stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 86/77 AUGUSTUS

L.J.M. GEURTS & L.G.L.T. MEERTENS

KEYWORD GRAMMARS

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A36, 68A42, 68A50,

ACM-Computer Review Categories: 4.12, 4.22, 5.23

Keyword grammars*

by

L.J.M. Geurts & L.G.L.T. Meertens

ABSTRACT

A simple condition is derived for the "keyword skeletons" in a language, guaranteeing that the language may be parsed by a simple no-backup parser that can be generated from the grammar of the language without preprocessing. It is also shown that this condition is, in some sense, necessary.

KEY WORDS & PHRASES: deterministic top-down parsing
LL(1) grammar
s-grammar

---

*This report is a preprint of a paper that will appear in the Proceedings of the Fifth Annual International Conference on the Implementation and Design of Algorithmic Languages, Guidel, 16 - 18 May 1977.

0. INTRODUCTION

In [1] an outline is given of a project to design a language, aimed at the "BASIC market" but equipped with an appropriate arsenal of structured-programming tools. This language has provisionally been called B. In the same paper, a line of reasoning is given that leads to the following approach for the syntax of B.

A program is composed of a sequence of statements. Each statement type is composed of "keywords", statements and expressions. Some possible statement types are "PUT expression IN expression", "WHILE expression statement" and "PASS".

For the purpose of the subsequent discussion, a "keyword" may be any terminal symbol, but in order to understand some issues it may be helpful to think of a keyword as being a word delimiter, specifically. A statement always begins with a keyword. An expression, on the other hand, never begins with a keyword (but it may begin with or even consist of an identifier that looks the same). Expressions may be thought of as being conventional formulas, which have a syntax of their own and which can be parsed by any conventional bottom-up technique. For this paper, however, only one property of expressions is relevant: if e is an expression and k is a keyword, then ekz is not an expression for any string z.

We define the "constructor" of a statement type to be the sequence of keywords used for its composition. So the set of constructors corresponding to the statement types given above is {PUT IN, WHILE, PASS}. In [1] a restriction is mentioned on the set of constructors, namely that the first keyword of each constructor be "unique", i.e., distinct from the remaining keywords of that constructor and from all keywords of other constructors. It is claimed there that under this "uniqueness condition" it is possible to use with great liberty almost any combination of keywords, statements and expressions for a statement type.

The aim of this paper is to investigate this claim more precisely. To this purpose, the notions of "keyword graph" and "keyword grammar" are introduced. It will be shown that for a keyword grammar satisfying the uniqueness condition a very simple top-down deterministic parser may be derived in a straightforward way. Not only is the uniqueness condition sufficient for this, but, as will be shown, if only information present in

the set of constructors may be considered, it is also necessary. This is especially relevant since we are considering the possibility of allowing user-defined statement types in B. For the admissibility of such a feature, not only should a condition be imposed to prevent ambiguity, but this condition should also ensure the applicability of a parsing method chosen beforehand and be extremely simple for the user to apply.

Finally, it will be shown that the language described by a keyword grammar satisfying the uniqueness condition has a prefix property. This property is relevant in view of the treatment of layout in B, where each statement may start at a new line (which is also obligatory, unless that statement is the last part of another statement), in conjunction with the wish to check each line separately for syntactic correctness.

Section 1 surveys some (well known) elements of top-down parsing, as given in [2]. Mental digestion of this section is not necessary to follow the remainder of the paper. In section 2 keyword graphs and keyword grammars are introduced, in order to make the paper more or less self-contained. Section 3 contains the results. Some final remarks are made in section 4.

Throughout the paper the treatment is informal. For example, we refrain from defining a keyword grammar as being a 10-tuple <Q,W,E,R,T,Y,U,I,O,P>, et cetera. In general, we aim at insight rather than rigour. Readers familiar with [2] should have no difficulty in giving a formal treatment. Some elementary familiarity with context-free grammars and parsing is assumed. The terminology is that of [2].

1. TOP-DOWN PARSING

A deterministic top-down parsing method is a method that allows the construction of a parse tree for input strings of terminal symbols produced by some given grammar in the following way: Start with a partial parse tree consisting only of the top node, which is labelled with the initial symbol of the grammar. At each step, the leftmost "untreated" node in the parse tree is taken, and treated as follows. If the node is labelled with a terminal symbol, it is equal to the first symbol of the input string, and that symbol is deleted. If, on the other hand, the node is labelled with a nonterminal symbol, a production for that nonterminal symbol is selected by

inspecting the first k symbols of the input string, and the parse tree is developed accordingly.

