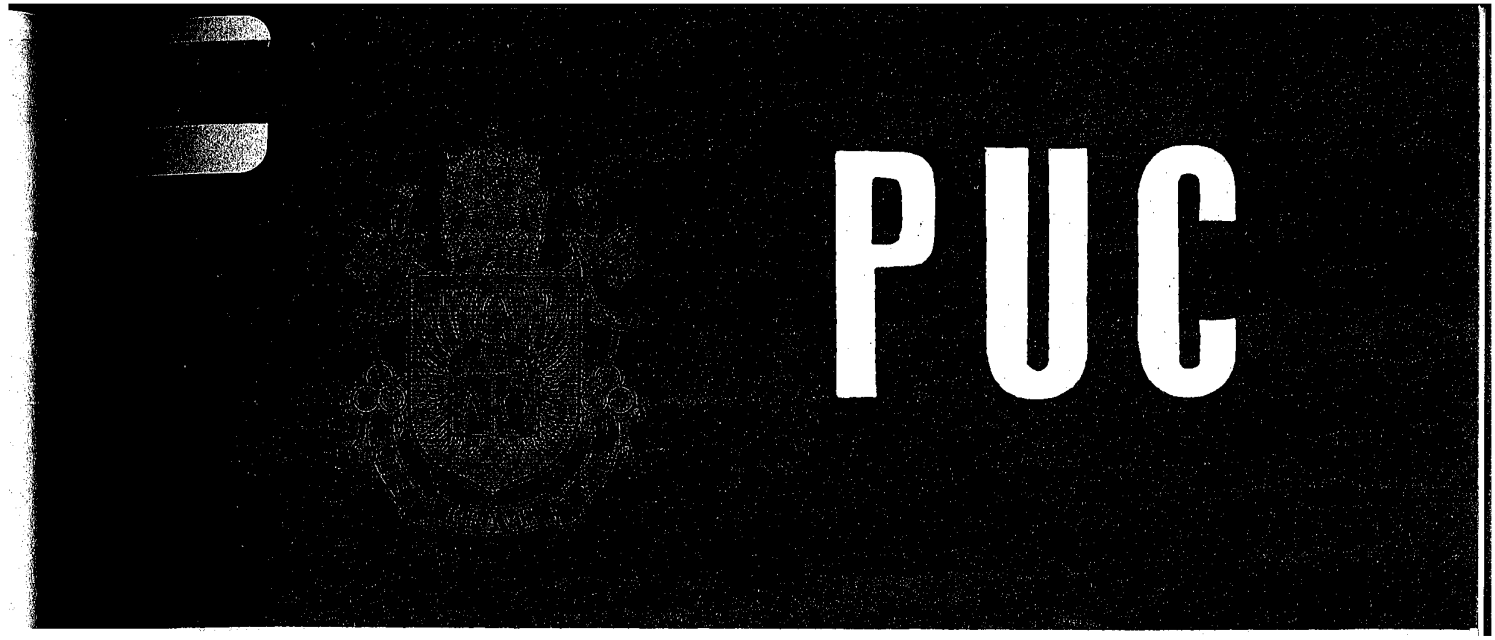

Series: Monographs in Computer Science
and Computer Applications

Nº 2/75

AN EFFICIENT WAY OF SELECTING LEXICAL TYPES

by

Sergio E. R. de Carvalho

Computer Science Department

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

AN EFFICIENT WAY OF SELECTING LEXICAL TYPES

Sergio E. R. de Carvalho
Associate Professor
Computer Science Department
(Informática) PUC/RJ

Series Editor: Prof. S. M. dos Santos February/ 1975

ABSTRACT

When constructing an analyzer for a given grammar, the choice of the lexical types for the grammar is usually based solely upon ad hoc criteria. In this paper we show a way in which more efficient cri - teria can be used to that effect. First we introduce some properties lexical analyzers should necessarily possess, obtaining a set of candidates for lexical types. Then we show how for a particular analyzer model the set of candidates can be refined, until we are left with a set of nonterminals which can be considered lexical types for the given grammar.

