and Java. At the same time, they do not depend on features unique to *Rats!* or xtc and can thus be implemented in other parser generators as well.

2 Design Space and Related Work

As summarized in Table 1, the design space for syntax trees spans three major axes, which entail trade-offs between expressivity, safety, and performance. The first axis captures whether the syntax tree specification is inline within a grammar or separate. An inline specification certainly simplifies the grammar writer's task, since she needs to describe how to map a language's syntax onto its tree. Consequently, most parser generators rely on inline specifications. At the same time, inline specification obscures the structure of a syntax tree, since the grammar contains considerably more information. As a result, it also complicates the work of tool developers, who may not be the same as grammar writers and need to understand the syntax tree to get their work done. Elkhound addresses this concern by featuring a separate syntax tree specification, but also requires explicit semantic actions in the grammar to tie the tree back to the syntax. SableCC features an intermediate design point, with the abstract syntax being specified separately but still within the same file. This approach results in a more accessible specification, but also leads to duplication of code, since concrete and abstract syntax typically have considerable overlap.

The second axis captures whether the syntax tree is concrete or abstract. Abstract syntax trees omit information such as comments and layout, which are not necessary for most language processing tasks. Due to this compactness,

Table 1. The three main axes of the design space for syntax trees.

Axis	Choice	Main Trade-Off
Specification	Inline	More convenient for grammar writer
	Separate	More accessible for tool developer
Syntax Tree	Concrete	Necessary for refactorings, but higher overheads
	Abstract	More compact, easier to understand and process
Typing	Dynamic	More flexible, easier to reuse and extend
	Static	Safer, easier to ensure tool is correct

they are easier to understand and process, while also having lower time and space overheads. In contrast, concrete syntax trees represent the complete input and thus have higher overheads. But they are also required for source refactoring tools, which need to preserve formatting as much as possible. Elkhound supports both concrete and abstract syntax trees based on the same grammar. In contrast, JJTree and SableCC default to a concrete syntax tree, which can then be refined to an abstract one with additional annotations. Alternatively, SDF2 embeds the abstract syntax tree within a concrete one, thus providing both at the same time; though as demonstrated in [9], this approach also leads to sub-optimal parser performance for tools that do not require the concrete syntax.

The third axis captures whether the syntax tree is dynamically typed—i.e., relies on a single node data structure, which includes (say) a name to distinguish between kinds of nodes—or statically typed—i.e., arranges nodes into a hierarchy of data structures reflecting a language's organization of constructs. Dynamically typed trees tend to be more flexible and more easily extensible, because any node can be a child of any other node. For example, SDF2's dynamically typed trees enable the embedding of abstract syntax trees in concrete ones; they also enable source code templates by embedding nodes representing pattern variables in an existing tree [8]. *Rats!* provides similar facilities, again by leveraging dynamically typed trees [11]. In contrast, statically typed trees are generally less flexible but statically prevent program errors, especially when leveraging the visitor design pattern for implementing processing phases [12]. At the same time, J& demonstrates that nested inheritance can provide many of the benefits of dynamically typed trees while still ensuring static type safety—albeit with some notational overhead [13].

Table 2 summarizes the different parser generators and their support for syntax trees. In addition to the three axes, it also lists the primary language for generated parsers, with ANTLR supporting several other languages besides Java and Elkhound also supporting O'Caml. Next, it lists whether a parser generator supports semantic actions, which are not only useful for interfacing with hand-written syntax tree representations but also for recognizing languages that are not completely captured by a tool's formalism. Only SableCC and SDF2 disallow semantic actions, thus locking users into their respective grammar formalisms and syntax tree representations. Next, the table lists the parser generator's formalism and support for modules. Both SDF2 and *Rats!* rely on a formalism that

Table 2. Overview of parser generators and their support for syntax trees.