If the input string was indeed produced by the given grammar, the process should terminate with a complete parse tree (all nodes treated) and an empty input string (all symbols deleted). For this method to work, the selection of a production should be completely determined by the grammar, given the nonterminal symbol and the first k symbols of the input string, and it should guarantee that terminal symbols will indeed be present in the input string when expected. Formal conditions for this are given in [2], and a grammar satisfying these conditions is known as an LL(k) grammar. Since each LL(k) grammar is also an LR(k) grammar, the LL(k) property also allows the application of a bottom-up parsing technique.

It is not necessary to actually construct a parse tree or, if one is constructed, to consult its nodes during parsing. Top-down methods may be easily implemented by a system of mutually recursive routines, one for each nonterminal symbol. During the parsing process, the untreated part of the tree is then reflected in the link stack.

In [2], a parsing machine (PM) is defined, and a straightforward way is given to transcribe a grammar into a PM program. The general PM works with a system of mutually recursive routines. The difference with the deterministic method given above is that it does not select a production, but instead tries the productions one by one, until it finds one that succeeds. If a production fails, e.g. by the absence of an expected terminal symbol, the machine "backs up" by reinserting the symbols that have possibly been deleted and tries the next production. If all productions for a certain nonterminal symbol fail, then the production from which the routine for that nonterminal symbol was called also fails in turn. The transcription of grammars into ALGOL 60 or ALGOL 68 programs implementing this method is described in [4].

A case of special interest is the restriction of PM by allowing no back-up. This requires that if the wrong production is tried, failure occurs within that production on the very first symbol of the input string. If on failure the PM should already have deleted symbols from the input string that correspond to the failing production, it simply fails on the whole input string. It is shown in [2] that under this restriction exactly those languages may be recognized that are produced by an LL(1) grammar.

However, the straightforward transcription of an LL(1) grammar may yield an incorrect PM program. In general, it is necessary to turn the grammar first into some standard form. For some grammars this is unnecessary, provided that the productions are properly arranged, i.e., tried in the right order. It is also possible that each arrangement of the alternative productions is fine and allows a straightforward transcription into a correct PM program. We shall abuse the notation "LL(1$^-$)" to indicate such grammars.

Below a property very similar to LL(1$^-$)-ness will be derived for keyword grammars satisfying the uniqueness condition. Examples of LL(1$^-$) grammars are the s-grammars introduced in [3]. It is not necessary to give a formal definition of LL(1$^-$) grammars here, since we will derive the necessary properties directly from a specialized version of the no-backup PM, which is tailored to keyword grammars and performs a recursive walk on a "keyword graph". However, we mention (without proof) a simple condition that, in conjunction with the well-known condition for LL(1)-ness, gives LL(1$^-$)-ness:

An LL(1) grammar is of type LL(1$^-$) iff none of its (reachable) "nonfalse" nonterminal symbols has two or more productions, where a nonterminal symbol A is "nonfalse" iff the grammar has a production of the form

a) A -> #, where # denotes the empty string,

or b) A -> $X_1 \ldots X_n$, where X_1 is a nonfalse nonterminal symbol.

Note that a production of the form A -> # in an LL(1$^-$) grammar is rather silly, since it is the only production with A as a left-hand side, so that A might as well be deleted throughout the grammar. So, without substantial loss of expressive power, all #-productions may be removed (with the trivial exception of a grammar for the language {#}). For a #-free grammar, all nonterminal symbols are easily seen to be nonfalse, so a #-free LL(1) grammar is of type LL(1$^-$).

2. KEYWORD GRAPHS AND KEYWORD GRAMMARS

A "keyword graph" over a set of "keywords" (terminal symbols) and a set of "statement types" (nonterminal symbols) is a set of directed graphs, one for each statement type, each of which has two distinguished (and distinct)

vertices: the "source" and the "sink". Each vertex lies on an (oriented) path from the source to the sink of one of the directed graphs. An arc from a vertex u to a vertex v is called an "out-arc" of u and an "in-arc" of v. A vertex with at least two out-arcs is a "fork". Each arc is either labelled with a keyword, or with a statement type, or with the special symbol E (which may be interpreted as "expression"). In addition, a keyword graph satisfies the following three restrictions:

(R1) the out-arcs of the source are labelled with keywords only; these are called "first keywords" of the corresponding statement type;

(R2) no vertex has both an in- and an out-arc labelled with E;

(R3) a sink has no out-arcs, but only in arcs. (A similar condition might be formulated for sources, but turns out to be irrelevant.)

