UDC 004.428

A simple implementation of grammar libraries

Julien Cervelle, Rémi Forax and Gilles Roussel

Université Paris-Est
Laboratoire d'informatique de l'Institut Gaspard-Monge UMR CNRS 8049
5 bd Descartes, 77454 Marne la Vallée Cedex 2, France

Abstract. This paper presents an extension of the Tatoo compiler compiler that supports separate compilation and dynamic linking of formal grammars. It allows the developer to define reusable libraries of grammars such as those of arithmetic expressions or of classical control operators. The aim of this feature is to simplify the development of domain specific languages especially for non specialists in grammar writing.

1. Introduction

This paper presents an extension of the Tatoo compiler compiler [1] that provides the developers of formal grammars with separate compilation. It allows to use predefined libraries of general purpose grammar, possibly with its semantic support. Surprisingly this feature is not available in popular compiler compilers such as Bison [2], ANTLR [3] or JavaCC [4].

We consider that such a mechanism would simplify the elaboration of formal grammars. This is especially necessary for non-specialists of grammar writing who want, for instance, to develop small domain specific languages (DSL) without being bothered by the cumbersome design details of long-time and oftentimes already written grammars. Analogous module mechanisms are available in LISA [5] or Rats! [6]. However, they work at source level and need re-compilation of all the parts of the grammar.

By separate compilation, we mean that the developer should be able to split the specification of the grammar into several sub-parts and to compile each independently into parsing tables. Afterwards, they can link these grammar fragments together dynamically with minimum overhead. With such a mechanism it is then possible to create pre-compiled libraries, such as arithmetic expressions or classical control structures, and to reuse them in different contexts at no extra cost. Modules separate compilation provides independent reporting of table construction errors, simplifying the tuning of grammars. There can be no LR conflicts neither coming from precompiled libraries nor due to the linking phases, since the global LR table is never constructed. Moreover, Tatoo architecture allows semantics to be embedded along with pre-compiled grammars, and possibly several different semantics for various purposes (building abstract syntax trees, writing intermediate code or java bytecode, etc.).

The base principle of the separate compilation provided by Tatoo is to construct special parser tables for incomplete grammars and to be able, at runtime, to call another parser when necessary. The grammar is incomplete in special places, called branch points, that are place holders for other grammar start symbols. The dynamic linking only requires specifying the precise grammar to be plugged in the branch point. At runtime, the basic idea is to switch from one parser (grammar) to another in branch point when a parse error is encountered. Note that with this approach there is no backtracking, the branching is done only when an error occurs. The current running parser is always of utmost importance. When the running parser is unable to recognize the input sequence, either a new parser is called or the error is considered as the end-of-input for the current parser. Due to this specific behavior, the language accepted by the parser may be slightly different from the one recognized without separate compilation (see Section 3..6) for some uncommon grammar cutting.

The rest of this paper is organized as follows. Section 2 presents an short overview of Tatoo. Section 3 provides details about the separate compilation and dynamic linking in Tatoo. Section 4 presents a complete example illustrating these features. Section 5 presents related works and a conclusion.

2. Tatoo overview

Given a set of regular expressions, a formal grammar and several semantic hints, Tatoo, generates Java implementations for a lexer, a parser (SLR, LR(1) or LALR(1)) and several implementation glues allowing to run a complete analyzer that creates trees or computes simpler values. Thanks to a clean separation, the lexer and the parser may be used independently. Moreover, these implementations are independent of a particular semantic evaluation and may be reused, in different contexts, without modification.

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 parametrized types and enumeration facilities to specialize this runtime support code. All memory allocations are performed at creation time.

One main feature of the generated lexer and parser resides in their 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. 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. A classical pull implementation based on this push implementation is also provided.

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 regular expressions according to the parser context: the lexer only tries to recognize the token that are in the expected lookaheads of the current state of the parser. This feature permits, in particular, the activation of keywords according to the version only when they are expected [1]. For instance, one could name a local variable in a Java method `public`.

