Excel@FIT 2015
http://excel.fit.vutbr.cz

Parsers based on programmed grammars

Luboš Vaníček*

Abstract
Compilers that are used to translate programming languages are based on context-free grammars. That means that they aren't as strong as they could be. The power of compilers is the key problem that this work is trying to solve. The main part of a compiler that is related to this problem is a parser. And that is what this work is trying to improve. The problem is solved by creating a special algorithm based on programmed grammars. The algorithm is than used in parsers. A special application is created to verify if the solution works. This application allows users to test the algorithm on user-created grammars and an input string of symbols. The created program is tested with grammatical constructions that are known as context-sensitive. The results are really successful and promising for the future development and application. This work could influence constructions in programming languages and introduce brand new ones. It's a good starting point for stronger and better compilers. It could shorten the whole code and increase its clarity. Another application of this work could be in translators that are translating natural languages, because those languages are context-sensitive.

Keywords: programmed grammars — parsing — translators

Supplementary Material: *N/A*

*xvanic07@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

In today's world, the programming languages are based only on context-free grammars. That said, we cannot use context-sensitive constructions in those languages. The transition from current programming languages to context-sensitive languages would be dramatic. This could create an opportunity for better, shorter and more understandable programming code. The idea could be pushed even further and it may be applied to translators of natural languages.

We tried to create a parser based on programmed grammars. A parser is the main part of a compiler or a translator. The basic difference between the compiler and the translator is that the compiler is transforming a code and turns it into a machine language. The translator is transforming a code in some language to another language of comparable level (e.g. from C to Java). The parser's behavior is based on an algorithm. The algorithm finds out whether a user inputted string belongs to the language generated by a user-created programmed grammar. Another step was a creation of a program that implements this algorithm and allows users to test it properly.

There are no implemented solutions of compilers or parsers that are based on programmed grammars we know about. Parsers that are used in compilers for today's programming languages are based on context-free grammars. That makes them much weaker than programmed-grammar-based translators because they could not translate context-sensitive constructions in programming languages. Programmed grammars were only discussed theoretically and no practical approach has been published so far. My work takes this theoretical basics and uses them in parser application.

Programmed grammars are context-free grammars that are extended by mathematical features allowing the grammar to control the use of its rules. The algorithm that takes user-created programmed grammars and input string, uses the basic ideas of top-down parsing and pushes them forward. If none of the current rules can be applied on the left-most nonterminal than this algorithm tries the same set of rules on another nonterminals in the input string.

A parser based on programmed grammars was created and tested. Not only did it work on context-free languages but it was also able to recognize context-sensitive constructions. Not all context-sensitive constructions were tested but that would be almost impossible to do so. This remains an open problem that would need a mathematical proof. The program was designed for theoretical languages so it can only handle one-letter terminals and nonterminals. In the future development this limitation could be eliminated, so this parser would be fully applicable in compilers of programming languages.

2. Expected knowledge

Formal languages are the basics for this work. Those languages must be precisely mathematically defined. For this paper we assume that the reader knows formal languages on the level of context-free grammars. The information about context-free languages can be obtained from this publication [1]. The mathematical background that is also needed for the understanding of this paper can be read from this book [2]. We also expect that the reader knows basics about parsing. If not, the book [3] is a good starting point.

3. Programmed grammars

This paper is based on programmed grammars [4]. The programmed grammar modifies a context-free grammar by extending its every rule from a set P with a set of rules Q. Symbols in the Q set determine which rules from the set P can be used in a next derivation step. A formal definition of programmed grammars is as follows:

Definition 3.1. A programmed grammar is quintuplet:

$$G = (N, T, \psi, P, S),$$

where

- symbols N, T, ψ, S are defined as in a context-free grammar;
- $P \subseteq \psi \times N \times (N \cup T)^* \times 2^\psi$ is a finite relation called the set of rules. If $(r, A, x, q_r), (s, A, x, q_s) \in P$, q_r, $q_s \in Q$, then $(r, A, x, q_r) = (s, A, x, q_s)$.

Instead of $(r, A, x, q_r) \in P$, we write $r\colon A \to x, q_r \in P$.

Let $V = N \cup T$ be the total alphabet. The relation of a *direct derivation*, symbolically denoted by $\Rightarrow$, is defined over $V^* \times \psi$ as follows: for $(x_1, r), (x_2, s) \in V^* \times \psi$, $(x_1, r) \Rightarrow (x_2, s)$ in G if and only if

$x_1 = yAz$,
$x_2 = ywz$,
$r\colon A \to x, q_r \in P$,
$s \in q_r$.

