Jiří Kepka

The employment of Prolog for a syntax analysis in syntactic pattern recognition applications

Kybernetika, Vol. 28 (1992), No. 1, 62--68

Persistent URL: http://dml.cz/dmlcz/124972

KYBERNETIKA — VOLUME 28 (1992), NUMBER 1, PAGES 62 – 68

THE EMPLOYMENT OF PROLOG FOR A SYNTAX ANALYSIS IN SYNTACTIC PATTERN RECOGNITION APPLICATIONS

JIŘÍ KEPKA

In the paper some possibilities of the employment of Prolog for a syntax analysis of context-free languages are shown with respect to syntactic pattern recognition applications. The new syntax-analyzer based on "top-down" strategy is proposed. All terminal symbols contained on a right side of a production of a context-free grammar are used for the decision whether or not the production shall be applied.

1. INTRODUCTION

A syntax analysis is one of major parts of a syntactic pattern recognition system. The decision whether or not the pattern belongs to the class of patterns described by the given grammar is performed by the "syntax analyzer" or "parser". Provided the pattern is syntactically correct, a complete syntactic description is usually needed, too.

The problem of parsing can be regarded in the following way: Given a sentence x and a (context-free or context-free programmed) grammar G, construct a derivation of x and find a corresponding tree [2]. Two major strategies called "top-down" parsing and "bottom-up" parsing are used to find a solution of this problem. Besides algorithms based on these strategies two efficient algorithms called the Cocke–Younger–Kasami parsing algorithm [2] and Earley's parsing algorithm [7] were developed.

In this paper some possibilities of the employment of Prolog for a syntax analysis of context-free languages are discussed with respect to requirements of syntactic pattern recognition. The new syntax analyzer based on "top-down" strategy is proposed.

2. THE EMPLOYMENT OF PROLOG FOR A SYNTAX ANALYSIS OF CONTEXT-FREE LANGUAGES

It is known that if G is a context-free or context-free programmed grammar the corresponding recognition device is, in general, a nondeterministic, pushdown automaton. Consequently, the parsing process is usually a nondeterministic procedure and backtrackings are often required. Especially for this reason Prolog appears to be very efficient

and advantageous tool for solving problems of syntax analysis of context-free languages because this programming language uses a backtracking mechanism. When a Prolog program is executed, the system tries to find all possible solutions that satisfy the given goal. Once one solution has been found, the backtracking mechanism causes Prolog to reevaluate any assumptions made to see if some new variable values will provide new solutions. It also backtracks if a subgoal fails, hoping to resatisfy a previous subgoal in such a way that the failed subgoal is satisfied with new variable values [4], [5]. Consequently, if a program carrying out a syntax analysis is written in Prolog, we need not worry about how to ensure backtrackings during the execution of this program. Moreover, Prolog is a declarative language and once the Prolog programmer has described what must be computed, the Prolog system itself organizes how that computation is carried out. In Pascal, in contrast, one must tell the computer exactly how to perform its tasks [5]. It is easy to see for every Prolog programmer that "top-down" parsing strategy and Prolog evaluation strategy are the same in principle.

The fact that the parsing process is, in general, a nondeterministic procedure and backtrackings are often required, makes the parsing process potentially inefficient. If there are several choices for each step (i. e. there are several productions with the same left side) then the total number of possible ways of parsing increases exponentially. The speed of parsing depends on avoiding false trials, which cause work to be done that is later rejected. Therefore, it is desired to determine some a priori tests that have to be satisfied before the chosen production is used. The situation such that at each step only one production is possible would be optimal because then backtrackings would not be needed and the parsing method could not be significantly improved [2] .

Several a priori tests were proposed to make the top-down parsing method more deterministic to avoid the exponential increase of the total number of possible ways of parsing. A production can be rejected because it must necessarily lead to more symbols in the terminal sequence than are available, or because it forces the use of a terminal that does not occur. Another a priori test for rejecting a production depends on the first symbol of the phrase that has to be derived from it. If the desired symbol cannot be derived as the first terminal, the production should be rejected. This modified top-down parser select only productions that can yield phrases starting with the same terminal symbol as is the current symbol of the sentence. Before starting syntax analysis the grammar has to be processed to produce a matrix of terminals against nonterminals, indicating which nonterminals can begin with which terminals [2].