(The reason for these restrictions will become apparent later on.)

A keyword graph over the set of keywords {BEGIN, END, PUT, IN, CASE, ELSE} and the set of statement types {S} is shown in figure 1, and one over

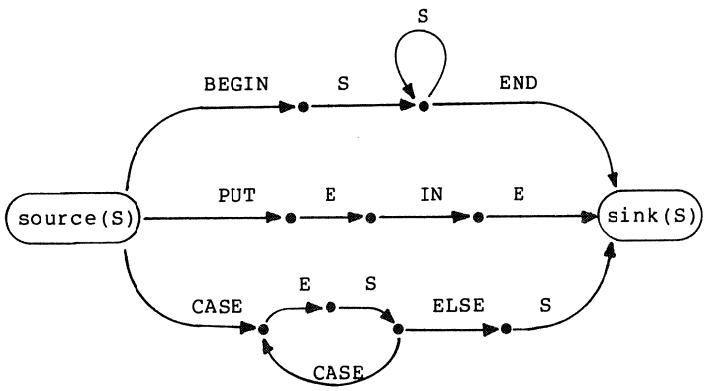

Figure 1. A keyword graph with one statement type.

the same set of keywords but over the set of statement types $\{S_1, S_2, S_3\}$ is shown in figure 2.

Given a context-free language L(E) for the expressions, a keyword graph determines a context-free language for each of the statement types, as follows. First, each arc labelled with a statement type S is relabelled with a new terminal symbol T(S). E is considered as a terminal symbol. Then, the subgraph for each statement type is a finite-state automaton that describes a regular language. Each string of this language corresponds to a walk through the subgraph from source to sink, and is composed of the

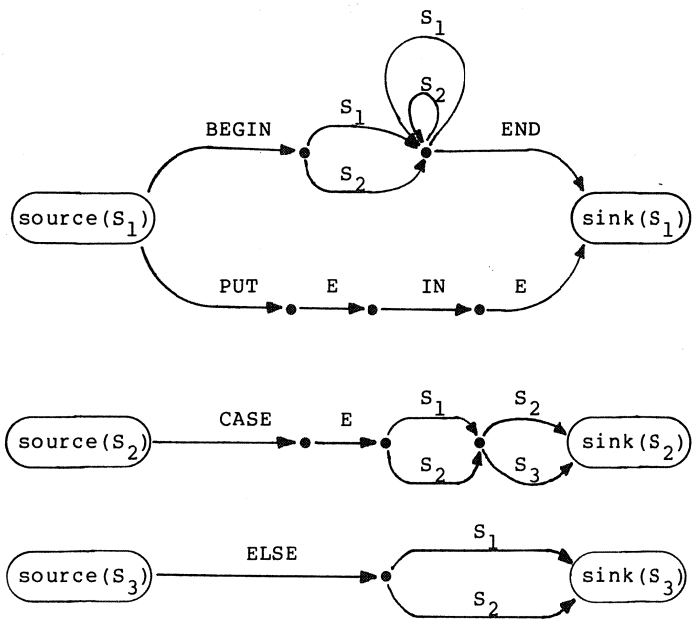

Figure 2. A keyword graph with three statement types.

labels of the arcs traversed. These languages may be described by context-free grammars. Next, change each T(S) back into S. Similarly, reclassify E as nonterminal and add the productions of its context-free grammar. It should be clear that each string produced by this grammar may also be produced by a "recursive walk" over the keyword graph, using an (initially empty) stack of vertices, as follows: Start at the source of the subgraph for the statement type for which a string must be produced. Each time, follow an arc leading from the current vertex. (This is always possible, unless the current vertex is the sink, since there is a path leading from it to the sink.) If the arc is labelled with a keyword, emit that keyword. If it is labelled with E, emit a string from L(E). If it is labelled with a statement type S, push the vertex to which the arc is leading on the stack and start a walk on the subgraph for S. On reaching a sink, the stack is popped and the walk proceeds from the (previous) top vertex. If the stack is empty, however, the walk is complete.

