ON THE USEFULNESS OF DETERMINISTIC GRAMMARS FOR SIMULATION LANGUAGES

Antonio P. Bongulielmi
Contraves AG
Schaffhauserstr. 580
CH-8052 Zurich
Switzerland

Francois E. Cellier
Institute for Automatic Control
The Swiss Federal Institute
of Technology Zurich
ETH - Zentrum
CH-8092 Zurich
Switzerland

ABSTRACT

The aim of this paper is to give some indications on how future simulation languages should be structured to guarantee a high degree of software robustness. For this purpose, we concentrated on the particular aspect of classifying languages into several groups out of which we recommend the group of the LL(1) languages for the construction of simulation software.

A general purpose parser program is presented which allows to check whether a particular language definition falls into the class of the LL(1) languages, and which can be considered as a very powerful tool for the development of new languages. This parser program has already been used during the design phase of the new simulation language COSY for COmbined (continuous and discrete) SYstem simulation.

In a final chapter we present another program which may be used to generate syntax diagrams out of an EBNF description of a grammar on any plotting device.

I) INTRODUCTION:

Comparing simulation languages to general purpose programming languages, one shall notice that the number of keywords (terminal symbols) of simulation languages is markedly larger than in the case of languages like PASCAL or FORTRAN. On the other hand, "complex" simulation programs will, in general, be shorter and show a simpler program- and data structure as compared to "complex" general purpose programs. The reason for this raises from the fact that, since the language is constructed for a particular application (namely simulation), it can provide for particular constructs which allow to express "complex" situations by means of language elements which have a "simple" syntactical structure but a very "complex" semantic meaning behind. As a result of this, either

a) the simulation compiler will have to expand the text drastically to reduce the high level source code constructions to constructions offered in the general purpose target language (be it any high level language like FORTRAN or be it even Assembly), or

b) the simulation run-time system will have to provide many quite complex procedures to deal with these language elements.

In both cases, it will not be easy to prove that the involved procedures are real algorithms, that is, that they shall run into a halt condition after executing a finite number of elementary steps for any combination of inputs, especially since they must be able to "digest" also incorrect input.

Many simulation systems exist and are widely used (without citing examples!) from which it is known that they (hopefully) handle correct programs correctly, whereas they run either into error modes or into infinite loops as soon as they are asked to "digest" only slightly incorrect input streams. Even worse, they often simply produce incorrect output without notifying the (credulous) user that something is wrong with his problem specification.

The average users of simulation languages are application oriented rather than computer oriented. Their knowledge of the machine and their capabilities of solving emerging troubles (e.g. by means of reading a computer DUMP) is very restricted. It is, therefore, important that these users are supplied with extensive and "verbous" error diagnostics. Such error diagnostics are, of course, not obtained for free. They are, however, feasible for two reasons:

a) Even a "small" simulation program involving only a couple of statements will require much more CPU-time for its execution than the average general purpose program of similar size. Accordingly, we can afford to let the simulation compiler execute somewhat slower by allowing better error analysis to be done than e.g. in the case of a FORTRAN compiler which is supposed to compile also small student programs efficiently.

b) Even a "small" simulation program involves implicitely very complex activities (like numerical integration of a stiff set of differential equations). These must be executed, and, consequently, the simulation run-time system requires much more core memory than the run-time code generated out of the average "small" FORTRAN program. Since it does not make sense to require the simulation compiler to occupy less core memory than the simulation run-time system,

the restrictions concerning the tolerable size of a simulation compiler are also much less stringent than in the case of a general purpose compiler.

Most of the existing simulation systems do definitely not meet the requirements resulting from the above discussion. This results mainly from the fact that simulation systems have, in most cases, been coded by simulation users, that is, by people wich are merely amateurs in information science. Their knowledge of the theories of information science hoperully reaches to (and most probably ends with) the capability of reading and understanding a BNF description of a language which they learnt while being confronted with the famous, and at its time very innovative, but nevertheless very incorrect [12] CSSL-specifications [15]. There exist, of course, exceptions from this general rule!

The aim of this position paper is:

a) to show what restrictions are necessary in the definition of simulation languages to allow for proper error testing and compiler construction, and

b) to present a general purpose parser program which allows to check whether a proposed language belongs to this desired class of languages, and if not, to give indications on how the language definition must be modified to meet these requirements. This program is a very powerful tool to assist in the design phase of a new hypothetical programming language. It has already been used for the design of COSY [5]. This tool is not restricted to simulation languages, but gets even more useful as the degree of complexity of the language to be analysed (~ number of keywords) increases.

