Efficient String Matching and Easy Bottom-Up Parsing
(A DRAFT, submitted to CIAA 2007)

Ján Šturc

FMPhI – CU Bratislava, jan.sturc@dcs.fmph.uniba.sk

Abstract. The paper consists of the two parts. In the first one we compare three string matching algorithms: Dömölki algorithm, also known as SHIFT OR algorithm, $O(n \times m)$, Table driven $O((n+m) \times \lg m)$ and Aho Corasick $O(m+n)$. Table driven algorithm pass trough the same states as Dömölki one but have compact states encoding. The table driven algorithm can be also considered as Aho Corasick algorithm with eliminated ϵ-transitions. We advocate that the table driven algorithm is the best solution for matching multiple patterns of reasonable size.
The second part of the paper deal with bottom-up syntax analysis. We have shown that the backward deterministic syntax analysis can be implemented via extension of a string matching automaton by a stack; or two stacks, if we want to go beyond context free grammars. The implementation of a parser this type is as easy as writing a recursive descent parser; we need to supply only the transition table, which can be easily derived from the grammar.
Finally, we discuss some compiler engineering details.

1 Motivations

One of motivations for this paper are doubts on some commonly accepted theoretical results. We want to clarify efficiency of pattern matching algorithms. The Aho Corasick algorithm is considered the best one, in the case the full scan of the input is necessary. But it does more transitions than it is needed. We have felt that the table driven algorithm may be better in the majority of practical applications.

Another such commonly accepted true is that an implementation of top down syntax analysis is easy, and the same for bottom-up analysis is difficult. Long time ago we have implemented [DS73] a translator writing system (TWS) based on Dömölki algorithm. To implement TWS was much easier than to implement a standard SLR parser.

In many practical application there is need to find specified patterns in a given environment. For instance bibliographic search, data mining, web mining or network intrusion analysis. Many of these cases prevent sublinear search for some; many patterns with many short one among them, data are provided as a stream which have to been proceed in real time or patterns are too complex, they can not be described by a regular language. We concentrate our attention to linguistic patterns, which are described by a grammar (context free or context sensitive). The top down methods are not well suitable for partial analysis (pattern search) the bottom-up methods are considered to be difficult. A method presented in the paper join advantages of pattern matching with syntax analysis.

2 Basic concepts

We use the basic concepts from automata theory, language theory and compilers as they appears in each compiler textbook. In particular, we use "Red Dragon" [ASU86].

3 Pattern Matching

3.1 Dömölki algorithm

In the year 1964 Bálint Dömölki [D64] presented a string matching algorithm based on only few bitwise logical operations. The idea of that algorithm is rather simple; to represent the state of the matching by a bit vector. The individual bits in the state vector represent the positions in the pattern. To each alphabet symbol is assigned a binary vector, which says where in the pattern that symbol appears. The pattern matching is now very simple: check coincidence with current input symbol and shift the state vector one bit to the right, to be prepared for checking coincidence of the next symbol. For the implementation we need two vectors which record the beginnings and the ends of the patterns. The core of this algorithm is very simple:

```
q := u; c := readinput; /* initialize */
while c ≠ EOF do
    begin q := q ∧ M[c];
          x := q ∧ v;
          if x ≠ 0 then
                report_patterns(x);
          q := shiftright(q) ∨ u;
          c := readinput
    end;
```

Here $\mathbf{M}[1:k][1:m]$ is a binary matrix, which rows correspond to symbols of the alphabet and columns correspond to the positions in the patterns. The patterns are ordered into a sequence and $\mathbf{M}[i,j] = 1$, if and only if the i-th symbol appears in the j-th position of this sequence. m is the total length of pattern sequence and k is the number of alphabet symbols. The binary vectors **u** and **u** indicate beginnings and ends of the patterns.