3. Grammar library implementation

In this section the details of the mechanisms that provide the separate compilation and the dynamic linking are presented.

3.1. Specification

In order to provide separate compilation, the developer must be able to specify incomplete grammars. This feature is supported by three of Tatoo's mechanisms in the grammar specification.

First, to specify an incomplete grammar, the developer omits some parts of the grammar specifying branch points that are special non-terminals. For each of these branch points, at runtime, another grammar (possibly also incomplete) start symbol is associated. In order to differentiate intentional splitting of the grammar from unused non-terminals (developer's errors that are statically checked) the non-terminals corresponding to branch points are explicitly designated. This is done through the special keyword `branches` in the grammar specification.

Second, to enable different input points in the grammar, it is possible in Tatoo to specify several start symbols [1]. In the context of the construction of libraries, this mechanism permits, for instance, the factorization of similar parts of different grammars into a single one, allowing the semantics associated with it to be shared. Multiple start symbols may be specified in Tatoo using the `starts` keyword.

Finally, the notion of version available in Tatoo [1] allows the developer to make its library grammars evolve without breaking the backward compatibility for the users of preceding versions.

3.2. Compilation

The compilation process of incomplete grammars consists of two steps. First, non-terminals corresponding to branch points are considered from the compilation point of view, exactly as classical terminals of the grammar (even if from a grammar theory point of view, the branch points behave more like non-terminals) and the construction of SLR, LR(1) or LALR(1) tables are produced by the Tatoo engine. Then, the base idea is to consider that parser branching or exiting will be performed when an error occurs. The semantic of this behavior is to consider that the running parser is of utmost importance since Tatoo branches to a sub-parser only when an error occurs. These errors may correspond to a branch point or to the end-of-input of a sub-parser or to real parsing errors. Thus, for all of the states where there is a non-error action associated to a lookahead corresponding to a branch point or the end-of-input terminal, Tatoo statically replaces the error action with an access to the branching table.

The branching table associates each state with the action to perform if an error occurs. There are four kinds of actions in the branching table: a classical reduction, an enter, an exit or an error.

The branching table is computed from the regular LR action table using the following algorithm:

```
For each state $s$ in the LR table:
    If $s$ is an accepting state:
        add an exit action in the branching table for $s$
    For each shift in $s$ via a branch point $t$:
        add an enter $t$ action in the branching table for $s$
        remove from $s$ the shift $t$ action in the LR table
    For each reduction of a production $p$ in $s$ with lookahead $l$:
        If $l$ is a branch point or the end-of-input terminal:
            add a reduction of $p$ action in the branching table for $s$
    If no action has been added for $s$ in the branching table
        add an error action in the branching table for $s$
```

The behavior of the actions in the branching table is:

- the enter t action notifies that the parser associated with t should be started;

- the exit action notifies that the current parser must exit and that calling parser must be resumed. It raises an error if the latter does not exists (i.e. the current parser is at top level).

The translation of actions of the regular table into actions the branching table is summarized below:

Lookahead	Action	Branch action
end-of-input	accept	exit
end-of-input	reduce	reduce
branch point	shift	enter
branch point	reduce	reduce

During the construction of the branching table some conflicts may occur since several actions can be associated to the end-of-input and/or multiple branch points in a given state. Among classical *shift-reduce* and *reduce-reduce* conflicts, new conflicts involving enters and/or exit could occur. For instance, if there is a state where two branch points are possible. Indeed, since, in our approach, the actions associated to end-of-input and the branch points all go into the same table, it is sometimes impossible to choose which action to perform on errors: enter, reduce or exit. These conflicts are detected statically when the branching table is constructed for the incomplete grammar. They may be resolved in Tatoo using priorities associated by the developer with the end-of-input, the branch points and the productions.