II) SOME BASIC DEFINITIONS FROM THE THEORY OF LANGUAGES:

If we wish to specify what attributes a simulation language should have, we must be able to characterize languages, and give a classification. More, if we want to use a program to determine whether a particular language belongs to a particular class of languages, this characterization must be very formal. For this purpose, we need to give some definitions first. An excellent collection of definitions of terms in connection with formal languages can be found in [1]. We shall restrict ourselves to those definitions which we need in our discussion.

Language:
A language over an alphabet T is a set of strings over this alphabet. The alphabet of the language is the set of "atoms" of the language, that is, the set consisting of the keywords (terminal symbols) of that language with the inclusion of identifiers, numbers, comments and strings.

This definition is, however, too vague to be directly applicable.

A language consists of syntax and semantics. In general, the syntax is identified with the conventions of notation of sentences in the language. The semantics are associated with the meaning of these sentences. In practice, however, these two quantities are not sharply distinguishable even in the case of formal languages. The precise border between them is an attribute of the compiler involved rather than of the language itself. Each compiler consists of the following three steps:

a) Lexical analysis (scanning):
This part of the compiler reads character after character from the input stream. Character means here any member of the character set of the machine. Symbols are then composed as strings of characters by the scanner to form the terminals, that is, the elements of the alphabet T of the language. A symbol (member of T) is said to be completed when the scanner detects a separator in the input stream. The set of separators Σ is a subset of the alphabet T of the language ($\Sigma \subset T$) which must be explicitly known to the scanner. In our parser description, we shall give the precise definition of separators as we use it in our analysis. If the order (~ number of elements) of any separator $\sigma_i \in \Sigma$ (denoted as $|\sigma_i|$) is larger than one, the scanner must use some backtracing algorithm to determine separators in the processing of the input stream.

b) Syntactical analysis (parsing):
This part of the compiler treats the members of the set T as "atoms", and checks whether specified sentences belong to the language or not. During the scanning procedure, all combinations of characters which cannot belong to T have already been rejected. Thus, the syntactical analyser will have to analyse only programs which belong to the closure set T^* of set T. The closure of a set is defined as the set whose members are any strings of members of the original set including the empty string ε. The duty of the parsing algorithm is to determine whether a particular program belongs to the subset L of the closure T^* ($L \subset T^*$) which forms the language. Since the order of the set T^* and even of the set L is usually infinite, L cannot be specified by noting down all correct programs (which are the members of the set L). One uses instead a grammar which we shall define below.

c) Semantic analysis (translating):
This part of the compiler maps sentences specified in the source language into sentences specified in the target language. It is obvious that it is to a certain degree left up to us to decide how much work is to be accomplished by the parser, and how much is detained to the translator step.

Grammar:
A grammar is a formal description of the rules which distinguish those members of T^* belonging to L from those which do not belong to L. Since this cannot be done by enumeration, we use a set of productions (P) describing the rules how to form correct sentences of the language. This description is often done re-

cursively which is even necessary if we want to describe an infinite number of members of L by a finite number of productions.

Example:

```
UNSIGNED_INTEGER = DIGIT | UNSIGNED_INTEGER DIGIT .
```

reading: An unsigned integer is either a digit or an unsigned integer followed by a digit.

For the specification of productions, we use a version of BNF which has been accepted as a standard by TC3 of IMACS. A formal definition is given in Appendix I.

The single production specified above describes (by recursion) the countable but infinite number of unsigned integers. In reality, the range of allowable integers is finite for any machine, but the required range test to determine whether a particular integer belongs to the allowable set is usually detained to the semantic analysis.

Would the allowed range be e.g. from 0 to 999, we could alternatively rewrite the production as:

```
UNSIGNED_INTEGER = DIGIT [ DIGIT [ DIGIT ] ] .
```

in which case the range test would belong to the syntactical rather than to the semantical description of the compiler.

The identifiers UNSIGNED_INTEGER and DIGIT do not belong to the alphabet of the language. They are auxiliary variables to denote productions. They form another set which we call the set of the nonterminal symbols (N).

To parse a particular program, we need a start symbol (s) which is a member of N to get the parsing algorithm started.