For later use, we give a more precise definition, which is not operational but recursive instead:

A recursive walk for a statement type S through a keyword graph is a finite sequence of arcs $u_1\ w_1\ \ldots\ u_n\ w_n$, $n \geq 1$, where

- u_1 is an out-arc of the source, and u_n is an in-arc of the sink, of the subgraph for S;
- each u_i, i = 2, ... , n, is an out-arc of the vertex of which u_{i-1} is an in-arc;
- each w_i, i = 1, ... , n, is a recursive walk for T if u_i is labelled with a statement type T, and is empty otherwise. (Note that w_1 is always empty.)

The "emission" of a recursive walk w is the string obtained by replacing each arc of w by

- the keyword k if it is labelled with k;
- empty if it is labelled with a statement type;
- a string from L(E) if it is labelled with E.

Because of the last type of replacements, w may emit several strings. The definition may be extended in an obvious way to partial walks.

The purpose of introducing this notion of a walk is that we want to use a keyword graph to parse a given input string by trying to walk the graph, directed by the input string. This corresponds to the performance of the PM mentioned in section 1.

A "keyword grammar" is a context-free grammar (whose terminal symbols are again called keywords and whose nonterminal symbols are called statement types), having a special symbol E (the classification of which is - for the moment - left open). However, where a conventional context-free grammar allows simply one nonterminal symbol at a time in the right-hand side of a production, we allow a non-empty set of statement types. Moreover, letting X stand for such a set of statement types, X^* may also be used, indicating zero or more repetitions of X. In addition, a keyword grammar satisfies the following three restrictions:

(R1′) the right-hand side of each production begins with a keyword;

(R2′) no production contains two consecutive symbols E;

(R3′) each form X^* is followed by a keyword.

But for the notational extensions and the special symbol E, keyword grammars would be s-grammars. These extensions can be undone, of course, by considering each X and X^* as new nonterminal symbols and by adding productions X^* -> #, X^* -> XX^* and X -> S for each S in X. The result is a conventional context-free grammar, which in general, however, will no

longer be a keyword grammar.

An example of a keyword grammar is given by

S1 -> BEGIN {S1, S2} {S1, S2}* END
S1 -> PUT E IN E
S2 -> CASE E {S1, S2} {S2, S3}
S3 -> ELSE {S1, S2}.

The "constructor set" of a keyword grammar is the set of sequences of keywords, one for each production, consisting of the keywords occurring in that production, taken in the same order. (Properly speaking, we should use the term "multiset" rather than "set", since we have not yet excluded the possibility of two productions with identical constructors.)

The constructor set of the example grammar is {BEGIN END, PUT IN, CASE, ELSE}.

It is straightforward to derive a keyword graph from a keyword grammar, and the keyword graph for the above grammar is that of figure 2. Forks in such keyword graphs are always of one of three types:

- sources, all of whose out-arcs are labelled with keywords;
- forks corresponding to a set X of statement types, where all out-arcs are labelled with statement types;
- forks corresponding to an X^*, where in addition one out-arc is labelled with the keyword following X^*.

The inverse transformation is not so straightforward; in general it is only possible by structural changes.

3. CONDITIONS FOR DETERMINISM

Let a keyword graph and an input string derived from a recursive walk over the graph be given, and consider a machine able to walk the graph. To this purpose the machine has an instruction "read expression", which, as far as it is concerned, functions as a black box. At any time, it has only knowledge of the first symbol of the input string, which it may delete. Rather than forbidding the reinsertion of deleted symbols (as for PM), we forbid the machine to backtrack in the graph, so it can follow an arc only if it is sure that its choice is correct (under the assumption of having a correct input string). Under this restriction, reinsertion is simply no

help, so the same effect is achieved. Similarly, we need not specifically require that the machine is aware of the out-arcs of the current vertex only, since information from subsequent arcs is irrelevant when only the next first input symbol is known.

What problems may the machine encounter? As long as there is no fork, its choice is determined by force. Suppose therefore that a choice between several arcs presents itself. If one arc is labelled with E, the situation is hopeless, since the machine has no knowledge of L(E) and therefore no way to decide between a continuation of the walk over this arc and any other continuation. Another obviously awkward situation arises if two arcs are labelled with the same keyword. If several arcs are labelled with statement types, the machine may only select one that has the current first input symbol as a first keyword. If the sets of first keywords for these types are not disjoint, it will find itself in a quandary if some shared first keyword happens to be the current input symbol. The same problem may arise if one arc is labelled with a statement type and another with a first keyword of that statement type.

