

Tatoo: an innovative parser generator

Julien Cervelle
Université de Marne la Vallée
Institut Gaspard-Monge
UMR CNRS 8049
77454 Marne-la-Vallée France
julien.cervelle@univ-mlv.fr

Rémi Forax
Université de Marne la Vallée
Institut Gaspard-Monge
UMR CNRS 8049
77454 Marne-la-Vallée France
remi.forax@univ-mlv.fr

Gilles Roussel
Université de Marne la Vallée
Institut Gaspard-Monge
UMR CNRS 8049
77454 Marne-la-Vallée France
gilles.roussel@univ-mlv.fr

ABSTRACT

This paper presents Tatoo, a new parser generator. This tool, written in Java 1.5, produces lexer and bottom-up parsers. Its design has been driven by three main concerns: the ability to use the parser with the non-blocking IO API; the possibility to simply deal with several language versions; a clean separation between the lexer definition, the parser definition and the semantics. Moreover, Tatoo integrates several other interesting features such as lookahead-based rule selection, pluggable error recovery mechanism, multiple character sets optimized support.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors; D.2.2 [Software Engineering]: Design Tools and Techniques

Keywords

Parser generator

1. INTRODUCTION

This paper presents Tatoo, a new parser generator. More precisely, given a set of regular expressions describing tokens, a formal specification of a grammar and several semantic hints, Tatoo, like many other existing tools [13, 10], can generate a lexer, a parser and several implementation glues allowing to run a complete analyzer that creates trees or computes simpler values. Thanks to a clean separation between specifications, the lexer and the parser may be used independently. Moreover, there implementations are not strongly linked to a particular semantic evaluation and may be reused, in different contexts, without modification.

Tatoo is written in Java 1.5 and heavily uses parameterized types. Currently, generated implementations are also Java 1.5 compliant but thanks to the simple extension mechanism of Tatoo, other back-ends may be provided for different target languages or implementations. This extension mechanism is already used to produce different lexer

and parser implementations. For the lexing, depending on the character encoding, Tatoo provides three implementations: interval based, table based and *switch-case* based. For the parsing, depending on the grammar, SLR, LR(1) and LALR(1) implementations are available.

For the front-end, the Tatoo engine relies on reified lexer or parser specifications where regular expressions or productions are Java objects. Thus, everything may be implemented in Java. However, for usability, a simple XML front-end for these specifications is also provided.

For the Java back-end, a library is provided for runtime support. It contains generic classes for lexer and parser implementation, glue code between lexer and parser, a basic AST support and some helper classes that ease debugging. Generated Java code uses parameterized types and enumeration facilities to specialize this runtime support code. Moreover, all memory allocations are performed at creation time.

One main feature of the Java back-end resides in its ability to work in presence of non-blocking inputs such as network connections. Indeed, the Tatoo runtime supports push lexing and parsing. More precisely, contrarily to most existing tools, Tatoo lexers do not directly retrieve “characters” from the data input stream but are fed by an input observer which may work on non blocking inputs. The lexer retains its lexing state and resumes lexing when new data is provided by the observer. When a token is unambiguously recognized the lexer pushes it to the parser in order to perform the analysis. It is easy to provide a classical pull implementation based on this push implementation.

Another innovative feature of Tatoo is its support of language versions. This is provided by two distinct mechanisms. First, productions can be tagged with a version and the Tatoo engine produces a shared parser table tagged with these versions. Then, given a version provided at runtime, the parser selects dynamically the correct actions to perform. Second, the glue code linking the lexer and the parser supports dynamic activation of lexer rules according to the parser context. This feature allows, in particular, the activation of keywords according to the version.

Moreover, Tatoo integrates several other interesting features such as a pluggable error recovery mechanism and multiple character sets optimized support.

The rest of this paper is organized as follows. An overview of Tatoo development process is given in section 2. Then, in section 3, innovative features of Tatoo are detailed. Related works are presented in section 4, before conclusion.

2. TATOO OVERVIEW

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPPJ 2006, August 30 – September 1, 2006, Mannheim, Germany
Copyright 2006 ACM 0-12345-67-8/90/01 ...\$5.00.