Thus, the grammar G can be formally described as a 4-tuple:

$$G = (N, T, s, P) .$$

For reasons of simplicity, we shall usually omit the explicit mentioning of the sets N and T, and replace this by the simple rule that terminal symbols are enclosed in apostrophy ('), whereas nonterminals are not.

In the original BNF notation, the opposite convention has been used (nonterminals were embraced by pairs of the meta-symbols '<' and '>'.) This is, however, bothersome since, in this case, we cannot distinguish between the meta-symbols of the BNF notation and the terminal symbols which automatically excludes all meta-symbols from the allowable set of keywords of the language. For this (and only for this) reason, assignments in the original BNF notation used the meta-symbol '::=' in the hope that no reasonable language would want to make use of this -- rather exotic -- combination of special characters as or within a keyword (!). In our redefinition, the set of meta-symbols must be disjoint from the set N which is, however, uncritical since the members of N denote auxiliary variables which can take any name.

For the starting symbol s, one can often find one of the following two determination rules:

a) s is the nonterminal which is defined by the first production of the production set (which implies an ordering relationship among the members of P), or

b) s is the only nonterminal which is defined but never referenced (which inhibits the definition of additional productions for documentation reasons).

We have chosen another definition:

c) s is the only symbol which belongs to both N and T. Each program belonging to L starts with s, and s does not appear anywhere else in the program.

We normally utilize the symbol PROGRAM for that purpose which is common use in many languages of today. Since our aim is to obtain a vehicle for the definition of new languages rather than to analyse existing languages (which are often anyway hard to identify with any senseful characterizable class of languages), this definition seems appealing.

Context-free grammar (CFG):
CFG's denote grammars in which each nonterminal is explicitely defined, that is, its definition does not depend on the environment in which it is used.

Context-free language (CFL):
A CFL is a language for which it is possible to derive a CFG. Note: Given a context-sensitive grammar (CSG) for a language, it cannot be concluded that the language is also a context-sensitive language (CSL). It may be, that a CFG exists for the same language, making it a CFL.

Deterministic grammar (DG):
The set of DG's is a true subset of the set of CFG's for which it is possible to identify each member of L by applying the rules as expressed by the productions in a unique manner (the program has then a unique syntax tree).

Deterministic language (DL):
A DL is a language for which it is possible to write down a DG.

LL(1) grammar:
LL(1) grammars are a true subset of the DG's for which it is possible to obtain a parsing algorithm which is able to determine the syntax tree by scanning the input file from left to right while looking only one symbol ahead.

LL(1) language:
A LL(1) language is a DL for which it is possible to give a LL(1) grammar.

LL(1) parser:
A LL(1) parser is an algorithm which determines the syntax tree of a LL(1) grammar in the above way.

These definitions are useful since

a) with each subset it is possible to derive simpler and faster parsing algorithms, and

b) the "documentability" of a language increases with each subset. CSL's are extremely difficult to describe whereas DL's with a LL(1) grammar are easily documented by explaining the syntactical rules by means of syntax diagrams [13] which are equivalent to a BNF notation, but which are much easier readable for human beings. Such a documentation is very important in the case of simulation languages since such languages usually involve a large number of productions to be explained.

The question must be raised what price we have to pay for restricting ourselves to LL(1) languages. It is obvious that, since there exist languages which are not of type LL(1), we reduce our freedom in how to design our new simulation language. However, even LL(1) languages are very powerful and allow structures to be defined which are by far more complex than needed by simulation languages. Moreover, most of the required restrictions are restrictions of the way in which the grammar is expressed rather than restrictions of the language itself. Most modern formal languages (like PASCAL, ALGOL, SIMULA-67), and even CSSL-type languages (like CSMP) are close to LL(1) parsibility. There exists just one simulation language concept used in many current simulation software systems which is not easily LL(1) expressible, namely the MACRO facility.

Currently, there is a project going on concerning the development of a translator from CSMP-III to COSY [3, 5]. For this purpose, we have derived a formal description of CSMP-III. This is, of course, achieved by use of the general purpose parser program. As a side effect of this project, we obtained a formal description of CSMP-III in EBNF notation and by use of syntax diagrams. Furthermore, we obtained an analysis of the LL(1) parsibility of CSMP-III.