Left recursion causes most difficulties in a left-right top-down parser because grammar productions of the form $A \rightarrow A\alpha$ generate infinite loops:

$$A \rightarrow A\alpha \rightarrow A\alpha\alpha \rightarrow \ldots$$

One way to avoid left recursion is to transform the grammar into standard form but then in the syntactic pattern recognition desired structural descriptions are lost [6]. Consequently, other ways must be used. For example, when from every nonterminal can be derived a terminal sequence containing at least one terminal, then if the number

of symbols in the remainder parse is greater than the number of terminals left in the sentence, the parse cannot be correct. This test is useful in practice only if the length of the sentence is short [2].

3. THE TOP–DOWN SYNTAX ANALYZER BASED ON THE NEW A PRIORI TEST

The a priori tests described so far are not efficient enough because the selection from several productions with the same left part only according to the first terminal symbol (modified top-down parser) is very short-sighted.

As follows, it is shown how all terminals contained on the right part of a production can be used for the decision whether or not the production shall be applied. Let w' be the analysed string, $w' = \alpha'\tilde{w}$ and let $\alpha'\beta$ be the sentence form derived by means of productions of a context-free grammar G:

$$S \Rightarrow_G^* \alpha'\beta,$$

where β begins with a nonterminal A, $A \in V_N$, $\beta \in (V_N \cup V_T)^+$, $\alpha' \in V_T^*$. Let exist such a left derivation that $\beta \Rightarrow_G^* \tilde{w}$ holds and let $\beta = A\delta$. A production with the nonterminal A on its left side is generally of the following form:

$$A \to \alpha_1\gamma_1\alpha_2\gamma_2 \ldots \alpha_i\gamma_i, \tag{1}$$

where $\alpha_1 \in V_T^*$, $\alpha_l \in V_T^+$, $1 < l \leq i$, $\gamma_i \in V_N^*$, $\gamma_k \in V_N^+$, $1 \leq k < i$. If $\beta \Rightarrow_G^* \tilde{w}$, then the following derivation must exist:

$$\alpha_1\gamma_1 \ldots \alpha_i\gamma_i\delta \Rightarrow_G^* \tilde{w}.$$

The first primary condition for the selected production to be valid is: $\tilde{w} = \alpha_1 w$. Let $w = w_1 w_2$, then the following two derivations must exist too:

$$\gamma_1\alpha_2\gamma_2\alpha_3 \ldots \alpha_i \Rightarrow_G^* w_1 \cap \gamma_i\delta \Rightarrow_G^* w_2$$

If we use terminal symbols contained on the right side of the production in $\alpha_2, \alpha_3, \ldots \alpha_i$ (we assume they are some in the production), we can determine other conditions. If the selected production is valid the following decomposition of the string w must be found:

$$w = v_1\alpha_2 v_2\alpha_3 \ldots \alpha_i w_2 \tag{2}$$

The proposed a priori test is performed in the following way. The decomposition of the string w is looked for:

$$w = v_1\alpha_2 w'$$

$$w' = v_2\alpha_3 w''$$

Then w'' is processed in the same manner as w and w' until the total decomposition of the form (2) is found. If the decomposition is not found the production should be rejected. As

follows, for the application of the selected production to be valid the following derivations must be found:

$$\gamma_1 \Rightarrow_G^* v_1 \cap \gamma_2 \Rightarrow_G^* v_2 ... \cap \gamma_i\delta \Rightarrow_G^* w_2 \tag{3}$$

Evidently, the a priori tests already described in this paper can be used too:

1. If the derivations (3) exist the leftmost nonterminal symbol of $\gamma_k \in V_N^+$ $(\gamma_i\delta)$, $k < i$ must yield the leftmost terminal symbol of the string $v_k \in V_T^+$ (w_2).

2. When from every nonterminal can be derived a terminal sequence containing at least one terminal symbol, then if the derivations (3) exist the number of symbols in $\gamma_k \in V_N^+$ $(\gamma_i\delta)$, $k < i$, must not be greater than the number of terminals in $v_k \in V_T^+$ (w_2).