In the presented algorithm is unspecified function *report_patterns*. The patterns here are recognized according to their final position. So to each *one* in vector x corresponds a pattern or a set of patterns. This looks to be an expensive operation; in a cycle $shiftleft$ and test whether the current bit is *one*. The floating point arithmetic usually provides the instruction ($shiftnormalize$) which finds first non zero bit. Using this instruction, the steps of the test cycle are reduced to several *jumps*.

Dual version of this algorithm has been rediscovered and spread by Baeza-Yates and Gonet [BG92] at the beginning of nineties under the name SHIFT OR string matching algorithm. For a reduced instruction set computer one of these implementations may be considerably better, but for the implementation on a complex instruction set computer this modification brings nothing.

3.2 Modifications, improvements or spoils

At the first look the main source of inefficiency of the presented algorithm is rather poor utilization of the state vector q. From the 2^m possible states only a small part is actually used. It is not difficult to see that the actually passed states correspond to the trie [K73] of the patterns. That means that only m states are actually used. To implement the algorithm, we replace the matrix $\mathbf{M}$ by a state transition table $\mathbf{T}[1:m][1:k]$ of the transposed dimensions. The table $\mathbf{T}$ defines the transitions among states. $\mathbf{T}[q,c] = q\prime$ means transition from the state q to the state $q\prime$ on the input symbol c. Some transitions are defined directly by the trie

of the patterns. The other transitions are defined as follows: $\mathbf{T}[q, c] = q\prime$ if and only if the longest prefix corresponding to $q\prime$ in trie is the possibly longest suffix corresponding to state q followed by symbol c.

This is Aho Corasick automaton [AC75] with eliminated $\epsilon - transitions$ in the case of unsuccessful matching. The core of the algorithm is reduced to:

$q := u$; $c := readinput$; /* initialize */
while $c \neq EOF$ **do**
  begin if q is $final$ **then**
    $report\_patterns(q)$;
  $q := \mathbf{T}[q, c]$;
  $c := readinput$
  end;

We need a bit of additional information in the table $\mathbf{T}$, a bit indicating for each state whether it is final, and if so, a pointer to the patterns which have to be reported. The only computation is the index calculation: $base + q \times m + c$.

Fusions of the overlapping patterns: In the trie if a pattern is a prefix of another pattern, a shorter pattern is absorbed in a longer one. This can be devised in original Dömölki matrix, too. We have one pattern with one start position in the vector u and two final positions in the vector v. This merging is safe and causes no problems.

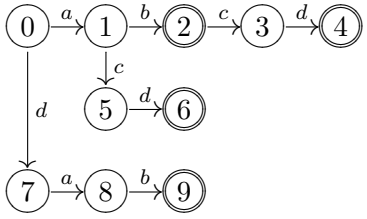
Other merging of the states are only possible, if in the final state of the longer pattern it can be easily determined whether the pattern has been recognized. In the case of many short patterns which appear in the analyzed string sparsely, it may be advantageous to merge as many states as possible and if a final state is reached, start the backward matching with a table for possible reversed strings.

Also cases when the last symbol of a pattern is the first symbol of another pattern can be treated. Actually, in this case the first pattern is shortened by cutting the last symbol and matching reported, only if the *lookahead* input symbol is the last symbol of that pattern.

3.3 Short example

Consider the set of patterns $\{ab, abcd, acd, dab\}$ in the alphabet $\Sigma = \{a, b, c, d\}$. We built the trie, the Dömölki's matrix $\mathbf{M}$ and the transition table $\mathbf{T}$ for modified algorithm.