System	Spec.	Syntax Tree	Typing
ANTLR	Inline	Abstract	Dynamic
Elkhound	Separate	Abstract, opt. concrete	Static
JavaCC/JJTree	Inline	Concrete by default	Dynamic by default
JavaCC/JTB	Inline	Concrete	Static
SableCC	Inline	Concrete by default	Static
SDF2	Inline	Abstract inside concrete	Dynamic
Rats!	Inline	Abstract, opt. inside concrete	Dynamic, opt. static

System	Lang.	Actions	Formal.	Modules	Traversal
ANTLR	Java	Optional	LL(∗)	—	Tree grammars
Elkhound	C++	Required	GLR/LR	—	Visitors
JavaCC/JJTree	Java	Optional	LL(k)	—	Visitors
JavaCC/JTB	Java	Optional	LL(k)	—	Visitors
SableCC	Java	—	LALR(1)	—	Visitors
SDF2	C	—	GLR	Supported	Rewrite rules
Rats!	Java	Optional	PEG	Supported	Visitors

is closed under composition, integrate lexing with parsing, and feature full modularity for grammars, which also improves grammar re-use. Finally, the table lists a parser generator's support for syntax tree traversal. In general, parser generators supporting statically typed trees also support their traversal with visitors. In contrast, ANTLR supports the statically safe traversal of dynamically typed trees through tree grammars. Finally, SDF2 is integrated with Stratego/XT, which provides extensive facilities for traversing and transforming trees through rewrite rules [14].

2.1 *Rats!* Approach and xtc Integration

Based on the convenience of inline specifications, the compactness and performance of abstract syntax trees, and the flexibility of dynamically typed trees, *Rats!* defaults to inline, abstract, and dynamic syntax trees. From this baseline, *Rats!* can automatically (1) extract a separate syntax tree specification to serve as documentation for tool developers, (2) embed an abstract syntax tree in a concrete one to facilitate source refactorings, and (3) infer a strongly typed view to improve static safety. The dynamically typed syntax trees created by *Rats!*-generated parsers can be traversed with xtc's Java-based visitors, which, in departure from the visitor design pattern, use reflection to dynamically select the closest matching visit methods. They can also be traversed with visitors written in Typical, which is a dialect of ML designed to simplify semantic analysis, leverages the statically typed view of the syntax tree to ensure static safety, and compiles down to Java to seamlessly integrate with other xtc code [15]. By default, both Java-based and Typical-based visitors ignore any concrete syntax tree nodes, thus enabling the re-use of code independent of whether an abstract

Table 3. The operators supported by *Rats!*.

Operator	Type	Precedence	Description
`'`*c*`'`	Primary	6	Literal character
`"` ... `"`	Primary	6	Literal string
`[` ... `]`	Primary	6	Character class
`_`	Primary	6	Any character
`{` ... `}`	Primary	6	Semantic action
`null`	Primary	6	Null value
`@`*name*	Primary	6	Generic node marker
`(`*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 expression
[`<`*name*`>`] $e_1 \ldots e_n$	n-ary	2	Sequence
e_1 `/` ... `/` e_n	n-ary	1	Ordered choice

syntax tree is embedded in a concrete one or not. The rest of this paper explores how *Rats!* automatically realizes the different options from the same grammar and how `xtc`'s visitors access the syntax trees.

3 Grammar and Syntax Tree 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 3 summarizes the expression operators supported by *Rats!*. They mostly mirror the operators of parsing expression grammars [16], 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, including literals, options, repetitions, sequences, and choices. They differ in that options, repetitions, and choices are greedy and in the inclusion of syntactic predicates, which match expressions without consuming them. As discussed in detail in [16], 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.

While productions can explicitly set semantic values by assigning `yyValue` inside semantic actions, *Rats!* encourages the declarative specification of syntax trees through void, text-only, list-valued, and generic productions:

1. Void productions are declared as `void`, have no semantic value, and are useful for recognizing layout. For example,

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

   recognizes ignored space characters.
2. Text-only productions are declared as `String`, may only reference other text-only productions, and have the text matched in the input as their semantic values. They are useful for recognizing identifiers, keywords, and literals. For example,

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

   recognizes Java string literals; its semantic value is the entire literal including the opening and closing double quotes. The `![\"\\] _` expression uses a not-followed-by syntactic predicate, which inspects the input without consuming it, to recognize any character (`_`) but a double quote or backslash (`![\"\\]`).
3. List-valued productions are declared as `Pair<T>` and have (functional) lists of non-void component expressions as their semantic values. For example,