2.1 General presentation

In this section a general overview of the typical construction of an analyzer using Tatoo is detailed. This process is divided into three steps:

- specification of the language to be parsed and of several semantic hints;
- automated implementation of parsing mechanism according to the chosen method and target language;
- implementation of the semantics.

2.2 Language specification

In order to simplify the bootstrapping of our system, we have chosen to describe the language using XML files, and a SAX-based XML parser. The specification of a language is provided by three files: one for the lexer (`.xlex`), one for the parser (`.xpars`) and one for the glue between them and a semantic part (`.xtls`).

For each file item (lexer rule, terminal, non-terminal, production, version, ...), a corresponding *id* is defined. At runtime, those items are referenced using this *id*.

In order to have the grammar unpolluted by code or semantic information, the grammar and the lexer specifications are clearly separated from the semantics, so that the same language can be used with different semantics.

2.2.1 Lexer

The lexing process consists in cutting the input text in lexical units called tokens. The specification file defines rules which are matched against the input. The lexer forwards the recognized tokens to a *listener* which, may or may not, send them to the parser.

2.2.1.1 Rules.

Each rule is described by one *main* regular expression and, optionally, a *following* regular expression. A rule is matched if the input matches the former expression and is followed by a sequence which matches the latter regular expression, if provided.

Regular expression can be expressed either using a Perl-like syntax [4], or using an XML syntax, the latter being mainly used for bootstrapping. For instance, an identifier is defined in XML by:

```
<rule-xml id="identifier">
  <main>
    <!-- definition of main regex -->
    <cat>
      <set>
        <interval from="a" to="z"/>
        <interval from="A" to="Z"/>
        <letter value="_"/>
      </set>
      <star>
        <set>
          <interval from="a" to="z"/>
          <interval from="A" to="Z"/>
          <interval from="0" to="9"/>
          <letter value="_"/>
        </set>
      </star>
    </cat>
```

```
<!-- no follow expression -->
</rule-xml>
```

and in Perl-like by:

```
<rule id="identifier"
      pattern=' [a-zA-Z_] [a-zA-Z0-9_]*' />
```

At runtime, when a particular rule is recognized, its *id* is forwarded to the listener using a back-end dependent type. The default Java back-end uses Java 1.5 enumeration types. The character sequence corresponding to the token is also provided, but it is not extracted from the input. It is the responsibility of the programmer to retrieve it, if needed.

As in other lexer generators, support for macros is also provided using `define-macro-xml` and `define-macro XML` tags.

2.2.1.2 Activator.

In order to speed up lexing process, a mechanism is provided to only select a subset of “useful” rules. The lexer is constructed using an object implementing the interface `RuleActivator` that selects, depending of the context, which rules are active.

One possible use of this mechanism is the implementation of Lex [12] start conditions, but as explained in section 3.2.1 it is also used to support language evolutions.

2.2.2 Parser

The parsing process consists in verifying that the flow of tokens produced by the lexer respects a particular organization described by a formal grammar.

The parser file specifies this grammar. It is described by the declaration of its productions, allowing an extended syntax of BNF using one-depth regular expressions (lists and optional elements). One or more starts (or axioms) for the grammar must be declared. If more than one axiom is declared for the grammar, the “real” axiom is precised to the parser at runtime.

For instance, Java method headers can be declared using the following production:

```
<production id="header">
  <lhs id="header"/>
  <rhs>
    <list id="modifiers" empty="true"
          element="methodModifier"/>
    <right id="returnType"/>
    <right id="identifier"/>
    <right id="leftPar"/>
    <list id="parameters" empty="true"
          element="variableDeclaration"
          separator="comma"/>
    <right id="rightPar"/>
    <right id="throwsDeclaration"
          optional="true"/>
  </rhs>
</production>
```

`right` means a single or possibly no occurrence of a terminal or non-terminal ; `list` means a list of terminals and non-terminals.

To resolve common conflicts, as in Yacc [9], priorities can be associated with productions and terminals. For simplicity, priorities of productions are automatically deduced from priorities of their terminals, if applicable.

In the grammar file an error terminal is also declared for error recovery (see section 3.3.2) and versions may be specified for productions (see section 3.2.2). Moreover, the parser can provide, back to the lexer, the set of expected terminals at each step of parsing (see section 3.2.1).