Trie:



$$\mathbf{M} = \begin{array}{c|cccccccccc} & a & b & c & d & a & c & d & a & b \\ a & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ b & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ c & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ d & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{array}$$

$$\mathbf{u} = 1\,0\,0\,0\,1\,0\,1\,0\,0$$
$$\mathbf{v} = 0\,1\,0\,1\,0\,0\,1\,0\,1$$

$$\mathbf{T} = \begin{array}{c|cccccc|l} & a & b & c & d & F & pattern \\ 0 & 1 & 0 & 0 & 7 & 0 & \\ 1 & 1 & 2 & 5 & 7 & 0 & \\ 2 & 1 & 0 & 3 & 7 & 1 & ab \\ 3 & 1 & 0 & 0 & 4 & 0 & \\ 4 & 8 & 0 & 0 & 7 & 1 & abcd \\ 5 & 1 & 0 & 0 & 6 & 0 & \\ 6 & 8 & 0 & 0 & 7 & 1 & acd \\ 7 & 8 & 0 & 0 & 7 & 0 & \\ 8 & 1 & 9 & 5 & 7 & 0 & \\ 9 & 1 & 0 & 3 & 7 & 1 & ab, dab \end{array}$$

3.4 About complexity

Complexity of these algorithms consist of two parts; complexity of preprocessing (building of the matrix or the transition table) and run of the algorithm on an input of the length n. We assume that there are μ patterns and sum of their lengths is m, and the alphabet cardinality (number of the symbols) is k.

We shall compare three algorithms; Dömölki (SHIFT OR)(D), Table driven (T), Aho Corasick (AC) on four computation models; asymptotic, RAM (Random access machine) with the uniform instruction price (RAM-u), RAM with logarithmic instruction price (RAM-log) and a real complex instruction set computer with the word length w (Comp-w).

Preprocessing Both Dömölki's matrix **M** and table **T** have size $m \times k$ and need additional information which is approximately of the $2m$ size (**u**, **v**, rsp. information about final states and recognized patterns). The AC algorithm may be implemented as an automaton with m states and $2m$ transitions (for each state one regular and one fail). Matrix **M** can be built in the time $O(k \times m)$. In the trie we need in each step to find a position where next pattern will be connected. This operation appears μ times and requires number of step proportional to average length m/μ of the patterns. So, this is $O(m)$. For the table driven algorithm we need to record all transition, they are $m \times k$. The table entry is a destination state. Because there are m states the size of a table entry is $\lg m$ (binary logarithm of m).

Running Both version of Dömölki's algorithm are implemented using main loop, which on the size n input runs n times. Loop of both versions consist of few operations. The difference is the first algorithm (D) except index calculations, manipulates with the vectors of the size m. The total cost is $O(m \times n)$, we neglect simpler index calculation. In the tabular version (T), the only needed calculation is $\mathbf{T}[q, c]$, which is translated to $base + q \times k + c$. The maximum is $(m \times k)$ and this need $\lceil \lg(m \times k) \rceil$ bits. So, the complexity is $O(n \times \lg(m \times k))$ may be the result must be multiplied by a factor $\lg \lg(m \times k)$ because complexity of the multiplication. We ignore the operand size for RAM with the uniform instruction size and divide them by the word length w for a real computer.

Any attempt to implement AC automaton via main loop spoils it asymptotic complexity at least by the factor $\lg m$, because of addressing the states. The only way to implement this automaton correctly is a "straight" program, where states are labels. In each state the automaton reads, tests the read symbol on EOF (termination), direct transition, or fail and jump to next state. Moreover the relative jump distance must be bounded. The last can not be ensured without expensive optimization in the preprocessing phase. For this reason we multiply the cost by $\lg m$ factor, for RAM with logarithmic price and real computer.

The results are summarized in the following table:

		Asymptotic	RAM-u	RAM-log	Comp-w
D	pre	$O(m \times k)$	$O(m \times k)$	$O(m \times k)$	$O(m \times k)$
	run	$O(n \times m)$	$O(n)$	$O(n \times m)$	$O(n \times m/w)$
T	pre	$O(m \times k \times \lg m)$	$O(m \times k)$	$O(m \times k \times \lg(km))$	$O(m \times k \times (\lg(km))/w)$
	run	$O(n \times \lg m)$	$O(n)$	$O(n \times \lg(km))$	$O(n \times (\lg(km))/w)$
AC	pre	$O(m)$	$O(m)$	$O(m \times \lg m)$	$O(m \times (\lg m)/w)$
	run	$O(n)$	$O(n)$	$O(n \times \lg m)$	$O(n \times \lg m)$

For reasonable pattern sizes and alphabet sizes the $lg(km) < w$ and the real runtime complexity is the complexity on RAM with uniform instruction price. Moreover in the case of Aho Corasick algorithm the real run time may be increased because of ϵ-transitions.