   ```
   Pair<Node> ExpressionList =
     Expression (void:",":Symbol Expression)* ;
   ```

   recognizes a comma-separated list of expressions and has a list of expression nodes as its semantic value. The string match `",":Symbol` is equivalent to

   fresh-id`:Symbol &{ ",".equals(`*fresh-id*`) }`

   but supported directly, because it represents a common idiom for matching specific keywords or symbols in PEGs. The value of the string match, in turn, is voided, so that *Rats!* can automatically deduce the repeated value to be an expression node.
4. Generic productions are declared as `generic` and have dynamically typed nodes as their semantic values. Each such generic node is an instance of class `GNode`. Its name is the production's name, unless specified through a generic node marker `@`*Name*. Its children are the values of a sequence's non-void component expressions. For example,

   ```
   generic ThrowStatement =
     void:"throw":Keyword Expression void:";":Symbol ;
   ```

   has a generic node as its semantic value. The node's name is "ThrowStatement" and its only child is an expression node.

To further eliminate the need for explicit semantic actions, *Rats!* automatically voids an option, repetition, or choice, if all of its component expressions are void, i.e., either nonterminals referencing void productions or explicitly voided expressions. Furthermore, if a sequence's expressions contain only one non-void expression or an explicit binding to `yyValue`, *Rats!* treats that expression's value as the overall sequence's value.

3.1 Separate Syntax Tree Specification

Given this combined grammar and syntax tree specification, the automatic extraction of a language's abstract syntax is straight-forward. First, *Rats!* resolves all modules into a single grammar, voids all voidable expressions, and deduces all sequences' semantic values. Second, it pares down the grammar by removing all void expressions. It also removes any extraneous information, including productions' attributes, sequences' names, and bindings not to `yyValue`. Third, it performs dead production elimination, which removes any productions that are not referenced anymore. Finally, it pretty prints the resulting grammar in either plain text or hyperlinked HTML format. Since *Rats!*' automatic deduction of semantic values excludes void expressions and those expressions are explicitly removed in steps two and three, the pared down grammar represents the abstract syntax, thus yielding the desired result. The implementation itself re-uses already existing functionality, with only step two requiring an additional 280 lines of Java.

4 Dynamically Typed Trees and Visitors

The visitor design pattern enables type-safe tree traversal through double dispatch [12]. Visitors implement a common interface `V` that has a `visit(N)` method for every distinctly typed node `N`, and nodes implement an `accept(V)` method that invokes the visitor on the node. While effective, the visitor design pattern can also be limiting. Most importantly for our purposes, it cannot dispatch on properties of a node, including the names of dynamically typed syntax tree nodes. Additionally, it does not support visit methods that accept a supertype of several nodes, thus leading to code duplication when processing several types of nodes the same way. Finally, changes to the syntax tree's structure tend to ripple through the entire code base, since the visitor design pattern requires changing the common interface `V` and thus all code implementing that interface.

`xtc` addresses these concerns by providing a more expressive alternative based on dynamic dispatch [17]. Under this model, the appropriate visit method is dynamically selected and invoked through Java reflection; method resolutions are cached to reduce the overhead of future dispatches. Intuitively, for a node of type `N`, `xtc` finds the closest supertype `N'`, for which the visitor has a method `visit(N')`, and calls that method. In practice, `xtc`'s visitor dispatch also considers interfaces, starting with the interfaces implemented by the node's class. For generic nodes, it precedes type-based resolution with name-based resolution, seeking a method `visit`*Name*`(GNode)` for generic nodes with name *Name*.

Overall, dynamic visitor dispatch provides a simple solution that meets our needs. Notably, we have implemented separate semantic analysis phases for Java and C using `xtc`'s dynamically dispatched visitors. We have then combined them into one visitor that performs semantic analysis for the Jeannie language, which integrates Java and C to simplify Java native interface programming [11]. Additionally, the combination of name-based and type-based resolution for generic nodes enables a simple implementation of generic tree traversal. A visitor only

needs to include `visit`*Name*`(GNode)` methods for generic nodes of interest and a catch-all visit method

```
public void visit(Node n) {
  for (Object o : n) if (o instanceof Node) dispatch((Node)o);
}
```

to process an entire syntax tree. While expressive, dynamic dispatch does eschew type safety. Section 6 describes how Typical complements dynamic dispatch with statically safe traversal of the same trees. Other efforts have explored different trade-offs between expressivity, safety, and complexity [18–20].