3.3. Dynamic linking

Now that we have provided separate compilation of grammar tables, the tables of different grammars have to be linked together to obtain a global parser. The user has to specify the association between the branch points and the sub-grammars. The association provides the start symbol and the version to be used by the sub-parser. This declarative specification is used to generate a Java implementation that registers proper listeners for enter and exit actions. It also creates an instance of each lexer and parser that will be necessary during the parsing process so that no new memory is needed during the parsing, except possibly parser stack extension.

3.4. Runtime

At runtime, the top level lexer and parser are started like classical Tatoo parsers. If a token is recognized by the lexer, since Tatoo runtime only activates rules corresponding to the terminals leading to non-error actions, the parser never fails and the classical table is used to retrieve the action, shift, reduce or accept that needs to be performed. When a lexer error occurs the branching table of the current grammar is queried.

If a reduce action is encountered, it is performed, a new state is reached in the branching table and a new action has to be perform.

If an enter action is found, the registered listener looks at the lexer and parsers instances to be activated, pushes a new stack frame to allow recursive calls to the same grammar and call the new parsing process. Moreover, the input characters previously read by the lexer before the error occurred, must be “pushed back” .

If it is an exit action, the registered listener closes the parser and the lexer, returns to the enclosing parser and lexer and restarts them in a state reached after the execution of the shift of the branching point.

Otherwise, a parsing error is thrown and the error recovery policy is activated.

The Figure 1 displays a diagram of the connections between lexers, parsers and other mechanisms.

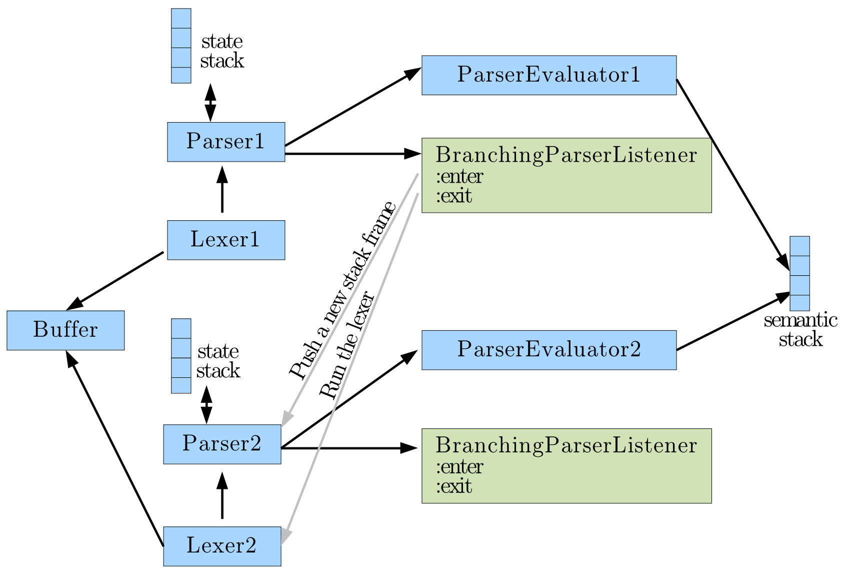

Fig. 1. Transfer between lexers and parsers at runtime

However, in special cases, the enter and the exit actions may result in a runtime error. The enter action may result in an error if no listener has been registered by the dynamic linking phase. These kinds of errors can be warned by the linking phase, checking that every branch point of a grammar is associated to a start symbol of ``another'' grammar, but are not considered errors in case a developper does not want to use all the branch points of the grammar library. The exit action may also result in an error if the parser is the top-most in the stack.

3.5. Dealing with the semantics

Contrary to most parser generators [3, 4, 7, 8, 2] Tatoo's grammar specification does not embed any foreign languages (like C, Java etc.) to express the semantics [1]. Tatoo's specification requires a name for each production and a type for each terminal and non-terminal. Based on this information Tatoo generates a Java interface that allows the developers to implement the semantics. Thus, the semantics of grammar libraries is easily reusable.