1. INTRODUCTION AND PREVIEW

The analysis phase of most compilers consists basically of two sections: lexical analysis and syntactical analysis. Algorithms for performing syntactical analysis have been described extensively in the literature (Aho and Ullman [1] provide a very thorough description of these methods). However, far less attention has been given to the much simpler lexical analysis, possibly due to the fact that a very important phase in this section, the choice of lexical tokens, is in most cases done using ad hoc criteria.

In this paper we shall be concerned with an efficient way of selecting lexical types for a given grammar. First we must state clearly what we mean by lexical token and lexical type. As in [1], we consider a lexical token to be a pair of the form (lexical type, pointer), where "lexical type" is the name of some lexical category, and "pointer" is a reference to some data corresponding to the lexical token in question. In this paper we are concerned with the recognition of substrings of terminal symbols that can be reduced during lexical analysis, and not with the other (semantic) actions taken when such substrings are recognized. Therefore we consider only the first element of the pair above.

The choice of the lexical types of a given grammar is (as stated in Aho and Ullman [1] and Gries [2]) in some cases somewhat arbitrary. The usual selection contains nonterminals like <identifier> and <unsigned integer> (as defined in Algol 60 [3]), multiple character delimiters as the exponentiation operator '**' in FORTRAN, and reserved or keywords like for instance DECLARE, DO. The method of selection we shall describe considers only nonterminals as possible lexical types, being however easily extended to consider for instance keywords as well [4].

The method here described is based upon a particular analyzer model, to be introduced in section 2. In this section we describe the basic structures of both the lexical and the syntactical analyzers we shall work with.

In section 3 we start by considering some properties the non-terminals of the grammar must fulfill in order to be lexical types. At the end of this phase we have a set of candidates for lexical types. Next we obtain from the basic lexical and syntactical analyzers the information necessary to refine the set of candidates. This consists in finding when a substring of an input string can be recognized as an instance of two or more candidates for lexical types. The main result of section 3 is the production of a graph showing precisely which candidates generate the condition above, and in what points during the analysis of input strings that situation may occur.

In section 4 we try to eliminate the ambiguities shown in that graph by considering some lookahead information, which we obtain from our syntactical analyzer. The extra context thus considered will be used to determine exactly when a certain substring constitutes an instance of a certain candidate. The situation may occur, however, in which no lookahead is sufficient to separate instances of candidates. In the end of section 4 we present a criterium for eliminating some candidates from the set, until we are left with a set of possible lexical types for the grammar.

Section 5 briefly mentions some practical results obtained with the analyzer model and the selection criteria here described.

2. THE ANALYZER MODEL

The model for the syntactical analyzer is the characteristic finite state machine (CFSM) of De Remer [5,6]. This model has been shown [7] to be comparable in efficiency to precedence models [8,9]. We shall now informally define a CFSM corresponding to a given grammar $G = (N, \Sigma, P, S)$. Our notation and terminology, in what follows, are that of Aho and Ullman [1].

First we assume the existence of a set $\#1, \#2, \ldots, \#p$ of symbols not in $N \cup \Sigma$, where $p = |P|$. Then we order the productions of P, and associate the symbol $\#1$ with production 1, $\#2$ with production 2, ..., $\#p$ with production p. Suppose now that the j^{th} production is $A \to \gamma$, and that $\beta = \alpha A w$ and $\beta' = \alpha\gamma w$ are rightmost sentential forms of G such that there exists a derivation $S \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha\gamma w$. Then the string $\alpha\gamma\#j$ is a characteristic string of G. A CFSM corresponding to a grammar G is a reduced, deterministic finite state machine which recognizes the set of all characteristic strings of G. For a rigorous definition of CFSM's the reader is referred to [5,6]. As an example, let G_1 be given by the productions below:

$\#1$: $S \to \vdash a \dashv$
$\#2$: $S \to \vdash A \dashv$
$\#3$: $A \to b$
$\#4$: $A \to A\ b$

The CFSM corresponding to G_1 is shown in Fig. 1, with its states labelled.