5 Concrete Syntax Trees

Rats!' support for concrete syntax is designed to preserve the structure of an abstract syntax tree—so that existing analysis code including type checkers can be re-used—while also capturing all characters in the input—so that source refactoring tools can preserve a program's formatting. To this end, it departs from established practice and does not represent each production with its own concrete syntax tree node. Rather, it annotates the abstract syntax tree with all character sequences in the input that are not already captured by the tree. Annotations in general wrap nodes and are represented as instances of class `Annotation`, which implements a node that references another node. Concrete syntax annotations in particular are represented as instances of class `Formatting`, which provides lists of nodes coming before and after the wrapped node. To ensure a uniform tree structure, all character sequences are represented by instances of class `Token`, which simply references a string.

The automatic embedding of abstract syntax trees in concrete ones is implemented in two phases. Since *Rats!* is scannerless, i.e., integrates lexing and parsing, the first phase determines the boundary between hierarchical and lexical syntax and then rewrites lexical productions to return tokens. To this end, *Rats!* first performs a bottom up pass over the grammar to identify all lexical productions; a production is lexical if the transitive closure of all reachable productions is either void or text-only. Next, *Rats!* performs a top down pass starting with a grammar's top-level production and marks all lexical productions referenced from non-lexical productions, i.e., hierarchical syntax, as token-level. Finally, *Rats!* rewrites all token-level productions to return tokens with the text matched in the input as their semantic values. At this point, all lexical productions referenced from hierarchical syntax have semantic values.

The second phase rewrites a grammar's hierarchical syntax to preserve all formatting. To this end, *Rats!* first changes all remaining void productions into generic productions. At this point, all productions referenced from the hierarchical syntax have semantic values, which can be captured in the syntax tree. Next, *Rats!* changes all productions that are list-valued, generic, or simply pass a node value through, so that the values of expressions that are void in the original grammar are referenced by the before and after lists of formatting nodes. The

formatting nodes, in turn, wrap the values of expressions that are not void in the original grammar. For instance, the production for Java throw statements in Section 3 contains two voided expressions whose values need to be captured in a formatting node. Using an explicit semantic action for expositional purposes, it is rewritten to

```
Node ThrowStatement =
  v$1:"throw":Keyword v$2:Expression v$3:";":Symbol
  { yyValue = GNode.create("ThrowStatement",
                           Formatting.round1(v$1,v$2,v$3)); } ;
```

where `round1()` creates a new formatting node wrapping `v$2`, with `v$1` the only value on the before list and `v$3` the only value on the after list.

One complication arises when a sequence does not contain nodes that can be wrapped by formatting nodes, for example, when a sequence contains only voided and list-valued expressions. Where possible, *Rats!* addresses this issue by lifting part of the sequence into an equivalent production, whose value can be annotated. For example, this production from the Java grammar

```
generic ClassBody =
  void:"{":Symbol Declaration* void:"}":Symbol ;
```

cannot be annotated and is split into

```
Node ClassBody =
  void:"{":Symbol ClassBody$Split1 void:"}":Symbol ;
generic ClassBody$Split1 = Declaration* @ClassBody ;
```

which can be properly annotated. If a sequence cannot be split, for example, when a void expression appears between two list-valued expressions, *Rats!* inserts a formatting node annotating a `null` value. This ensures that all formatting is preserved, but may change the structure of the abstract syntax tree in some corner cases. No changes to the tree structure occur for *Rats!*' C and Java grammars (in both the 1.4 and 5 versions).

To enable the re-use of analysis code for concrete syntax trees, the parent class of all visitors `Visitor` provides a default visit method for processing annotations

```
public Object visit(Annotation a) {
  return dispatch(a.node);
}
```

which ignores the annotation by simply returning the result of processing the wrapped node. Additionally, the superclass of all nodes `Node` defines not only a default getter

```
public Object get(int index) { ... }
```

for accessing the child at the specified index, but also `getGeneric()`, which strips any annotations before returning the child as a generic node, and `getString()`, which converts tokens into strings while also stripping any annotations before returning the child as a string. By using these facilities, tool code can process abstract and concrete syntax trees alike.