2.2.3 Tools

A complete language analyzer uses lexer and parser processes to perform semantic computations according to a particular input stream. The tools file describes how the lexer and the parser work together and their interfaces with the semantics.

First, the association between rules and terminals is described. This piece of information is declared for two purposes. On the one hand, Tatoo generates automatically a lexer listener which forwards the corresponding terminal to the parser. On the other hand, it generates a lexer activator that selects only useful rules according to the expected set of terminals provided by the parser (see section 3.2.1). Rules that have to be always selected (like blank skipping rule) are those not associated with any terminal.

For instance the following lines associate rule `identifier` with terminal `identifier` and says that `space` are not pushed to the parser.

```
<rule id="identifier" terminal="identifier"/>
<rule id="space" spawn="false"/>
```

Second, like in Yacc, the tools file declares the type (primitive or not) of the attribute associated with each terminal and each non-terminal.

For instance, following lines associate the type `String` with the terminal `identifier` and the type `int` with non-terminals `ref` and `expr`.

```
<terminal id="identifier" type="String"/>
<non-terminal id="ref" type="int"/>
<non-terminal id="expr" type="int"/>
```

This piece of information is used to generate automatically a lexer listener. The listener receives, at creation time, an object provided by the developer that computes the terminal attribute value from the character sequence matched by the rule. This object specify a particular semantics for terminal attributes implementing the interface `TerminalAttributeEvaluator` that is generated by Tatoo. Hence, the semantics can be modified without regenerating the lexer classes.

For our running example, tools generator produces the interface below:

```
interface TerminalAttributeEvaluator {
    void space(CharSequence seq);
    String identifier(CharSequence seq);
}
```

Types of non-terminals and terminals are also used to build a parser listener which performs semantic actions. Like for the lexer listener, the developer has to provide, at creation time, an implementation of an interface called `GrammarEvaluator` generated by Tatoo. For all productions this implementation constructs the attribute of the left hand side non-terminal from attributes of the right hand side elements. An implementation for the attribute stack and for shift handling are provided to simplify this implementation.

Here again, since the semantic computations conform to a predefined interface, changing the behavior of the parser

does not require to reprocess lexer and parser files. Moreover, different semantics may be attached to the very same generated lexer and parser.

For example, using previous tools specification and a grammar like the following:

```
<start id="expr"/>
<production id="p1">
    <lhs id="expr"/>
    <rhs>
        <right id="expr"/>
        <right id="plus"/>
        <right id="expr"/>
    </rhs>
</production>
<production id="p2">
    <lhs id="expr"/>
    <rhs><right id="ref"/></rhs>
</production>
<production id="p3">
    <lhs id="ref"/>
    <rhs>
        <right id="excl_mark"/>
        <right id="identifier"/>
    </rhs>
</production>
```

the interface generated by tools generator is:

```
interface GrammarEvaluator {
    int p1(int expr, int expr2);
    int p2(int ref);
    int p3(String identifier);
    void acceptExpr(int expr);
}
```

In this generated interface, there is one method for each production. Each one is strongly typed according to types associated with terminals and non-terminals in the tools file. Moreover, one `accept()` method is generated for each axiom non-terminal. Here, the only axiom specified for the grammar is `expr`.

Finally, tools generator offers the possibility, if needed, to produce a special semantic evaluator that performs the construction of an abstract syntax tree. In this tree, each node kind has a corresponding Java type. It allows efficient traversal of the tree using the visitor design pattern (see section 3.3.3).

2.3 Generation

Tatoo provides pluggable generators in order to build code for any target language. The Tatoo engine builds all the tables and maps needed to write the lexer, parser and tools code

- the transition tables of each finite automaton;
- the action table for the parser (according to state and lookahead);
- maps between various objects (versions and productions, rules and terminals...)

In order to generate class and interface files, these objects are made available in the generator, for back-end extensions, using a bus mechanism. The default Java back-end uses

Velocity [2], an Apache package that follows a model-view approach to generate files. More precisely, the Velocity language allows to specify template files used by the Velocity engine that directly interacts with Java objects provided by the Tatoo generator to produce output files.

2.4 Runtime

2.4.1 Memory allocation

One main concern about the runtime was to prevent the parser and the lexer from creating any new object. Thus, all memory required by the runtime is allocated when the lexer and the parser are created. No more memory is allocated during execution, except potential parser stack extensions. This feature is important in order to allow to embed Tatoo parsers in long-living applications such as web servers.