3.1 Generated language

The language generated by programmed grammars is formally described in the following definition:

Definition 3.2.

$$L(G) = \{w \in T^* \mid (S, r) \Rightarrow^* (w, s);\ r, s \in \psi\}$$

Generating the string w continues until the result string is not terminal (consists only of terminal symbols). If no rule from the set of Q from the last used rule can be applied, then the derivation ends up unsuccessfully.

Example 3.1. Let $G = (\{S, A, B, C\}, \{a, b, c,\}, \{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}, P, S)$ be a programmed grammar. Let ε denote an empty string defined as in the book [1]. The set P consists of the following rules:

$r_1 : S \to ABC, \{r_2, r_5\}$,
$r_2 : A \to aA, \{r_3\}$,
$r_3 : B \to bB, \{r_4\}$,
$r_4 : C \to cC, \{r_2, r_5\}$,
$r_5 : A \to \varepsilon, \{r_6\}$,
$r_6 : B \to \varepsilon, \{r_7\}$,
$r_7 : C \to \varepsilon, \{r_7\}$.

The listed example generates a language $L(G) = \{a^n b^n c^n \mid n \geq 0\}$. An example of a string from $L(G)$ is *aabbcc*. This string can be obtained using the programmed grammar G this way:

(S, r_1)
$\Rightarrow (ABC, r_2)$
$\Rightarrow (aABC, r_3)$
$\Rightarrow (aAbBC, r_4)$
$\Rightarrow (aAbBcC, r_2)$
$\Rightarrow (aaAbBcC, r_3)$
$\Rightarrow (aaAbbBcC, r_4)$
$\Rightarrow (aaAbbBccC, r_5)$
$\Rightarrow (aabbBccC, r_6)$
$\Rightarrow (aabbccC, r_7)$
$\Rightarrow (aabbcc, r_7)$

4. Parsing using programmed grammars

We use the basics of top-down parsing – the input string is read from left to right. However, we cannot try to apply all the rules from programmed grammar G on the left-most nonterminal and see if any of them work.

Only rules from the set Q can be used. The rules from the Q set are tried on all nonterminals in the input string from the left to the right till any of them works. If no such rule can be applied, the derivation is finished unsuccessfully with the result that the input string does not belong to the generated language of grammar G. Detailed look on parsing programmed grammars is written in algorithm 1. We were trying to make the algorithm as simple as possible. For that we have to made abstractions that is hiding some information. One piece of information that disappeared from the algorithm is the fact that the rules choosing is also based on the terminals in input string.

5. Implementation

The goal of this paper is to create a parser that is based on programmed grammars. As an implementation language we have chosen C#. For applications like parsers or translators it is not appropriate to make a graphical user interface. Programmers do not usually see an execution of compilers or translators. For this reason, command line interface fits here the best.

The program needs two different inputs in order to work. Firstly, it needs programmed grammar, whose format is described in section 5.1. Secondly, the program needs a user-defined input string. Finite automaton that works as in the algorithm 1 decides whether the input string belongs to the language generated by the input grammar or not. The result is then written to the console output.

5.1 Grammar format

Every set consists of comma-separated one-letter symbols. On the first line there is a set of terminals, followed by a set of nonterminals on the second line. After the set of nonterminals there are grammar rules. Every rule is written on a single line and begins with a number. At the end of each rule there is a set of the next rules that tells which rules can be used next. Symbol e is used to simplify writing the grammar file, it stands for an empty string. Every grammar file has the following structure:

```
 T: <set of terminals>
 N: <set of nonterminals>
 <number>: <left side>
-> <right side>,
 {<set of next rules>}
```

An example file can be seen at the beginning when executing the program on figure 1.

6. Testing

During application testing we focused on languages that are not context-free. We tested many inputs from these languages:

1. $L_1(G) = \{a^n b^n c^n \mid n \geq 0\}$
2. $L_2(G) = \{ww \mid w \in \{a,b\}^*\}$
3. $L_3(G) = \{wx \mid w \in \{a,b\}^* \ and \ x = reversal(w)\}$
4. $L_4(G) = \{ww \mid w \in \{a,b\}^*\}$
5. $L_5(G) = \{a^i b^j c^k \mid i,j,k \geq 0 \ and \ i < j < k\}$
6. $L_6(G) = \{a^i b^j c^i \mid i,j \geq 0 \ and \ j = i^2\}$
7. $L_7(G) = \{a^i b^j c^j d^i \mid i,j \geq 0 \ and \ j \neq i\}$
8. $L_8(G) = \{waw \mid w \in \{a,b\}^*\}$
9. $L_9(G) = \{0^i 1 0^i 1 0^i 1 0^i \mid i \geq 1\}$
10. $L_{10}(G) = \{wcv \mid w,v \in \{a,b\}^* \ and \ w = vv\}$