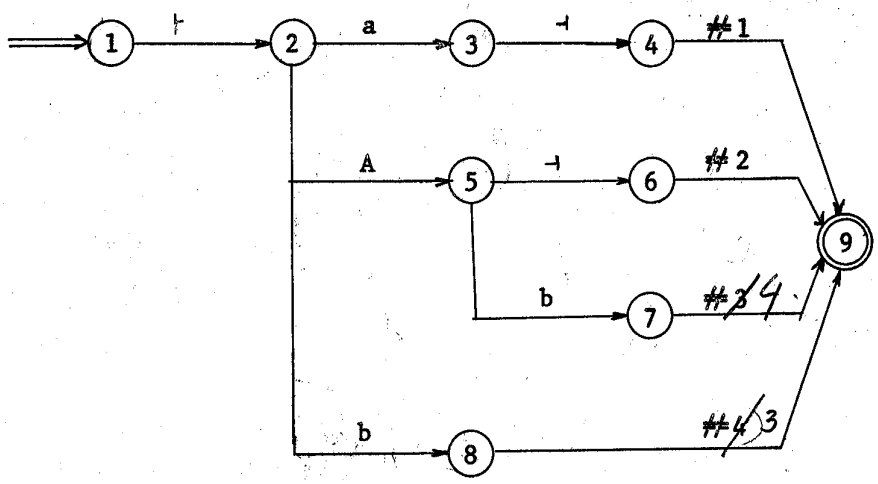

Fig. 1: Example of a CFSM

To illustrate the use of a CFSM as the basic model for a syntactical analyzer, let us consider the string $w = \vdash bb\dashv$. To see whether $w \in L(G_1)$, we input w to the CFSM of Fig. 1. After reading the prefix $\vdash b$, we reach state 8. The #3-transition from 8 indicates that a reduction has now to be performed, according to production #3. In other words, we replace the right side of #3 (b) by the corresponding left side (A), obtaining the sentential form $\vdash Ab\dashv$. Submitting this sentential form to the CFSM, we reach this time state 7, and a new reduction, according to production #4, is indicated. The new sentential form obtained is $\vdash A\dashv$, which, when input to the CFSM, reaches state 6. Performing the reduction according to production #2, we obtain the sentential form S, which signals the successful end of the recognizing process.

The model for the lexical analyzer is a deterministic finite automaton constructed from the individual finite automata accepting the languages generated by the lexical types, which we shall obtain.

The basic idea behind the analyzer model is to have the lexical and the syntactical analyzers connected in such a way that whenever instances of lexical types can occur in the input string, the lexical analyzer

is called by the syntactical analyzer and tries to find the longest possible instance of some lexical type. This system has the advantage that the lexical analyzer knows in advance which lexical types can occur at any particular call. In Aho and Ullman's terminology [1] this type of lexical analyzer is called an indirect lexical analyzer.

It is easy to see that CFSM's are well suited for such an analysis scheme, since we can promptly identify all the states of the CFSM which are origins of transitions labelled by lexical types. Therefore we know all the points during the parsing of an input string where the lexical analyzer should be called upon to recognize an instance of a lexical type. To illustrate, let us suppose that in the grammar G_1 above 'a' and 'b' are to be considered as lexical types. From state 2 of the CFSM (Fig. 1) we call the lexical analyzer to recognize either an instance of 'a' or of 'b'; from state 5 we call the lexical analyzer to recognize an instance of 'b'.

At this point it is interesting to point out one advantage of an indirect lexical analyzer. Suppose we are given the input string $\vdash ba \dashv$, which we know not to belong to $L(G_1)$. After performing the first reduction, we have the sentential form $\vdash Aa \dashv$, which, when input to the CFSM, reaches state 5. Now a call to the lexical analyzer is executed, in order that an instance of 'b' is recognized. Obviously the lexical analyzer fails, originating some error message. A less smart lexical analyzer would probably recognize the following 'a' as a possible instance of a valid lexical type, and the error would only be detected upon return of the control to the syntactical analyzer.

3. CANDIDATES FOR LEXICAL TYPES

In this section we shall establish some properties that candidates for lexical types should possess. We also show how, in an analyzer as described in section 2, we can efficiently narrow down the choice of the lexical types, to be finally chosen in section 4.