These a priori tests can be employed when a decomposition (2) is being looked for. First, the condition whether the leftmost terminal symbol of the string v_1 can be derived from the leftmost nonterminal symbol of γ_1 is checked. Second, the decomposition $w = v_1\alpha_2 w'$ is looked for. Third, the condition that the number of symbols in γ_1 must not be greater than the number of symbols in v_1 is checked. If the third condition is not satisfied backtracking is necessary to try to find an alternative decomposition of the w. When all conditions are satisfied w' is processed in the same manner as w. If whether first or second condition is not satisfied (for w') then backtracking is also necessary to find an alternative decomposition, etc. After the total decomposition of the form (2) is found the two last conditions that the leftmost terminal symbol of the string w_2 can be derived from the leftmost nonterminal symbol of $\gamma_i\delta$ and that the number of symbols in $\gamma_i\delta$ must not be greater than the number of symbols in w_2 are checked. If the decomposition of the form (2) is not found the production is rejected. Of course, if there are productions of the form $P \to e$, where e is the empty symbol, presented in the grammar, the a priori tests 1, 2 cannot be simply used without changes.

Example 1. Consider the grammar $G = (V_N, V_T, P, S)$, where S is the starting nonterminal, $V_N = \{S, A, B\}$, $V_T = \{a, b, c, d\}$, and P:

$$S \to AB \quad B \to d \quad A \to c \quad B \to Bb \quad A \to Aa$$

In general, $L(G) = \{ca^n db^m \mid n, m = 0, 1, 2, ...\}$. Let $w = caadbb$. The production $S \to AB$ is accepted because the first terminal symbol of w can be derived from the nonterminal A and the number of symbols of AB is not greater than the number of symbols of w. Then the production $A \to c$ is tested. The condition that the nonterminal B can yield the terminal symbol a is not satisfied and consequently, the production is rejected and the production $A \to Aa$ is tried. The first condition is satisfied because the first terminal symbol c of the string w can be derived from the nonterminal A. The decomposition $w = v_1\alpha_2 w_2 = c|a|adbb$ is found but the derivation $B \Rightarrow_G^* adbb$ does not exist because B cannot yield the terminal symbol a as the first one. Consequently,

backtracking is necessary to find another decomposition of w, $w = ca|a|dbb$. This decomposition satisfies all conditions. As follows, the derivations $A \Rightarrow ca$ and then $B \Rightarrow dbb$ are looked for.

If any of the subgoals (3) fails, backtracking is needed to find another decomposition of the form (2) that satisfies all conditions. In the case that it does not exist an alternative production is chosen and tested. Several decomposition of the form (2) satisfying all conditions can be generally found. With respect to the fact that the backtracking mechanism is one of fundamental features of Prolog the realization of this a priori test is quite easy, in contrast to other programming languages.

The realization of this a priori test using all terminal symbols on right sides of productions in Prolog is based on the known predicate *append*. For example, let's append the lists ['a','a','b'] and ['a','c'] to form the list ['a','a','b','a','c']. The predicate *append*(*List*1, *List*2, *List*3) with three arguments combines *List*1 and *List*2 to form *List*3. If *List*1 is empty, the result of appending *List*1 and *List*2 will be the same as *List*2:

$$append([\,],\ List2,\ List2).$$

Otherwise, *List*1 and *List*2 can be combined to form *List*3 by making the head of *List*1 the head of *List*3. The rest of *List*3 is obtained by appending the rest of *List*1 and the List2:

$$append([X|L1],\ List2,\ [X|L3]) : -append(L1,\ List2,\ L3).$$

This relation between three lists also holds if *List*1 (*List*2) and *List*3 are known but *List*2 (*List*1) is not known and if only *List*3 is known.

Example 2. To find which two lists could be appended to form a known list, we could use a goal of the form

$$append(List1,\ List2,\ ['c','a','a']).$$

for which Prolog will find the solutions

List1 = [ ]	List2 = ['c','a','a'],
List1 = ['c']	List2 = ['a','a'],
List1 = ['c','a']	List2 = ['a'],
List1 = ['c','a','a']	List2 = [ ].

Example 3. Let $A \rightarrow AabBCc$ be the production which causes left recursion and let $\alpha'\beta = \alpha' A\delta$ be the derived sentence form ($\alpha' \in V_T^*$) and W the part of the analyzed string W' ($W' = \alpha' W$). If the nonterminal A can yield the leftmost terminal of the string W then the decomposition of the type (2) is looked for. The realization of this proposed test (decomposition) in Prolog is based on the following *test* predicate

$$test(W,\ [V1, V2, W2],\ Number) : -$$
$$append(V1,\ ['a'|['b'|L1]],\ W),$$
$$append(V2,\ ['c'|W2],\ L1).$$

where $Number$ is the number of the production $A \rightarrow AabBCc$. If this test is succesfully evaluated, then the simpler derivations $A \Rightarrow_G^* V1$, $BC \Rightarrow_G^* V2$ and $\delta \Rightarrow_G^* W2$ must be found if the production is valid.