Enhancements For original Dömölki and table driven algorithms the same transitions; on several, few or all symbols, cause no problems. It is enough, simply to put more ones in one column of the matrix **M** or the same destinations in one row of the table **T**. In AC algorithm in a state must be implemented more complex test or many tests on the possible scanned symbol. These modifications degrade the efficiency of the AC-algorithm. For practical use on classic von Neumann architecture computer is the table driven algorithm the best choice. In the table driven algorithm there is no problem to add an arc to the trie which close a cycle. So we have iteration $(*)$, sets of the symbols $(|)$ and naturally, the concatenation $(||)$ corresponds to normal trie transitions.

4 Syntax analysis

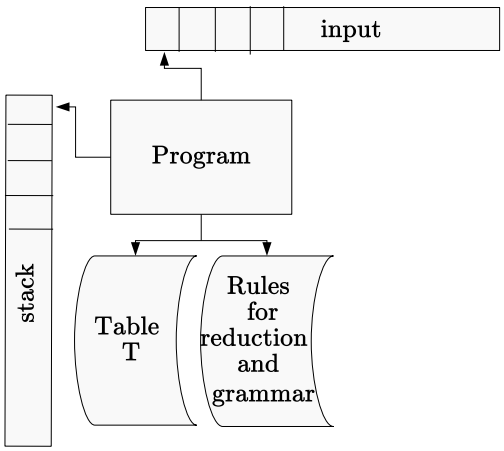
4.1 Main idea

The main idea of the syntax analysis algorithm can be trace back to original Dömölki's paper [D64]. In sixties the bottom-up parsing methods used backtracking. Parsing using pattern matching algorithm is very simple. It is a text book example of shift reduce scheme (see e.g. [AU72] or some of the "Dragons"). The patterns are right hand sides of the grammar rules. To the pattern matching program is added a stack. During the scan (searching a pattern) the states are pushed into the stack. If a pattern is found the program decides whether to reduce, if so: then the number of states equal to the length of the recognized pattern is popped from the stack, the current state is set to the top of the stack " and "the left hand side symbol of the reduced rule is read. The scan continues from the input. Trie transitions correspond to passing trough $LR(0)$ items. The transitions leading to the state 0 correspond to: the accept state if both the stack and the input are exhausted, otherwise it is an error.

4.2 Implementation

We implement the parsing automaton as a LR-parser [ASU86]. The parsing algorithm is a straightforward modification of the table driven pattern matching algorithm to a shift-reduce automaton, described at figure 1. The reductions can appear only in the final states.

We need to explain what is a $ReduceSet(A)$. By the definition this a set, of the terminal symbols. This set is used to decide whether to do a reduction. The easiest is to set $ReduceSet(A) = Follow(A)$. For a nonterminal A, the set $Follow(A)$ is a set of the terminals a, that can appear immediately to the right of A in some sentential form, i.e. $Follow(A) = \{a \in T : S \stackrel{*}{\Rightarrow} \alpha A a \beta\}$. The $ReduceSet(A)$ can be arbitrary subset of the $Follow(A)$. How to compute the set $Follow(A)$ or a bit better $ReduceSet$ based on LALR parsing can be find in any compiler textbook e.g. [ASU86]. There is also another way to solve shift-reduce conflict, simply to prefer longer rules or prefer rules according to the order, they are ordered in the grammar.