Table 4. Parser performance comparison.

Tree	Throughput	Decrease	Heap Util.	Increase
None	1919.0	—	47.1	—
Abstract	1421.0	1.35	58.4	1.24
Concrete	1019.0	1.39	74.8	1.28

5.1 Time and Space Overheads

To quantify the time and space overheads of concrete syntax trees, we compare the performance of *Rats!*-generated Java 1.4 parsers creating no syntax tree, an abstract syntax tree by itself, and an abstract syntax tree embedded in a concrete one. All three parsers are automatically generated from the same grammar; for the first parser, *Rats!* voids all productions besides those for keywords and symbols, whose values are used in string matches. As in [9], the experiments use a sample of 38 Java source files representing different programming and commenting styles and determine the overall throughput in KB/s and heap utilization in heap bytes per input byte through a least-squares-fit. All experiments are performed on an iMac from early 2006 with a 2 GHz Intel Core Duo processor and 2 GB of RAM running Mac OS X 10.4.10 and Apple's port of Java 5. Table 4 shows the results, listing both absolute numbers as well as factor decrease for throughput and factor increase for heap utilization. Both time and space overheads grow at approximately the same rate when switching from no syntax tree to an abstract syntax tree and when switching from an abstract syntax tree to a concrete syntax tree. In other words, the results demonstrate that *Rats!*' approach of relying on select annotations instead of turning all productions into concrete syntax tree nodes is effective at minimizing the overheads of concrete syntax

6 Statically Typed Syntax Trees

Rats!' automatic inference of statically typed syntax trees is designed to type as many trees as possible while still ensuring their static safety. To this end, it models syntax trees through ML's variant types. Variants are type-safe unions of one or more constructors and nicely capture different semantic categories such as declarations, statements, and expressions. Constructors, in turn, have a name and some number of arguments, thus providing a good fit for generic nodes having a name and some number of children. For example, the variant representing Java statements

```
mltype statement = ... | ThrowStatement of expression | ... ;
```

includes a constructor that directly mirrors the throw statement nodes created by the generic production in Section 3, and the variant representing Java class bodies

```
mltype class_body = ClassBody of declaration list ;
```

has a single constructor that directly mirrors the class body nodes created by the generic production in Section 5.

However, variants are not sufficient to type productions such as this one recognizing the initial clause of C for statements

```
Node InitialForClause = Declaration / Expression ;
```

or this one recognizing C sizeof expressions

```
generic SizeofExpression =
    void:"sizeof":Keyword UnaryExpression
  / void:"sizeof":Keyword
    void:"(":Symbol TypeName void:")":Symbol ;
```

which, by design, group different semantic constructs but do *not* introduce additional generic nodes. In the first example, the production simply passes the value of a declaration or expression through. In the second example, the production creates a generic node, whose only child is either an expression or a type name.

To statically type such productions, *Rats!* uses polymorphic variants [21, 22], which differ from regular, monomorphic variants in that the same constructor may appear in more than one (polymorphic) variant and which are already supported by O'Caml. At the same time, *Rats!*' use of polymorphic variants is restricted to each polymorphic constructor wrapping exactly one regular constructor value. For example, the variant representing C's initial clauses

```
mltype initial_for_clause = [
  | `SomeExternalDeclaration of external_declaration
  | `SomeExpression of expression ] ;
```

is polymorphic itself, thus mirroring the choice between declarations and expression in the above production for initial for clauses. Furthermore, the variant representing C's expressions

```
mltype expression = ...
  | SizeofExpression of [
      | `SomeExpression of expression
      | `SomeTypeName of type_name ] ;
```

contains an embedded polymorphic variant as the only argument to the `SizeofExpression` constructor, thus mirroring the sizeof expression nodes created by the above generic production. Polymorphic variants are distinct from monomorphic variants and thus recognizable as types automatically introduced for static safety but without corresponding values in the dynamically typed tree. They also enable a consistent naming convention, since polymorphic constructors, such as `` `SomeExpression `` in the examples, may appear in more than one polymorphic variant. Finally, since polymorphic constructors are restricted to wrapping regular constructor values, pattern matches on polymorphic variants can be implemented as set inclusion tests on the wrapped variants' constructor

Table 5. Overview of type constructs.