In another report, we may

a) present the formal description of CSMP,

b) discuss incompatibilities of the CSMP language with the concept of LL(1) languages, and

c) give some suggestions on how these incompatibilities could be overcome by as slight modifications of the language as possible. Of course, this modified version of CSMP would not be LL(1) parsible either since we have not in mind to suggest modifications of the underlying FORTRAN as well. FORTRAN is not at all LL(1) parsible which can be easily seen from the two correct(!) FORTRAN statements:

```
i)  DO 10 I=1,21

ii) DO 10 I=1.21
```

out of which (i) is a DO-loop specification statement, whereas (ii) is an assignment statement assigning a value of 1.21 to the variable DO10I.

Since LL(1) grammars allow efficient parsing algorithms to be coded and extensive error analysis to be carried out, we shall restrict our view to this class of languages, and suggest to SWISSL to recommend that future simulation languages should be of this type whenever feasible.

Recursions:
We have already seen that recursions are in a way inavoidable if a finite set of productions is to describe an infinite set of programs of a language. We can, however, often reduce recursions to "repetitions" using the curly braces (cf. Appendix I).

Unfortunately, LL(1) parsers are unable to handle left-recursions. We, therefore, must forbid them. For this reason, the syntax expressed for UNSIGNED_INTEGER is illegal. It can, however, easily be rewritten as:

```
UNSIGNED_INTEGER = { DIGIT } .
```

This additional rule does not imply a restriction in generality. It can be shown that any left-recursion can be reduced by means of a simple algorithm described in [1] (->Greibach Normal Form).

Ambiguities:
DL's are never ambiguous. It is, however, often not easy to decide whether an explicitely given grammar is ambiguous or not. As a matter of fact, it has even been shown that the question whether a particular CFG is ambiguous is not decidable (that is, it is impossible to write an algorithm answering this question for all CFG's) [1]. However, using LL(1) parsers, it is possible to decide whether a grammar is of type LL(1). This is not contradictory to the above statement since LL(1) grammars are a true subset of the set of CFG's, and since the proof concerning indecidability has only been given for all CFG's but not for subsets of them (there exist unambiguous CFG's which are not even deterministic). To test for LL(1) parsibility, the following two rules must be applied to all branching points (b.p.) of the syntax diagrams:

RULE 1: The starting symbols of all parallel branches must be disjoint. This set is called FIRST(b.p.).

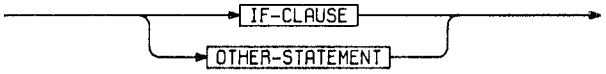
RULE 2: If FIRST(b.p.) contains the empty symbol (ε), also FIRST(b.p.') must be disjoint from FIRST(b.p.) where b.p.' denotes the branching point to which the empty path leads. If the empty path leads to the end of the production, FIRST(b.p.') is identical with FOLLOW(p) where FOLLOW(p) denotes the set of symbols which may follow the production p immediately.

If these two rules hold for all b.p., the grammar of the CFL is of type LL(1), and thus deterministic, and thus unambiguous.

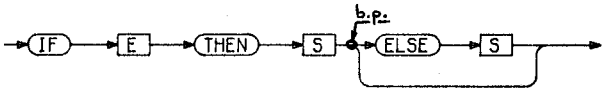
Example:

```
IF_CLAUSE = 'IF' E 'THEN' S [ 'ELSE' S ] .
S = IF_CLAUSE | OTHER_STATEMENT .
```

S



IF-CLAUSE



This grammar looks okay, but it is not. RULE 2 does not hold for the b.p. preceding the ELSE.

Proof: (a) FIRST(b.p.) = (ELSE,ε).
(b) The empty path branches to the exit. Therefore: FIRST(b.p.') = FOLLOW(IF_CLAUSE) = FOLLOW(S).
(c) S appears in IF_CLAUSE behind THEN. The symbols following S immediately contain ELSE and ε. Thus, ELSE belongs to the set FOLLOW(S), and RULE 2 does not hold since FOLLOW(S) is not disjoint from FIRST(b.p.).

q.e.d.

Indeed, this specifies an ambiguity which is well known and which can easily be seen when trying to understand the "correct" sentence:

IF e1 THEN IF e2 THEN s1 ELSE s2 ,

from which it is not clear to which IF_CLAUSE the ELSE must be bound. An algorithm testing the above two rules shall tell us that there exists a conflict of FIRST's at b.p. involving symbol ELSE.