$q := 0$; $push(stack, q)$; /* initialize */
repeat
  if q is $final$ **then**
    begin if for some rule $A \to \omega$, which right hand side has been
      matched, and the next input symbol is in the $ReduceSet(A)$
      then begin for $i := 1$ **to** $|\omega|$ **do** $pop(stack)$;
        $c := A$
      end
    end
  else $c := readinput$;
  $q := \mathbf{T}[q, c]$;
  $push(stack, q)$;
until $q = 0$;
if $empty(stack) \wedge empty(input)$ **then** $accept$ **else** $error$;

Fig. 1. Parser

The proposed automaton is not able to treat ϵ-rules (rules of the form $A \to \epsilon$) because they have no right hand side. Note, that if such rule is applicable, it must be applied in the starting state or in a final state just after another reduction. States, where the ϵ-reduction is applicable have to be precomputed and this reduction fused with the previous reduction. In the next section we avoid problem with ϵ-rules by adding some context to them.

Proposition 1. *A word generated by a grammar G is successfully accepted by the proposed parser, if and only if it is accepted by SLR(1) parser. In this case both parser pass trough the similar states.*

Proof. Kernel items of the LR(0) states correspond to the states of the presented algorithm. Because LR(0) states are fully determined by the kernel items the same states are passed. (There is a homomorphism from trie states onto LR(0) states.) Conversely, it is not fully

true. In case the grammar comprises useless nonterminals and useless rules i.e. nonterminals and rules not reachable from starting nonterminals, states corresponding to this rules are not generated by LR(0) parser. In the trie they are, but if the algorithm accepts input these states could be never reached.

Errors are recognized in generally latter than by LR-type syntax analysis; which recognize an error as soon as possible, at the stage when the scanned text become not to be prefix of any word of the language generated by the given grammar.

4.3 Parsing with context sensitive rules

In the next we sacrifice a bit of efficiency in trade of comprehensiveness and generality, and shall do syntax analysis in the textbook style. That means, we shall store in the stack not only the states but also scanned symbols. Moreover we shall assume that the input is a stack, too. The parser is a double stack automaton driven by the transition table. The rules are of the form $\alpha A\beta \leftarrow \alpha\omega\beta$, where α, β and ω are in $(N \cup T)^*$ (context sensitive rules). One need not to be afraid of the input stack; in case of using to parse the grammars raised from $SLR(k)$ or $LALR(k)$ grammars it will be shallow (just k symbols).

The parsing program consists of two kinds of steps:

shift the current state and top of the input is pushed onto stack and input is popped and

reduce the right context of recognized rule are "moved" from the stack back to the input and the states in between are omitted. As the last the left hand side nonterminal is pushed on the input.

```
q := 0; /* initialize */
repeat  if q is final then
            begin for i := 1 to |β| do
                      begin pop(stack);
                            push(input, top(stack)); pop(stack)
                      end;
                  for i := 1 to 2 * |ω| do  pop(stack);
                      push(input, A); q := top(stack)
            end
        else begin q := T[q, top(input)]; push(stack, q);
                   push(stack, top(input); pop(input);
             end
until  q = 0;
if empty(stack) ∧ empty(input) then  accept else  error;
```

Fig. 2. A backward deterministic context sensitive parser

Proposition 2. *Backward deterministic context sensitive parser (BDA) parses $SLR(1)$ and $LALR(1)$ languages as efficiently as $SLR(1)$ or $LALR(1)$ parser.*