Let us suppose we are given a grammar G, and the task of constructing an analyzer for G. We can start by constructing the CFSM for G. Our next step would be to select a set of lexical types for G. This is usually done in an ad hoc way, and a typical result is that identifiers, constants, keywords and delimiters are chosen as lexical types. Although this is in most cases good enough for lexical analysis purposes (simplify input strings so that the more important syntactical analysis can be made simpler) we may want in some cases to have our lexical analyzer recognize more sizeable portions of our input strings. In order to be able to possibly increase the set of lexical types of a given grammar, let us first of all informally examine some conditions that candidates for lexical types should fulfill. From now on we shall focus our attention on candidates which are nonterminals of the given grammar.

Since our lexical analyzer is basically a finite state machine, the first condition a certain nonterminal must meet is that the language it generates must be a regular language. Let A be a nonterminal in N. Consider the grammar $G_A = (N,\Sigma,P,A)$. Then $L(G_A)$ must be regular for A to be a candidate for a lexical type. In [4] we show a way in which for certain constructs of a grammar we can find out whether the language generated is or is not regular. Clearly, we may find nonterminals satisfying this condition in all "levels" in a grammar. To illustrate, the nonterminal <procedure heading>, in Algol 60, generates a regular language. The same is obviously true for nonterminals such as <identifier> and <unsigned integer>.

Our second condition can be motivated by another Algol 60 example. Consider the nonterminal <identifier list>. The language it generates is regular, and consists of sequences of identifiers, separated by commas.

However, since there are semantic actions associated with each identifier of the list, we shall delete <identifier list> from our set of candidates. The second condition then further restricts the set of candidates by eliminating those which do not satisfy the property of having a certain unity with respect to semantic actions.

The set of candidates can be refined in a third way. Consider the Algol 60 nonterminal <digit>. Obviously <digit> satisfies the above conditions. However, examining the remainder of the grammar, we find that both <identifier> and <unsigned integer> also satisfy conditions 1 and 2. Furthermore, <digit> occurs in Algol only as a part of <identifier> and <unsigned integer>. For this reason, we prefer to keep the latter two candidates, and delete <digit> from the set. In [4] it is shown how we can use simple concepts from graph theory to implement condition 3.

After examining the grammar with respect to the criteria above, we end up with a set of nonterminals which are candidates for lexical types.

To show how the selection of lexical types can be further refined, we must go back to the analyzer model. Suppose A and B are in the set of candidates, and let q be a state of the CFSM generating an A-transition and a B-transition. Suppose now that the regular languages $L(G_A)$ and $L(G_B)$ are disjoint. This means that in the finite automaton which will perform the lexical analysis there will be no final state accepting a string in both $L(G_A)$ and $L(G_B)$. Suppose now that when analyzing an input string we arrive at state q, and let w be the substring yet to be analyzed. Since we may have an instance of a candidate (either A or B) as a prefix of w, the lexical analyzer is called; and since we recognize either an instance of A or of B, but not both, we see that there is no "collision" between the two candidates, and therefore both can be lexical types.

Suppose now $L(G_A)$ and $L(G_B)$ are not disjoint. In this case the lexical analyzer could recognize instances in common to both candidates, by reaching a common final state. We shall try to solve this collision by

looking ahead in the substring yet to be analyzed. If instances of A and B can always be separated by a reasonable amount of lookhead, then we say that as far as the state q is concerned, A and B can still be lexical types. Note that A and B have to be separated with respect to each state of the CFSM originating both A- and B-transitions.

We proceed now to formalize the notions introduced above. Let $C = \{A_1, A_2, \ldots, A_n\}$ be the set of candidates for lexical types. First we must construct a deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ accepting the regular language $L(G_{A_1}) \cup L(G_{A_2}) \cup \ldots \cup L(G_{A_n})$ (the necessary steps are described for instance in [1]). Consider now the set $F' \subset F$ containing all final states of M accepting strings in $L(G_{A_i})$ and $L(G_{A_j})$, for $i \neq j$. If F' is empty, all candidates can be lexical types. If not, then we introduce the following

Definition: The final states graph (V_F, E_F) is a labelled graph such that:

(i) the set of vertices V_F corresponds to the set of candidates whose languages contain strings accepted by states in F';

(ii) edge (A_i, A_j) is in E_F with label q if and only if instances of A_i and A_j are recognized by q in F'.