In [1] a "solution" to this problem is given. The two productions stated above are replaced by a set of four productions of the form:

```
S1 = IF_CLAUSE1 | OTHER STATEMENT .
S2 = IF_CLAUSE2 | OTHER_STATEMENT .
IF_CLAUSE1 = 'IF' E 'THEN' S1 |
             'IF' E 'THEN' S2 'ELSE' S1 .
IF_CLAUSE2 = 'IF' E 'THEN' S2 'ELSE' S2 .
```

This grammar is a DG which binds the ELSE always to the closest IF_CLAUSE. Unfortunately, this DG is <u>not</u> of type LL(1). In production 3 there exist two parallel branches starting with the same symbol IF. Grouping leads to:

```
IF_CLAUSE1 = 'IF' E 'THEN' ( S1 | S2 'ELSE' S1 ) .
```

However, this does not solve the problem either since the two nonterminals S1 and S2 have all starting symbols in common. The proper solution would be to modify the language to:

```
S = IF_CLAUSE | OTHER_STATEMENT .
IF_CLAUSE = 'IF' E 'THEN' 'BEGIN' S 'END'
            [ 'ELSE' S ] .
```

which is a grammar of type LL(1). This solution shall also read better since we do not need an additional explanation to state how the ambiguous sentence is to be understood since:

a) IF e1 THEN BEGIN IF e2 THEN s1 ELSE s2 END

b) IF e1 THEN BEGIN IF e2 THEN s1 END ELSE s2

are both clear.

The algorithm which checks for ambiguities has, thus, helped to get rid of an ambiguity which is otherwise difficult to detect.

A recursive parser would be able to "solve" the above type of ambiguity also directly by the simple rule that it does not leave the level of recursion except if the symbol read from the input stream is not in the group of the FIRST's. Thus, a recursive parser would automatically bind the ELSE to the nearest IF_CLAUSE, as this is a common rule in several modern languages (e.g. PASCAL). This mechanism is better explained in the description of our parser program. It is, nevertheless, useful to have an algorithm which detects such ambiguities since they require at least a special explanation if we agree to allow them at all.

III) ROBUSTNESS:

The aim of this discussion is to show that restricting ourselves to LL(1) grammars can help us to obtain more robust simulation systems. The term "robustness" has, however, more of a slogan than of a well defined term. It is, therefore, necessary to give an explanation on how we are going to use it. A simulation system has three distinct parts which can all be "robust" in different sences:

a) The <u>simulation language</u> can be robust with respect to
   1) modeling
   2) programming

b) The <u>simulation compiler</u> can be robust with respect to
   1) programming
   2) maintainability
   3) implementability (portability)

c) The <u>simulation run-time system</u> can be robust with respect to
   1) procedures
   2) algorithms.

The robustness of simulation run-time systems has been discussed in [6]. It is, moreover, not necessary to repeat those results here since the use of LL(1) grammars has very little to do with these aspects of robustness. They are mentioned here only for reasons of completeness.

III.1) ROBUSTNESS OF SIMULATION LANGUAGES:

Aspect of Modeling:
The use of LL(1) grammars can help to improve the modeling capabilities of a language in two ways:

a) No ambiguities:
The fact that LL(1) languages are guaranteed not ambiguous will certainly improve the safety of modeling.

b) Documentability:
As previously stated, syntax diagrams can help to explain the syntactical rules of a language. The semantic meaning of each production will usually be assigned to the syntactic explanation in an informal manner. LL(1) grammars are especially well suited to be described in this way. Since nonterminals can be freely used in the definition of productions, it is even possible to achieve a hierarchical documentation by these means. In this way, it is, for instance, possible to explain first the basic structures of the language without going into details as for the programming of the single blocks. In further chapters of the documentation volume, these blocks can then be described in depth. Several simulation languages have already been documented by use of syntax diagrams (e.g. COSY [3], SCALE-F [8]). This aspect, therefore, can improve both the documentability of the software from the point of view of the software engineer who develops the language, in that it can be assured that the documentation is complete and easily updatable, as well as the understandability of the documentation from a user's point of view. SYSMOD [2] goes even one step further, in that its documentation volume is written such that the text editor can extract from it automatically all productions making up the grammar of SYSMOD. In this way, the input to our parser program is generated automatically from the user's manual. Syntax diagrams can thereafter be produced automatically which again form part of the user's manual. All modifications of the language therefore start by a modification of its manual which is a very nice feature.

