RAPID PROTOTYPING PARSER GENERATOR

Jiří Zuzaňák
Doctoral Degree Programme (1), FIT BUT
E-mail: izuzanak@fit.vutbr.cz

Supervised by: Pavel Zemčík
E-mail: zemcik@fit.vutbr.cz

ABSTRACT

This paper discusses the design and implementation of a parser generator (compiler-compiler). The basic requirements on the generator are simplicity and speed of parser design. Further in the text we discuss the structure of the generator input, and demonstrate the functionality of the generator on an example. The implementation language of the generator, and also of the generated parser is `C/C++`.

1 INTRODUCTION

Parser generator is a program that transforms textual parser description to the parser itself. At the moment, language parsers are no more designed without using tools created for this purpose. A set of these tools contains a syntactic and lexical analyzer generators. The most familiar tools created for this purpose are lex and flex lexical analyzer generators, and yacc and bison syntactic analyzer generators (see details in [3]).

The parser rules and the parser lexical symbols are described by textual representation in the generator's input. The syntax analyzer is usually described by a syntax grammar in *BNF* (Backus-Naur) form. *BNF* form of grammars developed by John Backus and Peter Naur is widely used as a notation for grammars of computer programing languages

The lexical analyzer of a parser is described by a set of regular expressions (*REG*). Each regular expression describes one terminal symbol (token) of the syntactic grammar.

The parser created by a generator is represented either by a set of structures in the generator's program memory, or stored as a source input for some programing language. The generated parser is in most cases separated from the lexical analyzer that is created by the designer or generated by a separate program.

2 PARSER GENERATION THEORY

Each regular expression from the generator's input must be transformed to a deterministic finite automaton (*DFA*). The theory of formal languages covers methods and algorithms describing the transformation of a regular expression to a finite automaton, and also covers algorithms describing the transformation of nondeterministic *FA* (*NFA*) to deterministic *FA*. Some of the

methods and algorithms are described in [2]. Automatons generated by this way are represented as memory structures or are coded to an output parser source.
Syntactic rules are defined by a grammar written in *BNF* form. The theory of formal languages describes algorithms for generating *LL* (left to right, left-most derivation) and *LR* (left to right, right-most derivation) parse tables from language grammar rules (detail information in [1]). There are two standard approaches to parsing: top-down and bottom-up. During a typical top-down parsing such as in *LL(1)*, the input is predicted and the prediction is verified against the real input. Bottom-up parsers shift input symbols on a stack, until the entire rule body is stored at stack top, and then it is recognized. This approach has been employed by parsers of *SLR(1)* (Simple *LR*), *LR(1)*, and *LALR(1)* (LookAhead *LR*) languages. For most grammar types, including *LL(1)*, *LL(k)*, *SLR(1)*, *LALR(1)* and *LR(1)* grammars, there are existing parser generators.
An algorithm used for generating parse table from *SLR(1)* grammar improves the *LR(0)* algorithm by looking at an additional lookahead. This modification helps to avoid erroneous reductions and thus avoid certain reduce-reduce and shift-reduce conflicts. The *SLR(1)* lookahead set is equal to *follow(A)* for a rule with head nonterminal *A*. *LALR(1)* improves the *SLR(1)* algorithm by attempting to do more careful lookahead analysis. In *LALR(1)*, is a computation of the lookahead set based on viable prefixes of a grammar. A full *LR(1)* grammar table is also called canonical *LR(1)* table. These grammars are not used often because *LR(1)* tables could be up to ten times bigger than *LALR(1)* tables.

3 RAPID PROTOTYPING PARSER GENERATOR

To generate parser source, compile it and test if the parser works well after each change of the parser grammar would be inefficient. A more efficient approach for the parser designer is to test the described parser directly on programs, even after minor changes to the parser grammar.
To meet the need for designing a script language intended for image processing, a parser generator was created, that is described by this paper. The generator was designed for fast parser prototyping and testing. It contains an interpret of a simple scripting language that can be used for describing programs attached to each rule of the parser grammar.
While testing the described parser on sample input program, these programs are executed every time when a reduction of a rule attached to the program occurs. The rapid prototyping parser generator scheme is shown in Figure 1.
This feature of the generator can be used for testing the designed parser on samples of input code. For example the designer can attach to each rule an output describing its form, and by checking the output text determine if the grammar works right or not. We show an example of usage of this feature at the example in section 4.