Construct	Models	Named After	Written As
Monomorphic			
– Variant	Semantic category	Production	`expression`
– Constructor	Generic node	Generic node	`SizeofExpression`
Polymorphic			
– Variant	Union of categories	Production	`initial_for_clause`
– Constructor	Single category	Monomorphic variant	`` `SomeExpression ``

names. Table 5 summarizes the constructs used for typing syntax trees, listing what they model, how they are named, and how they are written.

The type inference itself proceeds in two phases. The first phase creates variant types, i.e., semantic categories, for productions explicitly marked with the `variant` attribute and then propagates these types across a grammar's node-valued productions, while also assigning constructors, i.e., generic nodes, to variants. The second phase deduces consistent types for constructors' arguments, i.e., the children of generic nodes, by analyzing productions' component expressions, thus completing the strongly typed tree. The types are then printed as definitions for the Typical language, which extends ML with declarative constructs that simplify semantic analysis and which compiles down to Java to integrate with other xtc code [15]. However, the same analysis could produce valid O'Caml types with minor modifications—thus laying the foundation for *Rats!*-generated O'Caml parsers. The same analysis could also produce a statically typed Java class hierarchy, encoding each monomorphic variant as its own class, with a subclass for each constructor, and encoding each polymorphic variant as an interface that is implemented by the classes of wrapped monomorphic variants.

The goal of the first phase is to assign variants to productions and constructors; its algorithm is fully specified in the appendix. Basically, the algorithm creates variant types for productions explicitly marked with the `variant` attribute and then pushes and pulls these types through a grammar's productions until all node-valued productions have been annotated. This approach is motivated by the observation that many productions pass another production's semantic value through. For example, the following production from both the C and Java grammars

```
generic RelationalExpression =
    RelationalExpression RelationalOperator ShiftExpression
  / yyValue:ShiftExpression ;
```

either creates a new generic node representing a relational expression or passes the value of the production recognizing shift expressions through. In terms of the algorithm, `ShiftExpression` is a *contributor* to `RelationalExpression`, and both productions must have the same type.

So, to assign variants to productions, the algorithm either pushes the variant type from a production that already has a variant to contributors that have

not yet been annotated, or it pulls the variant type from contributors that already have a variant into a production that has not yet been annotated. While pushing, the algorithm also assigns constructors, i.e., the generic nodes created by a production, to variants. While pulling, it also creates polymorphic variants for productions, such as the above `InitialForClause` production, that pass the values of different variants through. Pushing and pulling alternate in rounds: productions newly annotated with variants while pulling become the candidates for the next round of pushing, and node-valued productions not yet annotated with variants after pushing become the candidates for the next round of pulling. Pushing and pulling stops when no productions have been annotated while pulling, i.e., when a fixed-point has been reached.

This push/pull fixed-point computation is actually performed twice. The first fixed-point computation groups productions and generic nodes into semantic categories based on explicit annotations in the grammar. It is initialized with all productions that are explicitly marked with the `variant` attribute. For each such production, the algorithm creates a new monomorphic variant, which is named after the production by converting from *Rats!*' CamelCase naming convention to ML's lower_case_with_underscores convention. The second fixed-point computation ensures that all node-valued productions have variant types. It is initialized with all generic productions that do not have variant types after the first fixed-point computation. For each such production, the algorithm again creates a new monomorphic variant, naming it after the production.

The second phase of type inference deduces consistent types for the arguments of monomorphic constructors, i.e., the children of generic nodes. Its algorithm is largely straight-forward: it first determines the types of component expressions for all alternatives in generic productions and then unifies the types corresponding to the same argument for the same constructor across different alternatives. For example, a string literal or string match has `string` as its type, a nonterminal has the corresponding production's type, and a repeated expression has `'a list` as its type, where `'a` is the expression's type. At the same time, the implementation relies not only on polymorphic variants to unify distinct monomorphic variants—as illustrated above for the production recognizing C sizeof expressions—but two more non-standard types, which also impact how Typical code accesses the underlying, dynamically typed syntax tree.