Aspect of Programming:
It is extremely important that errors in the source code of a simulation program are detected as early as possible by the simulation compiler, and that a precise diagnostic of the error is reported to the user. LL(1) languages shall allow such an error processing to a very high extent as we shall show in:

III.2) ROBUSTNESS OF SIMULATION COMPILERS:

Aspect of Programming:
LL(1) grammars are especially well suited to obtain robust simulation compilers in the sense that such a compiler can properly digest any input stream be it as incorrect as it wants. This is true for several reasons:

a) LL(1) parsibility:
Since the syntax tree for any correct program of the language can be obtained by parsing the input from left to right while looking only one symbol ahead, it is evident that a misplaced symbol must be detected at once. It can never happen that a syntactically incorrect program is accepted by the compiler resulting in an incorrect output which may be accepted as correct by the credulous user.

b) Recovering from errors:
Once an error has been detected by the parser, the compiler must try to "recover" as quickly as possible to become able to recognize further errors in the input stream. This error resynchronization cannot be done in a fully systematic way. The language designer must assume something about the probability of expected errors. However, LL(1) grammars allow to make this procedure somewhat more systematic since the language designer can augment his language definition by additional error paths which denote what actions the compiler has to take when particular types of errors occur. These error paths can already be introduced into the grammar before the compiler is written and allow to experiment with different kinds of error handling before the first line of the compiler is actually coded.

c) Cleanness of programming:
Compilers which are constructed on the basis of LL(1) parsers have an almost linear top-down structure. It is, therefore, possible to code such compilers in a way which guarantees that no input can "hang up" the compiler and which guarantees that the compiler is not entering infinite loops for any input, be it as malign as it wants. In other words: It is (at least theoretically) possible to guarantee that the compiler is an algorithm.

Aspect of Maintainability:
It is unavoidable that a complex program like a compiler has some "bugs" in it which are not detected until somebody stumbles upon them by chance. At that time, it is well likely that the programmer of the compiler has left the place already, and is no longer accessible. In such a situation, it is extremely important that somebody else is able to read and understand the compiler to be able to remove the bug. It is then very cumbersome if the software engineer is forced to read and understand the compiler as a whole. In most cases, it is not too difficult to identify and isolate the bug within the compiler. A local patch, however, bears the risk of unexpected side effects creating new bugs which are often worse than the removed one! For this purpose, it is important that the compiler has as much as possible a linear top-down structure since side effects result mostly from GOTO-statements pointing backward from below to beyond the patch position. If the effect of such a GOTO-statement is not taken into account, the patch creates often troubles which are difficult to explain and to correct. Since LL(1) grammars allow compilers to be written in an almost direct top-down

structure, the robustness of such a compiler with respect to its maintainability is markedly better than in the case of other grammars being used. It is, of course, furthermore very important to code the compiler in an appropriate language. FORTRAN is an atrocious language for this task. PASCAL proved very satisfactory as long as the language definition is not too large (PASCAL is very difficult to overlay), and as long as the language to be compiled does not involve too much file handling. Another good candidate for coding compilers for simulation languages is SIMULA-67 which, being a simulation language in itself, has even nicer features as a general purpose language than as a simulation language. Another good candidate might be ADA.

Aspect of Implementability (Portability):
It is neatly shown in [14] that coding a compiler for a LL(1) language is relatively straight forward. By use of PASCAL, this task can be achieved by utilizing standard features of PASCAL almost exclusively. Although FORTRAN programs are in general better portable (and thus implementable on another machine) than programs coded in any other language, this does not really hold for compiler programs since the string handling is not easily coded in somewhat like "standard" FORTRAN in that for instance the number of characters which can be packed into a word is extremely system dependent. There exist FORTRAN-coded compilers for simulation languages which can be truely called "portable" (e.g. DARE-P [11]), but coding such a compiler is really an art, and we do not know of any FORTRAN-coded compiler for any simulation language which guarantees for detection of all syntactical errors. For this reason, a carefully coded compiler on a PASCAL-, SIMULA-67 or ADA-basis shall most probably be superior with respect to portability. ADA seems really to be the best way to go in the future, as its strict formalization together with its environment definition shall grant a new and previously unattainable degree of portability.