As an example, suppose the candidates are $A_1, A_2, \ldots, A_5$, and suppose $F' = \{b, c, d\}$. Let b accept instances of A_1, A_2, A_3; let c accept instances of A_2 and A_4; d instances of A_4 and A_5. The final states graph for this case is shown in Fig. 2.

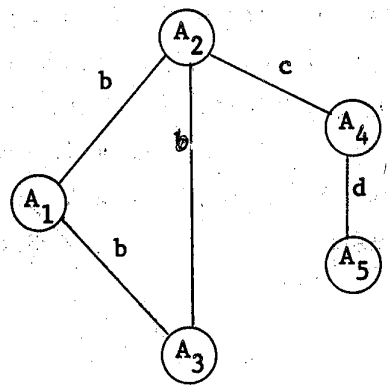

Fig.2: Example of a final states graph.

We must now find out which states of the CFSM originate transitions under more than one candidate.

Definition: A transition in the CFSM is called a token transition if its label is a nonterminal in C.

Definition: A state q in the CFSM is called a single token state if q is the origin of exactly one token transition. A state q in the CFSM is called a multiple token state if q is the origin of more than one token transition.

Definition: The token states graph (V_T, E_T) is a labelled graph such that:

(i) the set of vertices V_T corresponds to the set of labels of token transitions originating in multiple token states of the CFSM;

(ii) edge (A_i, A_j) is in E_T with label q if and only if there exists both A_i - and A_j - transitions from token state q.

Clearly we don't have to worry when, in analyzing a string, we reach a single token state, since in this case instances of only one lexical type can be recognized; the occurence of any other causes an error. To exemplify, consider Fig. 3 below. There, the multiple token state 1 originates transitions labelled A_1 and A_2; state 3 originates transitions labelled A_4 and A_5, and state 2 originates transitions A_2, A_3 and A_5.

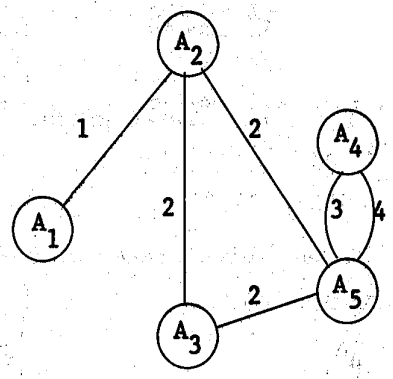

Fig. 3: Example of a token states graph.

Given the two graphs above, it is easy to find out which candidates have to be separated by lookahead.

Definition: The separation graph (V_S, E_S) is a labelled graph such that:

(i) $V_S = V_F \cap V_T$;

(ii) edge (A_i, A_j) is in E_S with label (p, q) if and only if edge (A_i, A_j) is in E_F with label p and in E_T with label q.

The separation graph resulting from the previous examples is illustrated below.

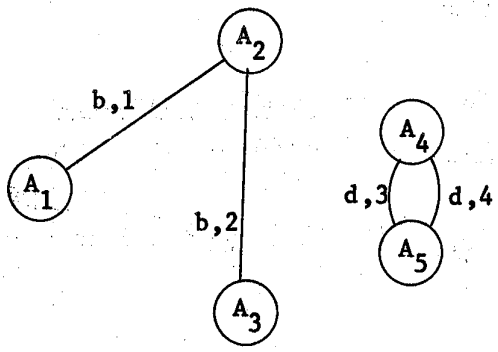

Fig. 4: Example of a separation graph.

We shall use the information contained in the separation graph to finally select the lexical types, as will be seen in the next section.

4. THE FINAL SELECTION

An edge (A_i, A_j) labelled (p, q) in the separation graph means that lookahead sets for A_i and A_j with respect to the token state q have to be found in the CFSM. If these sets are disjoint, we attach this information to the common final state p of the finite automaton (lexical analyzer) to separate strings in $L(G_{A_i})$ and $L(G_{A_j})$. If for each pair of vertices connected by an edge in the separation graph the corresponding lookahead sets are disjoint, then we do not have collisions and all candidates are lexical types. If, however, for some A_i and A_j the lookahead sets intersect, then one of them must be eliminated from the set of candidates.