2.4.2 Lexer

The lexer is built on top of a buffer used to store available characters from the input stream to be processed. When new characters are made available in this buffer by an external mechanism (such as a file reader or a network observer), the `step()` method of the lexer may be called. Then, all available characters in the buffer will be examined. For each character, once all rules have been applied, the lexer verify if a lexical unit has been completely recognized. When multiple rules match, the lexer uses the following priority: first the longest match, and then the first one declared (this behavior conforms to Lex [12]). Each time a “winning” rule is unambiguously found, the lexer listener’s only method, `ruleVerified()`, is called with this rule and an object implementing the interface `TokenBuffer` as arguments. The interface `TokenBuffer` provided a method `view()` to access (as a `CharSequence`) the recognized lexical unit and a method `discard()` to indicate that recognized characters may be pulled off the lexer input buffer (they are not discarded automatically).

2.4.3 Parser

Within the lexer listener generated by the tools generator, terminals are passed, when required, to the parser using its method `step()`. This method performs all the actions it can using this new input terminal (possibly some reduces and one shift). Like for the lexer, a listener is attached to the parser at creation time, and for each action, the listener is called in order to perform the semantic part. The parser listener has three methods, one for each kind of action: `shift()` which means a terminal is read; `reduce()` which means a production is applied and its left hand side replaces its right hand side; and `accept()` when the input is accepted for a specific axiom. To trigger acceptance, one has to call the `close()` method. It happens when the lexer reaches an “end-of-file” condition.

All these gears are described in figure 1.

Together with these mechanisms, error recovery (see section 3.3.2), selection of valid rules according to expected terminals and support for language versions (see section 3.2.2) are also available.

3. INNOVATIVE CONCERNS

3.1 Non blocking parsing

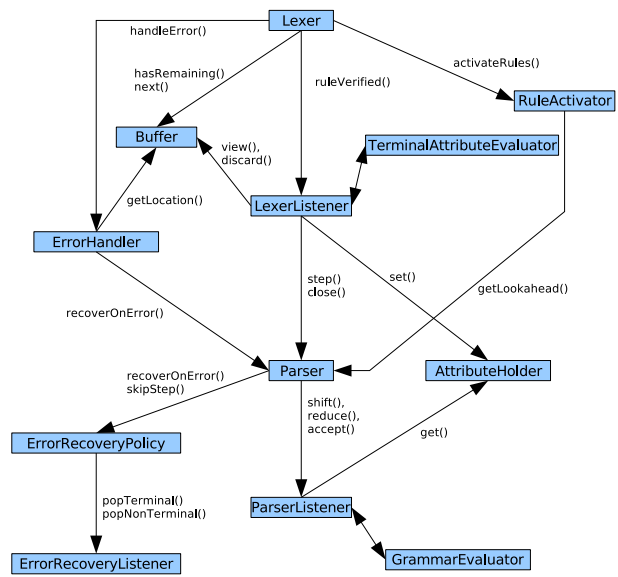


Figure 1: Lexer, parser and tools interactions

The primary goal of the development of Tatoo was to provide a parser generator compatible with non-blocking IO. Indeed, popular existing parser generators only produce lexers and parsers working with blocking API. These extract data from the stream and wait until data is available. This behavior is known as pull parser (see figure 2) and calls to the parser are blocking. This behavior is acceptable for parsers since they usually work on files where blocking periods are small, but this is not compatible with network streams where waiting periods may be long. In order to support an efficient parsing of network input streams Tatoo supports push lexers and parsers.

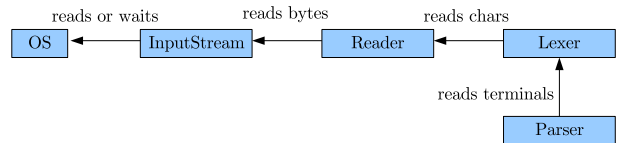


Figure 2: Pull parsers and lexers

3.1.1 Push behavior

In the push behavior (see figure 3), the lexer and the parser processes maintain a state. These processes are started explicitly when data is available and stopped when no more data remains in the input buffer which does not mean that the end of the stream is reached. Indeed, data may arrive later and thus the end of the stream has to be notified explicitly. Thus, the basic API of our push lexers and parsers contains two methods: `step()` and `close()`. The method `step()` is used to push characters or terminals and the method `close()` indicates the end of the stream. A method `run()` is provided for convenience to simulate pull behavior. It just performs a read/step loop until the end of the file is reached and then it closes the lexer that closes the parser.