IV) A GENERAL PURPOSE PARSER PROGRAM:

IV.1) INTRODUCTION:

The general purpose parser program which will be described hereafter has been developed on the basis of an idea expressed in [14]. It allows those who want to develop new languages to check both correctness and usefulness of the language under construction in a simple and efficient way. The parser can properly "digest" any CFG. While processing the input stream, the parser goes through the following three steps:

a) It reads in the syntactic definition of the language being expressed by use of the EBNF notation (as specified in Appendix I).

b) It checks whether the processed grammar belongs to the subclass of the LL(1) grammars. It checks especially if the grammar is free of left recursions and ambiguities.

c) It processes any number of sentences specified in the previously defined language and checks whether they belong to the language or not (if all the tests executed previously under (b) were positive!).

It is, thus, possible to test any language definition, be it as complex as it wants, automatically, to decide whether it belongs to the class of the LL(1) grammars (and is thus LL(1) parsible) which is a very difficult and tedious task to perform by hand.

As an additional advantage of using the parser program, we consider its ability to parse -- already during the design phase of the language -- even very large sample programs (sentences) for their correctness. With the aid of the parser, it is, therefore, possible to determine how easily problems can be specified in the hypothetical new language, before the first line of the compiler is coded.

The parser can also help in the documentation of the language since it can generate a good deal of information concerning the newly defined language which is directly usable when writing the user's manual for it. This output furthermore includes information which can help to improve the language definition, and it also includes information which is useful for the generation of the compiler for that language.

Beside of the basic rules of LL(1) grammars, the parser will require the observation of two additional rules:

a) From none of the branching points there may emerge more than one empty path.

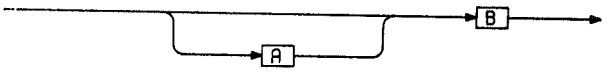
b) If an empty path emerges from a branching point, it must be specified as the last alternative.

These two additional rules may first seem to be further restricting the set of definable languages. This is, however, not the case. It does restrict not the expressable grammars but only their way of notation. Any LL(1) grammar can be brought into a form acceptable by the parser program.

Proof:
In the EBNF notation as described in Appendix I, empty paths cannot be noted down directly. They are codable only by use of the square brackets (option) or by use of the curly braces (repetition). These two cases shall be discussed separately.

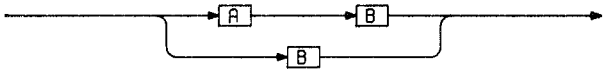
Empty paths resulting from options:
In a syntax diagram, empty paths resulting from options appear as alternatives to non-empty paths. Due to the rules of LL(1) parsibility, it is obvious that all such empty paths which branch to a point within the syntax diagram can be eliminated by simply duplicating the structure which can be reached via the empty path. This is illustrated in the following syntax diagram (INNER_OPTION):

INNER-OPTION



which is reducible to:

INNER-OPTION



For this reason, we can restrict our view to the "true" empty paths which are those alternatives branching to the end of the production. Since each production has only one end, there can never exist several such empty paths leaving from the same b.p..

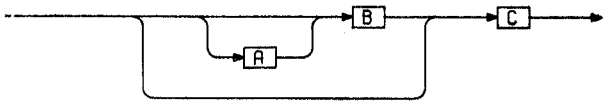
q.e.d.

The following example shall illustrate the reduction technique:

EXAMPLE = [[A] B] C .

with: $\varepsilon \in (A \cup B \cup C)$;
$FIRST(A) \cap FIRST(B) = \Phi$;
$FIRST(A) \cap FIRST(C) = \Phi$.

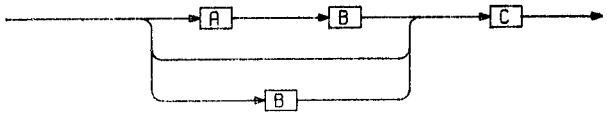
EXAMPLE



Applying the reduction algorithm leads to:

EXAMPLE = ((A B) | [B]) C .