Most of the languages are context-sensitive, but $L_3(G)$ is a context-free one. Languages $L_4(G), L_5(G), \ldots, L_{10}(G)$ were taken from [5]. Hundreds of inputs that could cause any problems were tried. Every tested input was evaluated correctly by the application. Those languages will not be listed because that would be too long. As an example, see figure 1 for one nontrivial tested string.

7. News for programming languages

we have already shown that the application can handle two languages that are context-sensitive. For example let's look at the generated language $L(G)$ mentioned in example 3.1. This language can give us the following construction in programming language:

$$x, y, z : int, bool, double : 1, true, 2.3;$$

This is a definition of three variables on a single line. If we admit, then any number of variables can be used, than this is the context-sensitive language $L(G)$ we were talking about. The listed construction is just a brief example of what could be used in new languages.

8. Conclusions

This paper is about creation of parsers that could parse languages that are not context-free. Those parsers are based on programmed grammars.

The results of this work are really promising. Hundreds of inputs were tested for some mostly context-sensitive languages.

The main contribution of this paper is an algorithm that can handle context-free and some context-sensitive grammars for theoretical languages. We do not know how many context-sensitive grammars this

Algorithm 1 Algorithm for parsing with programmed grammars

```
1: procedure AUTOMATON
2:     Initialization of the list of next rules so it contains only starting rule
3:     while Some rule from next rules can be used do
4:         Division of next rules into the rules with and without epsilon rules
5:         Go through the nonterminals in the string from left to right and look for the first applicable rule in non-epsilon rules (skip this step if the number of terminals in generated string equals the number of terminals in input string)
6:         if Not found then
7:             Go through the nonterminals in the string from left to right and look for the first applicable rule in epsilon rules
8:             if Not found then
9:                 Input string rejected.
10:        Use the found rule and update the list of next rules
11:        Compare all of the terminals from the left sides of generated and input string that hasn't been compared yet (the comparison is performed only until the leftmost occurrence of a nonterminal)
12:        if Mismatch then
13:            Input string rejected.
14:    Input string accepted
15:    Print the used rules
```

algorithm can deal with exactly. That would require mathematical verification of the application. Another contribution is the design of an application that was used for testing of user-defined strings.

This work can be taken as a starting point for translators that would be able to provide new grammatical constructions for programming languages. This is what will be done in the future. First, the parser will be extended to be able to handle any terminals and nonterminals (not just one-letter ones) and then a translator that will compile languages with new grammatical constructions will be created.

Acknowledgements

I would like to thank Prof. RNDr. Alexander Meduna, CSc. for giving necessary advices and guidance that greatly helped me finalizing this work.