3.1.2 Common abstract buffer API

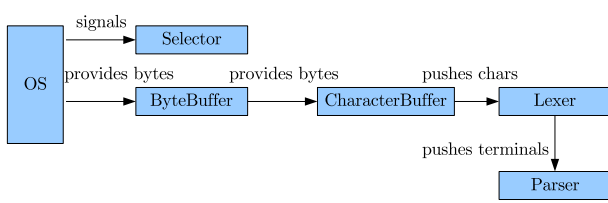


Figure 3: Push parsers and lexers

Lexers generated by Tatoo can run on several kinds of buffer. Indeed, the lexer only requires four methods encapsulated in the interface `CharacterBuffer` to be implemented. These four methods are: `next()`, `hasRemaining()`, `unwind()` and `previousWasNewLine()`. The first method returns the next character in the buffer; the second tells whether characters remain in the buffer; the third is used by the lexer to indicate that a token is recognized and that the lexing has to resume; the last one is used to select rules that require to occur at the beginning of a line.

Tatoo runtime provides several wrappers implementing this interface for Java NIO buffers, readers and input streams.

3.2 Language evolution

In order to follow language evolutions such as those of Java, Tatoo introduces two special mechanisms. The first one is called lookahead activator and it simplifies keyword additions. The second, called grammar versioning, permits to specify, in a single grammar, different language constructions tagged with their version.

3.2.1 Lookahead activator

Like several other lexer generators such as Lex [12], the lexer specification is composed of multiple rules and of an activator to activate or deactivate these rules during analysis. The activator is usually implemented by the developer to implement start conditions in the lexer. In Tatoo, by default, the tools generator implements a special activator based on the terminals expected by the parser, known as lookahead. More precisely, for each state of the push-down automaton, the parser only provides the terminals that do not cause syntax errors. These terminals are associated with particular lexer rules that the generated activator retains, deactivating other rules.

This feature allows first to speed up the lexing process since usually only few terminals/rules are selected in each parser state. Second, and more importantly, it allows to minimize conflicts in existing source codes when a new keyword is introduced in the language. Indeed, if the new keyword only appears in a particular production of the grammar, the lexing rules matching this keyword is not selected in other productions, and the keyword can be recognized as an identifier. For instance, the `enum` new keyword of Java 1.5 potentially conflicts with identifiers of previous versions. However, since `enum` keyword is never acceptable where an `enum` identifier is acceptable, a Java parser that uses the lookahead activator could suppress such conflicts.

This approach suppresses parser process errors. Indeed, the lexer never generates an unexpected terminal. Thus, it disables the classical grammar-based error recovery mechanisms. This is why the Tatoo tools generator translates lexing exceptions into parser errors, pushing a special error terminal to the parser.

3.2.2 Grammar version

The parser file allows to associate a particular version with each production and specify a single inheritance relation between these versions. The version hierarchy tree is specified by declaring for each version an optional implied version. The implication relation between versions is transitive. Each production can only declare one version, but if this version is implied by other versions, it is used by all these versions.

For instance, the following portion of parser file specifies two different versions `v1` and `v2` linked by an implication relation and two productions `p1` and `p2` with their respective versions. Since, `v1` is implied by `v2`, production `p1` is active for version `v2`.

```

<version id="v1"/>
<version id="v2" implies="v1"/>

<production id="p1" version="v1">
  <lhs id="A"/><rhs><right id="a"/></rhs>
</production>

<production id="p2" version="v2">
  <lhs id="A"/><rhs><right id="b"/></rhs>
</production>
  
```

Given a versioned parser specification, first, the parser generator computes the push-down automaton according to the grammar, using a bottom-up algorithm (SLR, LR(1) or LALR(1)) taking into account neither versions nor priorities. Afterwards, for each version, the Tatoo engine selects transitions compatible with this version. More precisely, a reduce transition is compatible with a version if the target production is active for the selected version. A shift transition is compatible with a version if one of the kernel items (productions) of its target state is active. Then, for each version, the classical conflict resolver based on priority and associativity determines the unique action to perform. Lastly, in order to reduce action lookup and the size of the table, if all versions lead to a same action, version information are not saved in the table. Otherwise an action is saved for each version using a composite action. The “real” version, provided at runtime, allows to select the correct action to perform. Figure 4 illustrates all these steps.