3.1 LEXICAL ANALYZER

Each lexical symbol in the parser generator input is described by a regular expression. These expressions serve as unambiguous identification of the lexical tokens. In contrast to other automated tools for generating lexical analyzers, every token must be described by exactly one regular expression. The regular expressions cannot be composed from other expressions. This approach is quite limiting, but the generator defines a large amount of build-in symbols for representing families of symbols such as digits, letters, white symbols and others. On the other

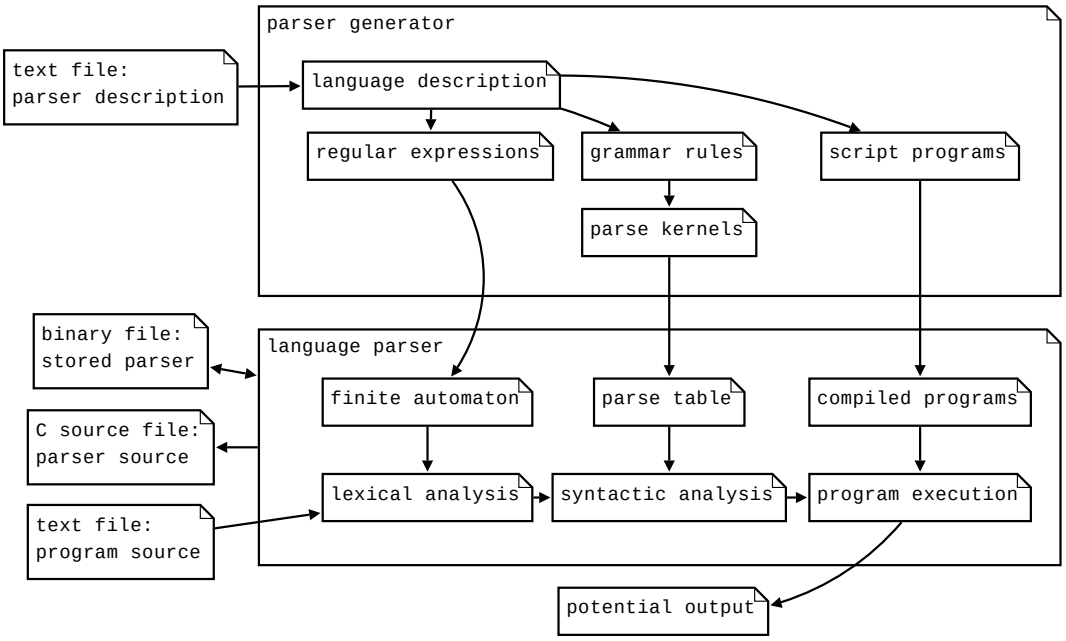

Figure 1: Parser generator scheme.

hand this approach simplifies the process of creating the lexical finite automaton by the generator. This automaton is optimized (is minimal and is composed from distinguishable states) and each of its final states corresponds to exactly one token.

3.2 SYNTACTIC ANALYZER

We chose the bottom-up approach for our parser generator. The generated syntactic analyzer is described by a *SLR(1)* grammar defined by its rules composed from terminal and nonterminal symbols. The grammar must be defined in *SLR(1)* form because the generator solves the shift-reduce and reduce-reduce conflicts only by testing the follow set of the rule's head nonterminal. A new version of the software will use lookahead method for solving the shift-reduce and reduce-reduce conflicts, and so will enable usage of *LALR(1)* grammars.
As is mentioned before, each grammar rule is bounded to a program whose code is executed when reduction by the rule is performed. This program is described in a simple built-in scripting language which allows basic constructions such as conditions, loops, access to terminal strings and data types definitions.