A developer can choose to provide a single class for the whole parser by implementing all of the semantic interfaces or to reuse library classes implementing the interface of each grammar semantic part. The sharing of the semantic stack allows the semantics to work altogether: when the embedded parser exits, the result of its evaluation is pushed on the common semantic stack. The enclosing parser accesses the value popping it from the stack.

3.6. Separate compilation versus source level approach

Given a grammar decomposed into several parts, the set of cooperating parsers produced using Tatoo's separate compilation mechanism do not recognize exactly the same language as the parser produced if the different grammar parts are combined into a unique source file. Indeed, in our approach the table construction is purely local and there is no information propagation between the different grammar parts. In particular, transitive closure of states in the LR table construction does not (and could not) propagate through non-terminals corresponding to branch points.

For instance, given the following grammar:

```
A = B 'y'
B = B 'y' | 'y'
```

The language recognized by this grammar is $\mathtt{yy}^+$. Supposing that the grammar is decomposed into two parts: one for `A = B 'y'` and one for `B = B 'y' | 'y'` where `B` is a branch point. The set of cooperating parsers produced by Tatoo's separate compilation do not recognize any input sequence. Indeed, given the input sequence `yy` the sub-parser associated with the first production leads to an error on the branch point `B` since the ``internal'' terminal `'B'`, which is the only one expected, is never found in the input sequence. Using the branching table, the parser activates the sub-parser corresponding to the second production. This new parser recognizes `y+` until the end-of-input is reached. Then, the parser associated to the first production is restarted with a shift on the ``internal" terminal `'B'`. Since the input sequence no longer contains `y` anymore, an error is produced and the classical error recovery mechanism is called.

This kind of problem occurs when there is a race for the same terminal between the enclosing parser on reset and the internal parser before the end-of-input.

Furthermore at lexer level: different regular expressions may recognize a same input sequence. It is even more complex since one regular expression may only recognize a prefix of input sequence recognized by another. However, this problem is also present in source level approaches. Indeed, lexer specifications usually rely on regular expression priorities based on textual order. Merging these specifications also raises the problem of a global order on the regular expressions which is comparable to ours.

Since one of the aims of Tatoo was to allow efficient ``synchronous" parsing, backtracking has not been considered for solving this problem. We consider that this kind of breaking down of the grammar should not be very common in practice since grammar parts should correspond to loosely coupled self-contained modules.

3.7. Library approach versus inheritance

For the sake of simplicity we have chosen to use our library mechanism to extend incomplete grammars in special places. However, a comparable mechanism could be used to implement an inheritance mechanism for complete grammars.

The inheritance approach has the advantage of freeing the developer from the specification of branch points. Any non-terminal of the grammar may be extended with special library, simplifying the reusability of grammars.

However, it has drawbacks. To permit the extension in any non-terminal, the grammar has to be extended by Tatoo. A new production must be created for each non-terminal deriving in a new branch point. This enlarges the tables produced by Tatoo and may lead to conflicts. Indeed, for each empty production, there is potentially an enter-enter conflict.

For instance, if the grammar is the following:

```
S = A B;
A = 'a' | ;
B = 'b';
```

It is extended as follows by Tatoo:

```
S = A B;
A = 'a' |
  | branchA;
B = 'b'
  | branchB;
```

Then, a conflict appears in the branching table between actions enter `branchA` and enter `branchB`. The conflict may be solved using priorities on non-terminals.

This approach to support inheritance is not very efficient. Indeed, most of the information added in the tables is not used and the conflicts usually never appear.

4. Example

In this section, we present a complete example demonstrating how Tatoo's separate compilation feature may be used to develop "grammar libraries" and how it helps people who are not accustomed to writing grammars to incorporate them into simpler grammars.

Supposing that one wants to design a simple DSL for an application, such as simple routines to automate tasks in an application. Usually, the syntax for the routine carries the form of a well-known language but with special entry points and special end points and constructions. For instance, in a spreadsheet program, the programming language consists of mathematical expressions and the entry point is an "=" symbol, and the end points are the special naming of cells (like `A3:A4`, `A$3`, etc.) and function call..