Let us now consider the lookahead information. Since we are using this information to separate lexical types, it is reasonable to assume that a lookahead of one symbol should suffice. Let q be a state of the CFSM originating an A_i - transition, for some A_i in C, and let q' be the state

reached by this transition.

Definition: The set of immediate successors of A_i with respect to q is denoted by I_{qA_i} and defined as follows:

$I_{qA_i} = \{b \in \Sigma \mid$ there exists a b-transition from $q'\}$.

Definition: The set of remote successors of A_i with respect to q is denoted by R_{qA_i} and is recursively defined as follows:

(i) if there are no #j-transitions (reduce transitions) from q', then $R_{qA_i} = \emptyset$; otherwise

(ii) if there is at least one #j-transition from q', then, for each such transition:

1. Find the left side of the #j production, say B_j;

2. Find all B_j-transition in the CFSM, and find the states reached by such transitions. For each such state:

   a. if some b-transition is originated, then

   $$R_{qA_i} = R_{qA_i} \cup \{b\};$$

   b. if some #m-transition is originated, then

   $R_{qA_i} = R_{qA_i} \cup R_{qB_m}$, where B_m is the left side of the #m production.

The sets above can be easily found from inspection of the CFSM.

Definition: The lookahead set corresponding to a token state q and to a nonterminal A_i is denoted by L_{qA_i} and is defined as:

$$L_{qA_i} = I_{qA_i} \cup R_{qA_i} .$$

After finding the lookahead sets indicated by the separation graph, we can delete from the graph all edges connecting two vertices labelled by candidates which can be separated by lookahead with respect to a certain state of the CFSM. Let us call the resulting graph the collision graph, denoted by (V_C, E_C). The final step in the obtention of the lexical types for the given grammar consists in the eliminating, from the set of candidates, of some nonterminals corresponding to vertices of V_C.

If we want the set of lexical types to be as large as possible, we can start the elimination process from the nodes having the greatest number of edges incident to them, and proceed to do so until no more edges remain in the graph. In this way we delete the minimum number of candidates from the set. Other criteria can be used as well, and decisions based on real experience with the language, simplification of lexical analyzer construction, and so on, can be made from the collision graph.

Once the lexical types are chosen, the lexical analyzer has to be modified to accomodate the necessary lookahead information. This can be easily done [4] by adding to each common final state "lookahead transitions" corresponding to the nonterminals whose instances are accepted by the state.

5. CONCLUSIONS

The analyzer model described in section 2 was used in the compiler for LDS (a language for systems design) constructed at the Pontificia Universidade Católica in Rio de Janeiro. The methods of sections 3 and 4 were used in the selection of the lexical types for that compiler. It was found that our systematic selection eliminated some of the guesswork one usually has to do when performing this task. Also, this approach was found to be a good help in improving both the language and the grammar's design, for example by pointing out where some special delimiters could be used in order to simplify the analysis phase.

BIBLIOGRAPH

1. AHO, A.V. and ULLMAN, J.D. The Theory of Parsing, Translation and Compiling, Vol. 1, Prentice-Hall, N.J., 1972.

2. GRIES, D. Compiler Construction for Digital Computers, Wiley, N.Y., 1971.

3. NAUR, P. (ed.). Revised report on the algorithmic language ALGOL 60. Comm. ACM 6:1, 1-17, 1963.

4. de CARVALHO, S.E.R., Design of interrelated lexical and syntactical analyzers. Ph.D. Thesis, University of Waterloo, Waterloo, Ontario,1973.

5. DE REMER, F.L. Practical translators for LR(k) languages. Ph.D. Thesis., M.I.T., Cambridge, Mass., 1969.

6. DE REMER, F.L. Simple LR(k) grammars. Comm. ACM 14:7, 453-460, 1971.

7. LALONDE, W.R., LEE, E.S. and HORNING, J.J. An LALR(k) parser generator. Proc.IFIP Congress 71, TA-3. North Holland Publ.Comp., Netherlands, 153-157.

8. FLOYD, R.W. Syntactic analysis and operator precedence. J. ACM 10:3, 316-333, 1963.

9. WIRTH, N. and WEBER, H. EULER - a generalization of ALGOL and its formal definition, Part 1. Comm. ACM 9:1, 13-23, 1966.