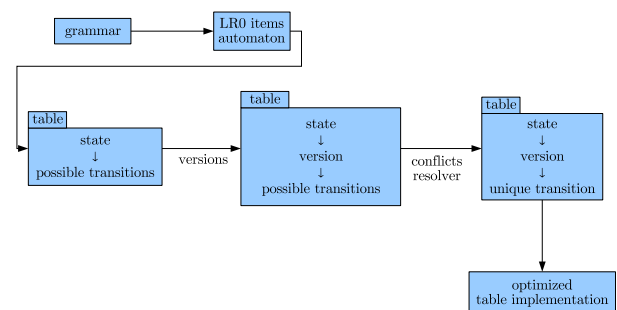


Figure 4: Parser table construction with versions

3.3 Other features

Even if non-blocking API and grammar evolution supports were primary goals for the development of Tatoo, generated analyzers also support several other interesting features.

3.3.1 Charset selection

In order to support multiple charsets, Tatoo can rely on the Java `java.util.Reader` mechanism. However, it is also possible to encode, at generation, the lexer tables in the target charset of the input stream. This mechanism avoids unnecessary decoding of characters at runtime. The drawbacks of this approach are that lexer implementations is linked with a particular charset and that interval specification in regular expressions, such as `[a-d]`, depends of the character encoding. However, these intervals should only concern letters or digits whose intervals do not depend on the charset.

Another optimization that depends on the charset is the lexer implementation of rule automata. The classical implementation of transitions in these automata uses character intervals since for unicode charset they cannot be implemented directly using a character array. Two other implementations are provided. The first one uses arrays if the charset is small enough. The second one uses a *switch-case* implementation and is valuable if most transitions can be implemented by the default case, which is often the case for programming languages tokens.

3.3.2 Pluggable error recovery

As explained in section 3.2, only lexer errors may occur during analysis, but these errors are forwarded to the parser in order to trigger an error recovery mechanism. This mechanism is plugged into the parser, at creation time, by the developer. The default algorithm provided to the parser does not perform any error recovery.

However, it is quite simple (even if less user-friendly than for LL(k) parser) to develop an error recovery mechanism, specific to a particular context, implementing the abstract class `ErrorRecoveryPolicy`. Two main methods have to be implemented: `recoverOnError()` and `skipStep()`. When an error occurs, the method `recoverOnError()` is called with the parser state in order to recover from the error. Then, either the error is recoverable and parsing may continue, or parsing is aborted. Moreover, the method `skipStep()` is called each time a terminal is pushed to the parser. It can be used by the error recovery mechanism to indicate that this terminal must be ignored; for instance to skip all terminals until a semicolon is found.

We provide the implementation of two classical error recovery algorithms. The first one is described in [1] and used by Yacc [9] and the second one is similar to the ANTLR [13] error recovery algorithm.

Because these two algorithms can pop a state out of the stack, the implementation uses a specific listener, `ErrorRecoveryListener`, in order to signal to the semantic part that the error recovery mechanism has popped a state.

3.3.3 Generic AST construction

Abstract tree construction is an optional feature of the tools generator that does not require extra specification. The AST generator uses the tools file and it produces a specific semantic evaluator linked with the parser.

Like in SableCC [5], types of nodes are directly deduced from the grammar. Furthermore, these types accept visitors [6] for semantic evaluations.