To employ the a priori test that "the leftmost nonterminal symbol must yield the leftmost terminal symbol" consider that Prolog facts

$$list\_t(Nonterminal,\ List\_of\_terminals)$$

indicating which nonterminals can begin with which terminals, are created. Then the predicate *test* can be modified into the following form:

$$\begin{aligned} &test([T1|W],\ [V1, V2, W2],\ Number) :- list\_t('A',\ ListA),\\ &\quad member(T1,\ ListA),\ append(V1,\ ['a'|['b'|[T2|L1]]]),\ [T1|W]),\\ &\quad list\_t('B',\ ListB),\ member(T2,\ ListB),\\ &\quad append(V2,\ ['c'|W2],\ [T2|L1]). \end{aligned}$$

where the known *member* predicate is defined by [5]:

$$\begin{aligned} &member(X,\ [X|\_]) :- !.\\ &member(X,\ [\_|Y]) :- member(X,\ Y). \end{aligned}$$

The proposed a priori test using all terminal symbols reduces essentially the total number of possible ways of parsing. Moreover, the difficulties caused by left recursion are removed if every right side of the production that causes left recursion contains at least one terminal symbol. If the given grammar contains productions that have not any terminal symbol on their right sides and if these productions cause left recursion there are following possibilities how these problems can be solved. One of them already described is to check whether the number of symbols in the remainder parse is not greater than the number of terminals left in the sentence. This a priori test can be used only when from every nonterminal at least one terminal symbol can be derived. Thus, the grammar must not contain the productions of the form $A \rightarrow e$. Another way how to remove left recursion is to modify the given grammar according to [3] without losing the desired structural description of the given string. More details about the proposed test and the description of the program (analyzer) can be found elsewhere, see [8].

4. CONCLUSIONS

The syntax analysis of context-free languages is a nondeterministic procedure and backtrackings are often required. For this reason Prolog appears to be very convenient tool for performing the parsing process. Moreover, both productions of a context-free grammar and top-down parsing strategy can be easily described by means of simple Prolog facts and rules.

The syntax analyzer based on the new a priori test using all terminal symbols on the right side of a production of the given grammar was proposed. Backtrackings are in general needed. The proposed a priori test essentially reduces the total number of possible ways of parsing and the difficulties caused by left recursion are removed if every right side of the production that causes left recursion contains at least one terminal symbol. The syntax analyzer based on this a priori test can be used in syntactic pattern recognition systems, where the structural description according to the given grammar is desired.

(Received September 25, 1989.)

REFERENCES

[1] M. Chytil: Automata and Grammars (in Czech). SNTL, Prague 1984.
[2] K. S. Fu: Syntactic Pattern Recognition and Applications. Prentice-Hall, Englewood Cliffs, N. J. 1982.
[3] R. Kurki-Suonio: On top-to-bottom recognition and left recursion. Comm. ACM *9* (1966), 7, 527 – 528.
[4] W. F. Clocksin and C. S. Mellish: Programming in Prolog. Second edition. Springer-Verlag, Berlin – Heidelberg – New York 1984.
[5] Turbo Prolog – the natural language of artificial intelligence. Owner's Handbook, Borland International, Inc. 1988.
[6] T. V. Griffiths and S. R. Petrick: On the relative efficiencies of context-free grammar recognizers. Comm. ACM *8* (1965), 5, 289 – 300.
[7] J. Earley: An efficient context-free parsing algorithm. Comm. ACM *13* (1970), 2, 94 – 102.
[8] J. Kepka: The Top-Down Parsing of Context-Free Languages and Its Realization in Turbo Prolog (in Czech). Research Report No. 1681, ÚTIA ČSAV, Prague 1990.

Ing. Jiří Kepka, Ústav teorie informace a automatizace ČSAV (Institute of Information Theory and Automation – Czechoslovak Academy of Sciences), Pod vodárenskou věží 4, 182 08 Praha 8. Czechoslovakia.