We want to prove that this informal examination has given precisely the situations that must be excluded to guarantee a determined choice for all input strings and all possible expression languages L(E). In order to show this, we need a more precise definition of "determined choice":

A keyword graph has the "determined-choice property" iff for all L(E), for all pairs paw and pa′w′ of recursive walks for the same statement type, where p denotes a partial walk and a and a′ denote arcs, and for all strings xs of aw and x′s′ of a′w′, where x and x′ denote symbols, x = x′ implies a = a′.

Note that the determined-choice property is not sufficient to guarantee by itself that the same walk will be re-produced that emitted the input string. For the instruction "read expression" might not delete exactly the same part as was emitted as an element of L(E). Therefore, we simply postulate that if a is labelled with E, then for any emissions e of a and s of w, where aw is part of a recursive walk, read expression started on es will exactly delete e. (This might be an outright impossible requirement if, e.g., es is genuinely ambiguous. If the natural assumption for a bottom-up device is made that read expression always deletes a maximal expression, then a simple sufficient condition on L(E) that excludes such

ambiguities is the ekz-condition mentioned in the introduction. The proof of this statement is omitted.)

THEOREM 1. A keyword graph has the determined-choice property iff it satisfies the following four conditions:

(i) no out-arc of a fork is labelled with E;

(ii) no two out-arcs of a fork are labelled with the same keyword k;

(iii) no two out-arcs of a fork are labelled with statement types S_1 and S_2 such that S_1 and S_2 have a common first keyword k;

(iv) no two out-arcs of a fork are labelled with a statement type S and a keyword k such that k is a first keyword of S.

Proof. (If part) Let G be a keyword graph having the determined-choice property. Consider two recursive walks paw and pa′w′ and strings xs and x′s′ emitted by aw and a′w′, and suppose that x = x′. We will show that a ≠ a′ leads to a contradiction. Obviously, if a ≠ a′, a and a′ are out-arcs of a fork. Because of condition (i), their labels are keywords or statement types. If a is labelled with a keyword k, we find immediately that x = k. Otherwise, a is labelled with some statement type S. By the definition of "recursive walk", the first arc of w is an out-arc of the source for S, so its label is some first keyword k of S. In obtaining a string xs emitted by aw, a is replaced by empty, and the first arc of w by k. Therefore, x = k. For a′ we find in the same way that x′ is equal to some keyword k′ that either labels a′ or is a first keyword of the statement type labelling a′. Since x = x′, k = k′, so one of the conditions (ii) through (iv) is immediately violated. Therefore, a = a′.

(Only-if part) For each violation of one of the four conditions a keyword graph does not have the determined-choice property. If condition (i) is violated, there must be some fork (reachable with a partial walk p) with an out-arc a labelled with E and at least one other out-arc a′. If a′ is labelled with E too, it is obvious that any pair of continuations aw and a′w′ have emissions xs and x′s′ with x = x′, provided that L(E) is chosen such as to contain at least one non-empty string. If a′ is labelled any other way, let x′s′ be the emission of some continuation a′w′ and take L(E) = {x′}. Then each continuation aw has only emissions xs with x = x′. For each violation of one of the remaining three conditions, it should be clear from the considerations in the if part that either out-arc of the fork is

the begin of some continuation whose emission starts with the keyword k mentioned in those conditions. □

Now the time has come to apply this result to keyword grammars. First, we extend the determined-choice property from keyword graphs to keyword grammars in an obvious way: a keyword grammar has the determined-choice property iff the keyword graph derived from it has that property.

We are interested in relating this property to the constructor set of keyword grammars. Therefore, we define the notion "to warrant determined choice":

A set of constructors C warrants determined choice iff all keyword grammars having C as their constructor set have the determined-choice property.

Finally we define a uniqueness condition for sets of constructors, expressed in terms of such sets themselves, without reference to grammars:

A set of constructors satisfies the uniqueness condition iff the first keyword of each constructor is distinct from the remaining keywords of that constructor and from all keywords of other constructors.

THEOREM 2. Let C be a set of constructors. C warrants determined choice iff C satisfies the uniqueness condition.