However, Tatoo generated types are particular since they allow two views of the same node. For the first one, Tatoo creates a specific set of classes and interfaces directly deduced from the grammar specification. This view is compa-

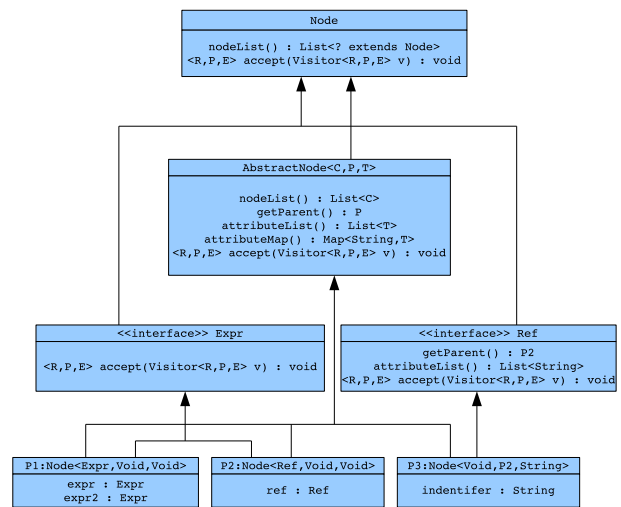


Figure 5: Example of AST type hierarchy

table with the tree implementation produced by the JAXB [14] approach for XML document. There is one interface for each non-terminal and a concrete class for each production. The right hand side of the production is implemented using attributes. Their types are as precise as possible.

The second view allows a generic access to the tree comparable with the XML DOM [11] approach. Tree nodes are viewed through the common interface `Node`. Its method `nodeList()` returns the read-only list of its children and its method `getParent()` returns its parent. The interface `Node` is implemented by all specific classes generated by Tatoo and a generic list of children is lazily generated from their attribute list, on demand.

For instance, given the following grammar in BNF notation, the type hierarchy represented in figure 5 is generated by Tatoo tools.

```
p1: expr ::= expr '+' expr
p2: expr ::= identifier
p3: identifier ::= '!' 'identifier'
```

Generated trees accept two kinds of visitor: a generic one and a specific one that respectively correspond to the different views of the tree. Here is a simplified version of the generic visitor `NodeVisitor`:

```
class NodeVisitor<R,P,D,E extends Throwable> {
    R visit(Node node,P param) throws E {
        throw new RuntimeException();
    }
}
```

This visitor is parameterized by the signature of the method `visit()` and only permits a simple traversal of the tree.

A strongly typed visitor of the generated type hierarchy is also produced. It inherits from `NodeVisitor` and, by default, its methods `visit()` delegate their implementation to a less precise method according to the type hierarchy.

For instance, if we consider the previous grammar and the type hierarchy of figure 5, the generator produces a visitor that contains the following methods:

```
R visit(P1 p1, P param) throws E {
```

```

    return visit((Expr)p1, param);
}
R visit(Expr e, P param) throws E {
    return visit((Node) e, param);
}

```

This delegation model allows developers to implement general behavior for some types and specific ones for others. More precisely, they do not have to implement all methods `visit()` if one of the super-type visitor captures all behaviors of its subtypes. For instance, if the behavior of the visit is the same for nodes of types P1 and P2, the developer only has to implement the method `visit(Expr e, ...)`.

3.3.4 Runtime lexer and parser

By default, the lexer and the parser are generated offline by the Tatoo generators using XML specifications. However, it is possible to construct them entirely at runtime specifying everything in Java. Indeed, XML specifications are only available for convenience. Internally, rules or productions are reified into Java objects that the developer may construct directly.

For instance, the following portion of code creates a rule with id `value` recognizing numbers using a rule factory. A similar factory is also available for grammar dynamic specification.

```

RuleFactory rf = new RuleFactoryImpl();
Encoding charset = LexerType.unicode.getEncoding();
Map<String,Regex> map =
    Collections.<String,Regex>emptyMap();
PatternRuleCompilerImpl ruleCompiler=
    new PatternRuleCompilerImpl(map,charset);
ruleCompiler.createRule(rf,"value","[0-9]+");

```

Then, given the description of rules, it is possible to directly obtain a lexer without intermediate code generation. However, lexer creation induces several costly computations at runtime, such as automata minimization.

The following portion of code constructs a lexer given a rule factory `rf`, a character buffer `b` and a lexer listener `l`.

```

SimpleLexer lexer =
    RuntimeLexerFactory.createRuntimeLexer(rf,b,l);

```

The lexer constructed this way is then almost as efficient as offline-constructed lexers.

4. RELATED WORKS

In this section, the overview of existing compiler generators is restricted to those written in Java and producing Java implementations. Of course, many other tools exist for other languages.