4.1. Defining the library

Then, to implement such a spreadsheet program, one could use an already written and compiled grammar library for all mathematical expressions and control structures.

The following grammar is an example of incomplete grammar written in Tatoo's EBNF format that defines this library:

```
priorities:
  plus_minus= 1 left
  star_slash= 2 left
tokens:
  lpar= '\('
  rpar= '\)'
  plus= '\+'  [plus_minus]
  minus= '-'  [plus_minus]
  star= '\*'  [star_slash]
  slash= '\/' [star_slash]
blanks:
  space= "( |\t|(\r?\n))+"
```

```
branches:
  idBranchPoint
starts:
  expr
productions:
  expr = expr 'plus' expr    [plus_minus]
       | expr 'minus' expr   [plus_minus]
       | expr 'star' expr    [star_slash]
       | expr 'slash' expr   [star_slash]
       | 'lpar' expr 'rpar'
       | idBranchPoint
       ;
```

In this specification one should notice in the `branches` section, the use of the branch point `idBranchPoint`. This allows the developer to choose their own atomic expressions: the spreadsheet endpoints.

This grammar is then taken as an input of the Tatoo compiler. It generates the LR table together with the branching tables related to this grammar. Thanks to priorities, there is no conflict in these tables.

4.2. Using the library

If the ``arithmetic expressions'' library is available, the user only needs to write the grammar specific to the spreadsheet application. This specification has to contain a start symbol `start` for the global parser and a special start symbol `endpoint` to define the syntax for the cell description and functions.

Using the Tatoo ebnf syntax[1], the spreadsheet grammar may be specified as follows:

```
tokens:
  assign= '='
  comma= ','
  colon= ':'
  lpar= '\('
  rpar= '\)'
  dollar= '\$'
  number= '[0-9]+'
  identifier= '[a-zA-Z]+'
blanks:
 space= "( |\t|(\r?\n))+"
branches:
  exprBranchPoint
starts:
  start
  endPoint
productions:
  start = 'assign' exprBranchPoint
```

[1]The notation `exprBranchPoint/'comma'*` means zero or more `exprBranchPoint` separated by `comma`.

```
       ;
  endPoint = 'dollar' 'identifier' 'dollar'? 'number' range?
           | 'identifier' identifierOrFunction
           ;
  identifierOrFunction = 'lpar' exprBranchPoint/'comma'* 'rpar'
                       | 'dollar'? 'number' range?
                       ;
  range = 'colon' 'dollar'? 'identifier' 'dollar'? 'number'
       ;
```

This specification contains one branch point `exprBranchPoint` which is the place holder for the arithmetic expressions. The arithmetic expression library calls back this grammar to recognize the end points (identifier or function) of the spreadsheet application. Thus, this specification contains two parts related by the arithmetic expressions library.

Note that to work properly, neither identifiers nor functions should appear as a token in the arithmetic expressions library. Should this be the case, an input sequence such as `A3:B4` would not be recognized properly (see section 3..6). Indeed, in this case `A3` would be recognized as a function identifier by the library and the error would occur on the `':'`. For sake of efficiency, we do not want to backtrack on errors and `':'` is not a correct branch point so the branching fails. However, as libraries are usually provided by grammar-aware individuals, they are typically well-written and DSL writers should not be aware of this difficulty.

4.3. Dynamic linking

The last part of the specification has to do with wiring up the grammars together. This is done with a specific Tatoo language. The specification details the files that contain the different grammar parts[2] and the associations between the branch points and the grammar start symbols together with their version.