Proof. Let G be a $SLR(1)$ ($LALR(1)$) grammar. Let a set R be the ReduceSet for a rule $r : A \leftarrow \omega \in G$. (In case of SLR, $R = Follow(A)$.) We replace each such rule of G by context sensitive rule $r\prime : AR \leftarrow \omega R$ (augmented grammar). Really, it is a set of rules. The core of algorithm (string matching) is able do treat such class of rules as one rule. The instantaneous descriptions of the program have general scheme: $stack \downarrow input$. They look like $q_0 S_0 \dots q_i S_i \downarrow i_j \dots i_n$. They are the same as the instantaneous descriptions of a LR parser. The only difference is that at the beginning of the input may appear nonterminals. Compare now parsing the original grammar by xLR automaton and parsing the augmented grammar by BDA. Let both automata parse according to the rule $A \leftarrow \omega$. During shift phase they pass trough the same states. In the case of reduction the xLR automaton look ahead for the next symbol according to the lookahead it decides to reduce and pass to the state after reduction on A. The BDA must shift the look ahead symbol onto stack, because the grammar is augmented. In the next state it does reduction unconditionally and both lookahead symbol and nonterminal A are pushed back to the input. Now A is on the top of input. The state is a shift state because $Follow$ sets contain only terminals. So, A is shifted to the stack and BDA passes to the stated after reading A. Now it is in the same state as xLR after reduction by the rule $A \leftarrow \omega$. We can conclude, both automata pass the same states with the exception of local difference in the reduction, for which BDA needs three transitions.

4.4 Backward deterministic rewriting

From the algorithm point of view there is no need to restrict the algorithm on context sensitive rules. Using arbitrary rules is possible. The difference is in reduction. Let the rule is of the form $\alpha\beta \rightarrow \alpha\omega$, where α, β and ω are in $(N \cup T)^*$. The α is a common prefix of left and right side of the rule. In particular α may be the empty string. In this case ω is popped from the stack and β is pushed to the input.

```
q := 0; /* initialize */
repeat  if q is final then
            begin for i := 1 to |β| do push(input, β_i)
                  for i := 1 to 2 * |ω| do  pop(stack);
                  q := top(stack)
            end
        else begin q := T[q, top(input)]; push(stack, q);
                   push(stack, top(input); pop(input);
             end
until  q = 0;
if empty(stack) ∧ empty(input) then  accept else  error;
```

Fig. 3. A backward deterministic rewriting

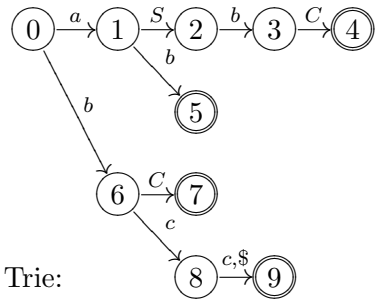
4.5 Example

Consider the language $L = \{a^n b^n c^n : \ n > 0\}$. This language can be generated by the grammar with the following rules:

Grammar:

$$S \ \rightarrow aSbC$$
$$aSb \rightarrow ab$$
$$Cb \rightarrow bC$$
$$bCc \rightarrow bcc$$
$$bC\$ \rightarrow bc\$$$

Trie:



	a	b	c	\$	S	C	F	*pattern*
0	1		6				0	
1	1	5			2		0	
2	1	3					0	
3	1	6				4	0	
4							1	$aSbC$
5							1	ab
6	1	6				7	0	
7							1	bC
8	1	6	9	9			0	
9							1	$bcc, bc\$$

The symbol \$ is used as a sentinel. For the reason of readability the error transitions are kept blank. There are no transitions from the final states because there are no shift reduce conflicts.

5 Error diagnostics

We can a bit modify transition table (or transition graph). The trie transitions are untouchable. The fail transitions are the key for improvements. We need no transitions from a final states because the reduction take place and the state after each reduction is defined. The fail transitions correspond either to starting embedded production (closure) or to an error. There is simple criterion to decide it. Let a backward transition transition leads from a position $\alpha \cdot \beta$ to the position $\gamma \cdot \delta$, where α, β, γ and δ are in $(N \cup T)^*$; then this transition is surely beginning of the error parsing if γ is not a prefix of a string ω such that $\beta \stackrel{*}{\Rightarrow} \omega$.

In spite of the last is undecidable for general rewriting rules in the case of context free and context sensitive rules it is quite easy. In particular if the destination of the suspicious backward transition is a state that immediately follows state zero this test is reduced to the test; whether a terminal symbol $c \in First(\beta)$ or whether there is a series of rules beginning with a nonterminal symbol which leads from the first symbol of β to γ which both have to be nonterminals.