4.1 JavaCC

Developed by Sun Microsystems, Java Compiler Compiler [10] is a parser generator written in Java generating top-down parsers.

JavaCC by default only works for LL(1) grammars and let the user annotate the grammar with a specific k lookahead to resolve ambiguities.

The semantic part of the parser is directly integrated in the grammar specification. The actions are specified in Java and some keywords allow to obtain parsing information. The

fact that grammar and semantic part are mixed in the same file is error prone with large grammar and does not permit to reuse the grammar without its semantics.

4.2 ANTLR

ANother Tool for Language Recognition [13] written by Terence Parr, is a parser generator generating top-down compilers from grammatical specifications with back-ends in Java, C#, C++, or Python.

Like JavaCC, ANTLR is a LL(k) parser generator. The grammar is augmented with actions specifying the semantics of the compiler or translator. ANTLR provides a domain specific language for tree construction, tree walking and tree transformation.

4.3 JFlex/Cup

JFlex [7] and Cup [8] are ports of Lex and Yacc in Java. JFlex is a lexer generator in Java that reuses the same regex format than Lex on the unicode charset. Cup generates bottom-up compilers from LALR(1) grammar using a syntax similar to Yacc. Like JavaCC, it mixes the grammar specification and the semantic actions specified in Java.

4.4 SableCC

Sable Compiler Compiler [5] was initially written by Etienne Gagnon during its master thesis. It generates a bottom-up compiler from a LALR(1) grammar. However, it does not allow to resolve conflicts with associativity and priorities.

SableCC does not permit to specify semantics in the grammar but it generates an AST letting the developer rely on visitors to express the semantics. Thus it provides a clean separation between grammar specification and semantics. However, even if the generated AST is tweakable there is no way to generate a specific tree without creating another AST.

4.5 Beaver

Beaver [3] is an LALR(1) parser generator that claims to generate fast compilers.

Like Tatoo, beaver generates only a parser table and relies on a runtime parser. Furthermore the table contains actions to perform using late-binding call that frequently outperform the traditional switch implementation.

5. CONCLUSION

In this paper we have presented Tatoo a new compiler generator written in Java which main innovative features are:

- push lexing and parsing in order to support non blocking IO;
- support for grammar evolutions.

Tatoo has been used this year for compilation courses. During these courses, students have implemented a compiler for a simple language that produces Java bytecode. We did not encounter special difficulties induced by Tatoo's specific architecture compared with SableCC used previous years. However, we still have to implement a parser for a real language such as Java or C and to perform benchmarks to prove usefulness of Tatoo features.

In order to enhance Tatoo usability, we plan to support complete EBNF specification. Second, we also want to add

automatic documentation comparable to Javadoc in input specifications. Finally, other language back-ends, such as C++ and C#, will be added.

The latest stable version of Tatoo is freely available on the Web and can be downloaded from <http://tatoo.univ-mlv.fr/>.

6. REFERENCES

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [2] Apache Software Foundation. Velocity 1.4 User Guide. <http://jakarta.apache.org/velocity/>, Jan. 2006.
- [3] A. Demenchuk. Beaver - a LALR parser generator. <http://beaver.sourceforge.net/index.html>, 2006.
- [4] J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, 2nd edition, July 2002.
- [5] E. M. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *Technology of Object-Oriented Languages and Systems*, pages 140–154. IEEE Computer Society, 1998.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [7] K. Gerwin. *JFlex User's Manual*, July 2005.
- [8] S. E. Hudson. *CUP User's Manual*. Usability Center, Georgia Institute of Technology, July 1999.
- [9] S. C. Johnson. Yacc: Yet Another Compiler Compiler, 1979.
- [10] V. Kodaganallur. Incorporating language processing into java applications: A JavaCC tutorial. *IEEE Software*, 21(4):70–77, Aug. 2004.
- [11] A. Le Hors and P. Le Hégarret. Document object model level 3 core. <http://www.w3.org/DOM/DOMTR>, Apr. 2004.
- [12] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical Report 39, Bell Laboratories, July 1975.
- [13] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.
- [14] Sun Microsystems Inc. Java(TM) architecture for XML binding specification. <http://java.sun.com/xml/downloads/jaxb.html>, Jan. 2003.