3.3 INPUT STRUCTURE

The input of the generator must meet the structure displayed in table 1. The first part of the input defines the script code executed at the start of parsing. This code is used for declaration and initialization of variables. Each variable defined in the script is global (shared by all programs) in the simulation of the parsing process. The next part of the input indicated by the `terminals` keyword, is used for declaration of lexical symbols, and definition of their regular expressions. The regular expressions of the lexical symbols are enclosed in curly brackets. The third part consists of a list of nonterminal symbols of the grammar which are enclosed in angle brackets. This part is determined by the `nonterminals` keyword. All terminals and nonterminals

```
init_code: {<initialization code>}

terminals: /* list of terminals */
   <terminal identifier> {<regular expression>}

nonterminals: /* list of nonterminals */
   <<nonterminal identifier>>

rules: /* list of rules */
   <rule head> -> <rule body_0> ... <rule body_x> ->> {<reduction code>}
```

Table 1: Generator input structure

```
S -> E          T -> F
E -> E + T      F -> id
E -> T          F -> ( E )
T -> T * F
```

Table 2: Example grammar

must be unique symbols that are later used in grammar rule descriptions. The last and most important part of the input is denoted by the keyword rules and contains the list of the grammar rules. Each rule is composed of the rule head defining nonterminal separated by string `->` from the rule body, which is separated by string `->>` from the program attached to this rule. The rule body consists of a list of terminals and nonterminals separated by whitespace. The script program assigned to each rule is enclosed in curly brackets.

3.4 GENERATOR OUTPUT

There are more ways how the generator can handle with the generated parser. The generator could store the parser to a binary file, use the parser for simulated parsing of a program or generate a parser `C/C++` source.
`C/C++` source created by the generator contains a *LALR* table and two functions. These functions implement lexical and syntactic analyzers of the compiler. The result obtained by compiling this source is executable binary, which, given appropriate input programs (passed as arguments), prints out its right-most derivation.

4 EXAMPLE OF GENERATOR INPUT

This section describes an example of a simple input of the generator. The input describes a simple parser designed for parsing expressions, generated by the grammar shown in table 2. Table 3. shows the input text describing the example grammar. The first part of the input is formed by initialization of the script and definition of global variables, the example contains blank initialization code denoted by the keyword `null`.
Definition of the terminal symbols *id*, *plus*, *asterisk*, *lr_br* and *rr_br* follows. Terminals whose identifier contains string `SKIP` are ignored, and terminals whose identifier contains string `END` are considered as last symbol of the input. The next part of the input defines nonterminal symbols *S*, *E*, *T* and *F* that must be enclosed in angle brackets. We define one extra nonterminal called *start*. Each rule (except the first rule) contains a command that prints the structure of the rule to standard output.

```
init_code: { null }
terminals:
   id {('_'+l).('_'+l+d)*} plus {'+'} asterisk {'*'} lr_br {'('} rr_br {')'}
   _SKIP_ {w.w*}
   _END_ {'\0'}
nonterminals: <start> <S> <E> <T> <F>
rules:
   <start> -> <S> _END_      ->>   {null}

   <S> -> <E>                ->>   { out("S <- E \n"); }
   <E> -> <E> plus <T>       ->>   { out("E <- E + T \n"); }
   <E> -> <T>                ->>   { out("E <- T \n"); }
   <T> -> <T> asterisk <F>   ->>   { out("T <- T * F \n"); }
   <T> -> <F>                ->>   { out("T <- F \n"); }
   <F> -> id                 ->>   { out("F <- id(",rule_body(0),") \n"); }
   <F> -> lr_br <E> rr_br    ->>   { out("F <- ( E ) \n"); }
```

Table 3: Generator input describing example grammar

```
F <- id(a)      F <- ( E )
T <- F          T <- F
E <- T          E <- E + T
F <- id(b)      F <- id(d)
T <- F          T <- F
F <- id(c)      E <- E + T
T <- T * F      S <- E
E <- T
```

Table 4: Output generated by parser (displayed in two columns)

The parser generated from the above input, can be used for parsing simple expressions. The output generated by the parser for expression `a + (b * c) + d` is displayed in table 4. As you can see, the parser prints a text representation of every performed reduction.

5 CONCLUSION

The created generator enables fast prototyping of a designed parser. The generator was successfully used to design a parser of scripting language intended for rapid prototyping of image operations, and image processing. The designed language has a structure similar to java language, it allows object oriented programing and its variables are dynamically typed. The main reason for creation of the described parser generator was the design of the mentioned script language.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison Wesley, 1985.

[2] Anne Bruggemann-klein. Regular expressions into finite automata. Technical report, February 21 1996.

[3] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.