1. The implementation uses the built-in Typical type `'a opt = 'a` to mark constructor arguments resulting from optional expressions in the grammar. The corresponding values in the dynamically typed tree may be `null`, indicating that the expression did not match the input. For example, this production from the C and Java grammars

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

   results in

   ```
   ReturnStatement of expression opt
   ```

   as the type of generic nodes with name "ReturnStatement". Since Typical already injects a `bottom` value corresponding to Java's `null` value into all

types, the `'a opt` type largely serves as documentation. Though an O'Caml implementation would instead use O'Caml's

```
mltype 'a option = None | Some of 'a ;
```

to make the optional value explicit in the syntax tree.

2. The implementation uses the built-in Typical type `'a var = 'a` to mark constructor arguments that do not appear in all instances of a generic node with that name. It accounts for a generic production having different alternatives with different numbers of non-void component expressions. For example, this production from the C grammar

```
generic CaseLabel =
    void:"case":Keyword ConstantExpression
    void:"...":Symbol ConstantExpression void:":":Symbol
  / void:"case":Keyword ConstantExpression void:":":Symbol ;
```

results in

```
CaseLabel of expression * expression var
```

as the type of generic nodes with name "CaseLabel". Before accessing a `var` argument, Typical-generated code first checks whether the generic code has the requisite number of children, otherwise returning `null`. An O'Caml implementation would instead use the `'a option` type to declare such arguments and the `None` value to pad all constructor invocations.

Rats!' type inference correctly types the syntax trees of the C and Java grammars (both 1.4 and 5 versions). The C grammar required 8 productions to be annotated with the `variant` attribute. We also had to rewrite two productions to avoid a unification error caused by a string and node value appearing as the same argument to the same constructor. The Java 1.4 grammar required 7 productions to be annotated with the `variant` attribute, with one more annotation to correctly type the Java 5 grammar, which is implemented as an extension of the 1.4 version using *Rats!*' module system. For the C grammar, the annotated productions recognize declarations, declaration specifiers, declarators (both abstract and concrete), statements, labels, expressions, and designations in compound literals. For the Java grammar, the annotated productions recognize declarations, modifiers, types, type names, statements, the control clause of regular and enhanced for loops, switch clauses, and expressions. Overall, these results demonstrate that *Rats!*' type inference is effective, while only requiring minor modifications to a grammar.

7 Conclusion

Given a single grammar, this paper has presented automated techniques for (1) extracting an abstract syntax tree specification, (2) capturing the concrete syntax as annotations on the abstract syntax tree, and (3) inferring static types for abstract syntax tree nodes. Taken together, these techniques significantly increase the flexibility of parser generators, while also facilitating the re-use of

grammars. All techniques are implemented in the *Rats!* parser generator and integrated with the `xtc` toolkit for source-to-source transformers. They have also been applied to real-world grammars for C and Java. The open source distribution of *Rats!* and `xtc` is available at `http://cs.nyu.edu/rgrimm/xtc/`.