For instance for the previous example this file looks like[3]:

```
grammars:
  expr= "grammars/expr/expr.tlib"
  spreadsheet= "grammars/spreadsheet/spreadsheet.tlib"

links:
  spreadsheet.exprBranchPoint = expr.start
  expr.idBranchPoint = spreadsheet.endPoint
```

This file is translated into a Java code. This implementation provides a global analyzer that creates one lexer and one parser for each grammar. All the lexers share the same input buffer. To support recursive calls of grammars, like the

[2]Compiled parsers are packed-up into a specific `Jar` file with extension `.tlib` that contains their tables together with necessary meta-information.

[3]Since our grammar examples do not contain version, no version information appears in this file.

spreadsheet example, a each parser uses its own stack frame. The stack for the semantics is shared by all the parsers.

Each parser is configured using a parser listener. Each listener contains an association between a branch point and a pair lexer/parser to be started on a given start symbol.

5. Related works and conclusion

As far as we know there is no parser generator that supports the separate compilation of grammars.

Several tools such as ANTLR2 [3], Rats! [6] or LISA and its extensions [5, 8] provide built-in features that simplify software engineering and modular development. However, these tools require the construction of a global specification before the generation of tables. They can be viewed as pre-processors in the compiler compiler chain.

The ELL Parser Library [9] is a tool that supports the dynamic extension of parsers. It allows the developer to extend the grammar at runtime and uses a backtracking mechanism to solve the ambiguities in the LL table extended dynamically. It proposes a mechanism to suspend the parser at runtime which is comparable to Tatoo's one.

Tatoo's approach is not in contradiction with the previous approaches. However, to preserve online parsing with low memory consumption, we chose not to introduction backtracking in Tatoo's implementation. Moreover, Tatoo's modularity works at a lower level. It avoids the construction of a global grammar before the generation of a parser and it sees the parser as a set of small cooperating parsers. A very simple semantics has been adopted to follow or to exit branch points in the grammar. The current running parser is always of the utmost importance and branching points, or leaving of the parser are only considered when the lexer detects an error. Priorities may be used to solve potential conflicts.

Further work under consideration involves checking if other mechanisms that support modularity such as aspect programming can be implemented using separate compilation. We could also use a Tatoo library mechanism to implement dynamic extensions of the grammar since Tatoo already supports the creation of tables at runtime.

Moreover, it is still not clear whether or not this splitting approach could extend to the class of global grammars accepted as input for Tatoo. Further investigation needs to be undertaken in this direction.

Acknowledgments

Thank you to Marjan Mernik and Boštjan Slivnik for their fruitful discussion.

6. References

1. Cervelle, J., Forax, R., Roussel, G.: Tatoo: An innovative Parser Generator. In: Proc. of the 4th Int. Conf. PPPJ'06. ACM International Conference Proceedings, Mannheim, Germany (August 2006) 13--20
2. Aaby, A.: Compiler construction using flex and bison. `http://cs.wwc.edu/~aabyan/464/Book/` (1996)
3. Parr, T.J., Quong, R.W.: ANTLR: A predicated-LL(k) parser generator. Software Practice and Experience **25**(7) (1995) 789--810
4. Kodaganallur, V.: Incorporating language processing into java applications: A JavaCC tutorial. IEEE Software **21**(4) (August 2004) 70--77
5. Mernik, M., Korbar, N., Zumer, V.: LISA: A tool for automtic language implementation. SIGPLAN Notices **30**(4) (1995) 71--79
6. Grimm, R.: Better extensibility through modular syntax. In: Proc. of PLDI'06. (2006) 38--51
7. Gagnon, E.M., Hendren, L.J.: SableCC, an object-oriented compiler framework. In: Technology of Object-Oriented Languages and Systems, IEEE Computer Society (1998) 140--154
8. Rebernak, D., Mernik, M., Henriques, P.R., Pereira, M.J.V.: AspectLISA: An aspect-oriented compiler construction system based on attribute grammars. Electr. Notes Theor. Comput. Sci. **164**(2) (2006) 37--53
9. Plesner Hansen, C.: An Efficient, Dynamically Extensible ELL Parser Library. PhD thesis, University of Aarhus, Denmark (May 2004)