Proof. (If part) Let G be a keyword grammar having C as its constructor set, and consider the keyword graph H derived from G. We have to check the properties (i) through (iv) of the determined-choice property for H. We recall from section 2 the three possible types of forks for derived keyword graphs, and notice that never is an out-arc of a fork labelled with E. This gives property (i). If two out-arcs are labelled with keywords, then the fork is a source, so the uniqueness condition implies (ii). If two out-arcs are labelled with statement types S_1 and S_2, the S_1 and S_2 must be distinct elements of a set of statement types. So property (iii) follows from the uniqueness condition too. Finally (iv) is immediate.

(Only-if part) Let C be a set of constructors violating the uniqueness condition. This means that the first keywords of C are not distinct, or that a first keyword of C also occurs at another position. For either case, we construct a counter-example keyword grammar that shows that C does not warrant determined choice. As for the first case, this is rather trivial. Let $k_1\ k_2\ \ldots\ k_m$ and $k_1\ k_2'\ \ldots\ k_n'$ be the constructors of C causing non-

uniqueness, and take any grammar G containing the productions

$$S \rightarrow k_1\ k_2\ \ldots\ k_m \text{ and}$$
$$S \rightarrow k_1\ k_2'\ \ldots\ k_n'.$$

As for the second case, let the constructors of C causing non-uniqueness be $k_1 \ldots k_m$ and $k_1' \ldots k_{i-1}'\ k_1\ k_{i+1}' \ldots k_n'$, $i \geq 2$. Take any grammar G containing the productions

$$S \rightarrow k_1 \ldots k_m \text{ and}$$
$$S \rightarrow k_1' \ldots k_{i-1}'\ \{S\}^*\ k_1\ k_{i+1}' \ldots k_n'.$$

(In the special case where $k_1 \ldots k_m = k_1' \ldots k_n'$, only the second of these two productions should be taken.) □

4. FINAL REMARKS

(a) The uniqueness condition depends solely on the constructor set. It is easy to state, and checking this condition is much simpler than, e.g., that for LL(1)-ness. A compiler for an extensible keyword language can incrementally check the uniqueness condition as new statement types are added.

(b) The reason for the requirements R1′ and R3′ in the definition of "keyword grammar" is the following: if one of these requirements is not met, the grammar would not warrant determined choice, regardless of the constructor set! (Consider, e.g., $S \rightarrow \{S\}^*\ k_1 \ldots k_n$ and $S \rightarrow k_1 \ldots k_n\ \{S\}^*\ \{S\}$.) As for R2′, if two expressions were allowed to follow each other, this would give problems for expression languages containing, e.g., "I", "- 1" and "I - 1", or "P", "(K + 1)" and "P (K + 1)".

(c) If some string s is produced by a keyword grammar G satisfying the uniqueness condition, then each attempt to re-produce that string by a recursive walk over the keyword graph derived from G proceeds by forced choice and ends on a sink. From that sink no out-arcs emerge, so no string st, with non-empty t, could also be produced by G. This is known as the "prefix property". The relevance of this property has already been explained in the introduction. We have cheated a little, however, since again L(E) intervenes. Indeed, it is quite possible that, e.g., both "X" and "X + 1" are expressions. If E is considered terminal, the prefix property holds without more.

(d) Not all keyword graphs are derived from keyword grammars. Some simple transformations are possible on keyword graphs that do not influence the strings emitted. We will not go into details, but it should be clear that the keyword graph in figure 1 may be obtained in this way from the one in figure 2. A simpler parser results, with iteration partially replacing recursion.
(e) Because of the determined-choice property, errors may be caught at the first erroneous symbol (with again some reservation where L(E) is concerned). Together with the fact that each transition to a new line marks a new statement, this gives superior error detection and recovery.

REFERENCES

[1] Leo Geurts & Lambert Meertens,
Designing a beginners' programming language, in
New Directions in Algorithmic Languages - 1975,
S.A. Schuman (ed.), IRIA, Rocquencourt, 1976.

[2] Donald E. Knuth,
Top-Down Syntax Analysis,
Acta Informatica 1 (1971) 79-110.

[3] A.J. Korenjak & J.E. Hopcroft,
Simple deterministic languages,
Proc. IEEE Symp. Switching and Automata Theory 7 (1966) 36-46.

[4] C.H.A. Koster,
Syntax-directed parsing of ALGOL 68 programs, in
Proceedings of Informal Conference of ALGOL 68 Implementation,
U.B.C., Vancouver B.C., August 1969.