References

1. Parr, T.: The Definitive ANTLR Reference. The Pragmatic Programmers (May 2007)
2. McPeak, S., Necula, G.C.: Elkhound: A fast, practical GLR parser generator. In: Proc. 13th CC. Volume 2985 of LNCS., Springer (March 2004) 73–88
3. java.net: JJTree reference documentation. `https://javacc.dev.java.net/doc/JJTree.html`
4. Palsberg, J., Tao, K., Wang, W.: Java tree builder. `http://compilers.cs.ucla.edu/jtb/`
5. Gagnon, É.: SableCC, an object-oriented compiler framework. Master's thesis, McGill University (March 1998)
6. Agbakpem, K.K., Gagnon, É.: SableCC. `http://sablecc.org/`
7. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
8. Bravenboer, M., Visser, E.: Concrete syntax for objects. In: Proc. 2004 OOPSLA. (October 2004) 365–383
9. Grimm, R.: Better extensibility through modular syntax. In: Proc. 2006 PLDI. (June 2006) 38–51
10. Levine, J.R.: lex & yacc. O'Reilly (October 1992)
11. Hirzel, M., Grimm, R.: Jeannie: Granting Java native interface developers their wishes. In: Proc. 2007 OOPSLA. (October 2007)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (January 1995)
13. Nystrom, N., Qi, X., Myers, A.C.: J&: Nested intersection for scalable software composition. In: Proc. 2006 OOPSLA. (October 2006) 21–36
14. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fundamenta Informaticae **69**(1–2) (2005) 123–178
15. Grimm, R., Harris, L., Le, A.: Typical: Taking the tedium out of typing. Technical Report TR2007-904, New York University (November 2007)
16. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: Proc. 31st POPL. (January 2004) 111–122
17. Blosser, J.: Java tip 98: Reflect on the visitor design pattern. JavaWorld (July 2000) `http://www.javaworld.com/javaworld/javatips/jw-javatip98.html`.
18. Grothoff, C.: Walkabout revisited: The runabout. In: Proc. 17th ECCOP. Volume 2743 of LNCS. (July 2003) 101–125
19. Millstein, T.: Practical predicate dispatch. In: Proc. 2004 OOPSLA. (October 2004) 345–364
20. Visser, J.: Visitor combination and traversal control. In: Proc. 2001 OOPSLA. (October 2001) 270–282
21. Garrigue, J.: Programming with polymorphic variants. In: Proc. 1998 ACM SIGPLAN Workshop on ML. (September 1998)
22. Garrigue, J.: Simple type inference for structural polymorphism. In: Proc. 9th Workshop on Foundations of OO Languages. (January 2002)

Appendix: The Push/Pull Fixed-Point Algorithm

```
global pushlist, pulllist                          ▷ Declare global worklists.

procedure Assign-Variants(grammar)      ▷ Assign variants to productions.
    ▷ Process productions with variant attribute.
    pushlist ← nil
    for all p in productions[grammar] do
        if variant ∈ attributes[p] then
            if Has-Node-Value(p) then
                pushlist ← pushlist ‖ p
                variant[p] ← Create-Monomorphic-Variant(name[p])
            else
                error "Invalid variant attribute for production " ‖ name[p]
            end if
        end if
    end for
    Fixed-Point()

    ▷ Process generic productions still without variant type.
    for all p in productions[grammar] do
        if Is-Generic(p) ∧ variant[p] = nil then
            pushlist ← pushlist ‖ p
            variant[p] ← Create-Monomorphic-Variant(name[p])
        end if
    end for
    Fixed-Point()
end procedure

procedure Fixed-Point()                            ▷ Compute fixed-point.
    while pushlist ≠ nil do
        for all p in pushlist do Push(p) end for

        pushlist ← nil
        for all p in productions[grammar] do
            if Has-Node-Value(p) then
                pulllist ← nil
                Pull(p)
            end if
        end for
    end while
end procedure
```

▷ Continued on next page.

```
procedure PUSH(p)                                   ▷ Push variant through production.
    ▷ Assign variant to constructors.
    for all c in constructors[p] do
        if variant[c] = NIL then
            variant[c] ← variant[p]
        else if variant[c] ≠ variant[p] then
            error "Inconsistent variants for constructor " ‖ name[c]
        end if
    end for

    ▷ Assign variant to contributors.
    for all q in contributors[p] do
        if variant[q] = NIL then
            variant[q] ← variant[p]
            PUSH(q)
        else if variant[q] ≠ variant[p] then
            error "Inconsistent variants for production " ‖ name[q]
        end if
    end for
end procedure

function PULL(p)                                    ▷ Pull variant into production.
    ▷ Avoid infinite recursions.
    if variant[p] ≠ NIL then
        return variant[p]
    else if p ∈ pulllist then
        return NIL
    end if

    ▷ Determine contributors' variants.
    pulllist ← p ‖ pulllist
    variants ← { PULL(q) | q ∈ contributors[p] }
    pulllist ← TAIL(pulllist)

    ▷ Determine production's variant.
    if variants − {NIL} = {v} where IS-MONOMORPHIC(v) then
        variant[p] ← v
        pushlist ← pushlist ‖ p
    else if NIL ∉ variants ∧ constructors[p] = ∅ then
        variant[p] ← CREATE-POLYMORPHIC-VARIANT(variants)
    else if variants ≠ {NIL} then
        error "Inconsistent variants for production " ‖ name[p]
    end if
    return variant[p]
end function
```