We can not explicitly say which kind of error diagnostics is better the original one (pattern matching style) or the improved one (LR style). One can find examples in favor both. The compiler designer may exploit the advantages of the both one; to call the first time an error routine in LR style and then to decide, whether proceed in an error recovery mode or to return parsing into original destination state.

6 Compiler engineering

In this section we discuss some practical aspects of the compiler engineering. In spite of the fact, that grammars deal with individual symbols and rules, often happen that in some place

may appear several symbols or several rules produce similar or the same meaning. It is useful to treat such set of symbols as one symbol and such set of rules as one rule.

To define sets of symbols, we propose term like notation. Arguments each set creating symbol are its subsets or members.

Example

ALL(Terminal((LetterOrDigit(Digit(Binary(0,1),2,3,4,5,6,7,8,9),
Letter(Capital(A,B,C, . . . , Z),
Small(a,b,c, . . . , z))),
Punctuation(. , !, ?,'comma', . . .)
Blank(␣, 'tab', 'newline'), . . .))
Nonterminal(identifier, . . .))

The rule for an identifier is $identifier\ Blank \rightarrow Letter\ LetterOrDigit^*\ Blank$. Finally, such a notation is commonly accepted in the manuals of practitioners.

Joining rules into class rules require a bit care not to produce overlapping classes and do not join rules with different semantics. One could also use difference of the classes. Ad absurdum each symbol can have its own class.

Moreover, we propose to create internal encoding of the symbols by depth first numbering of the leaves of this term tree. This is recommended in the cases: we want exploit the position which hardware norms assign to the symbols never used by designed compiler, or we want to change collating sequence according to local habits.

Semantics: In the bottom-up parsing the semantic routines are called in the time of reductions. The compiler designer must decide whether he uses the rule like

$$identifier\ Blank \rightarrow Letter\ LetterOrDigit^*\ Blank$$

and then reconstructs the recognized identifier from symbols in the stack (semantic routine must be called before actual reduction), or he uses three rules:

$$\begin{aligned} identifier\ Blank &\rightarrow id\ Blank \\ id &\rightarrow Letter \\ id &\rightarrow id\ LetterOrDigit \end{aligned}$$

and call semantic routine in the time of each reduction, like during top down parsing. Marking nonterminals with ϵ rules can help.

Scanner: Despite the scanner is usually separated part of the compiler, if one decides to recognize reserved words via table look-up, the lexical analysis may be integrated to the syntax analysis. This way designed compiler becomes very compact.

7 Conclusions

Concerning the string matching. In the case that full scan is necessary, the table driven program is superior. The original Dömölki and SHIFT OR algorithm can not achieve comparable efficiency. Aho Corasick algorithm is asymptotically better. For patterns of reasonable size AC can not take advantage of it asymptotic performance and it is less efficient than Table driven algorithm because of ϵ-transitions. Both Dömölki and Aho Corasick are important mainly for hardware and parallel implementations.

The proposed implementation of syntax analysis trough string matching make bottom-up methods as easy as top down methods and allows broader classes of languages to be parsed.

References

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM, 18(6)* 1975, pp. 333 - 340.

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman: Compilers Principles, Techniques and Tools. Addison Wesley, 1986.

[AU72] A. V. Aho and J. D. Ullman: The theory of parsing, translation and compiling, Prentice-Hall 1972.

[BG92] R. A. Baeza-Yates and G. H. Gonnet: A new approach to text searching. *Communications of the ACM, 35(10)* 1992, pp. 74 - 82.

[DS73] J. Duplinský and J. Šturc: TWS translator writing system. Technical report, CRC UNDP Bratislava, 1973

[D64] B. Dömölki: An algorithm for syntactic analysis. *Computational Linguistics, 8* 1964, pp. 29 - 46.

[K73] D. E. Knuth: The art of computer programming. Vol. III; Sorting and searching, Adison Wesley 1973.