References

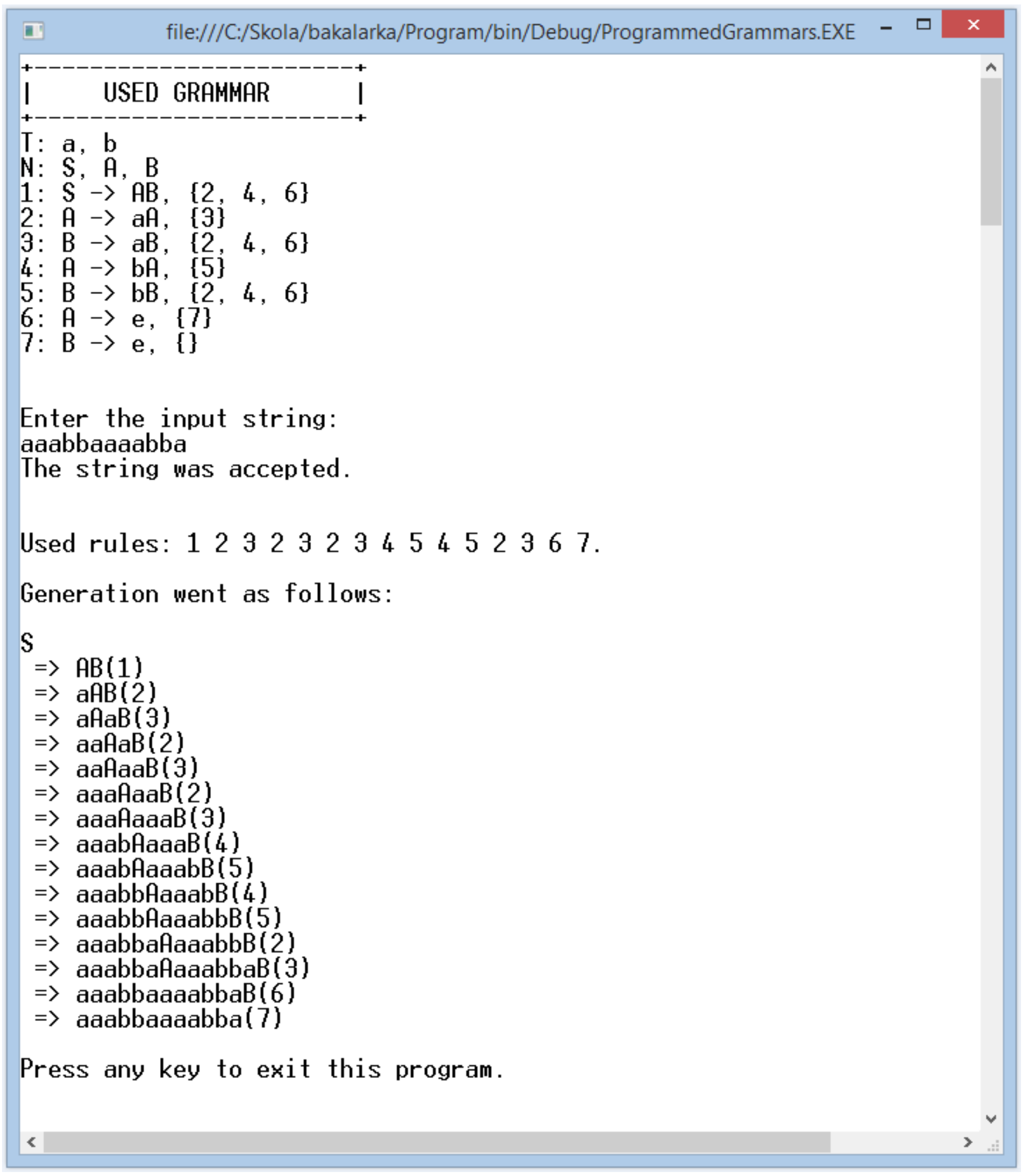
[1] Alexander Meduna. *Automata and Languages: Theory and Applications*. Springer, 2000. ISBN: 1–85233–074–0.

[2] Joachimg Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2nd Edition, 2003. ISBN: 978–0521826464.

[3] Alexander Meduna. *Elements of Compiler Design*. Auerbach Publications, 2008. ISBN: 978–1–4200–6323–3.

[4] Alexander Meduna and Petr Zemek. *Regulated Grammars and Their Transformations*. Brno University of Technology, 2010. ISBN: 978–80–214–4203–0.

[5] Alexander Meduna. *Formal Languages and Computation: models and their application*. CRC Press, 2014. ISBN: 978–1–4665–1345–7.


file:///C:/Skola/bakalarka/Program/bin/Debug/ProgrammedGrammars.EXE

```
+------------------------+
|      USED GRAMMAR      |
+------------------------+
T: a, b
N: S, A, B
1: S -> AB, {2, 4, 6}
2: A -> aA, {3}
3: B -> aB, {2, 4, 6}
4: A -> bA, {5}
5: B -> bB, {2, 4, 6}
6: A -> e, {7}
7: B -> e, {}


Enter the input string:
aaabbaaaabba
The string was accepted.


Used rules: 1 2 3 2 3 2 3 4 5 4 5 2 3 6 7.

Generation went as follows:

S
 => AB(1)
 => aAB(2)
 => aAaB(3)
 => aaAaB(2)
 => aaAaaB(3)
 => aaaAaaB(2)
 => aaaAaaaB(3)
 => aaabAaaaB(4)
 => aaabAaaabB(5)
 => aaabbAaaabB(4)
 => aaabbAaaabbB(5)
 => aaabbaAaaabbB(2)
 => aaabbaAaaabbaB(3)
 => aaabbaaaabbaB(6)
 => aaabbaaaabba(7)

Press any key to exit this program.
```

Figure 1. Application that accepted an input that is not from context-free grammar