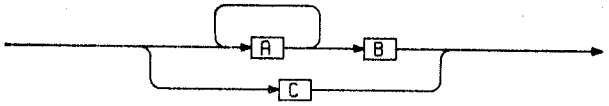
EXAMPLE



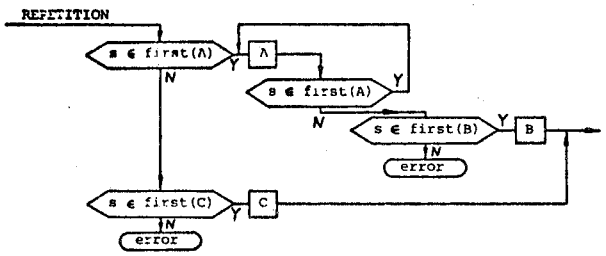
Empty paths resulting from repetitions:
Another way of creating empty paths is by use of repetitions. Such empty paths are not easily reducible. We shall show that the parser can process multiple empty paths resulting from repetitions. So, there exists no urgent need for reducibility. Given the following production:

REPETITION = ({ A } B) | C .

REPETITION



The way how the parser processes such a structure can be made better understandable by augmenting the syntax diagram for this production by some tests as they are performed by the parser program.



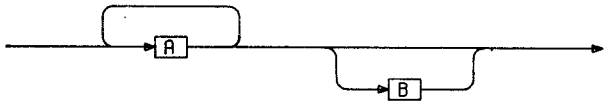
From this modified syntax diagram it is evident that the parser does not look at FIRST(B) before it is sure that the symbol read from the input file does not belong to FIRST(A). FIRST(A) and FIRST(B) even need not be disjoint in this case. However, those members of FIRST(B) which also belong to FIRST(A) will never be addressable, and are thus redundant.

According to this discussion, there exists always a priority scheme for empty paths resulting from repetitions which allows to process the syntactical structure without need for any backtracing. This priority scheme is directly visible from the syntax diagrams drawn by the program discussed in the next chapter. All paths may only be followed in their "correct" direction even if there exists nothing but a connecting line between "two" b.p..

This shall be illustrated by the following production:

REP_AND_OPT = { A } [B] .

REP-AND-OPT



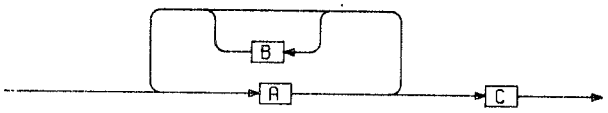
On a first look, it seems that there emerge two

empty paths from the same b.p.. However, since the first empty path results from a repetition, one can split the b.p. into two. While processing a sentence using this rule, the parser will repeat A as many times as possible before it starts looking at B. Only at this moment, the second empty path may become active if the symbol read from the input file does not belong to FIRST(B) either.

Complications may result if both ways of generating empty paths appear in a combination. This can be illustrated by the following production (PROD):

```
PROD = { A $ [ B ] } C .
```

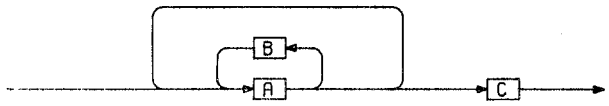
PROD



This production could e.g. be used to denote a list whose elements are optionally separated by ',' (B=','), and which is terminated by ';' (C=';'). The syntax diagram is, however, incorrect since -- according to our rules -- the parser remains in the repetition as long as possible. Since the empty path is a member of the possible alternatives for starting the repetition, the parser can never proceed to C. Even such a situation is, however, properly codable by use of a simple transformation:

```
PROD = { { A $ B } } C .
```

PROD



The two additional rules are necessary since the algorithm inherent in the parser program has no built in backtracing capabilities. It checks for alternatives until it finds a match or until it finds an empty path. As soon as an empty path has been found, it follows this path to its ending node, and checks for further matches there. As a direct consequence, alternatives which follow an empty path can never be accessed.

Another consequence of this algorithm -- together with its capability of being used recursively -- is that ambiguities as those treated in the previous chapter are automatically resolved by the parser in a unique and well defined manner. The parser searches for matches at one level of recursion at a time. Only when no match could be found, the parser decreases the level of recursion.

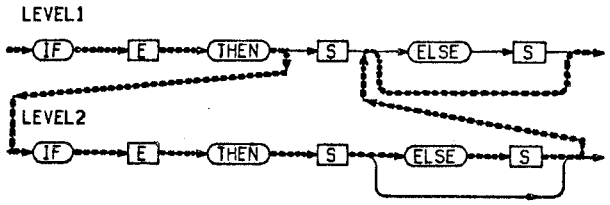
As an example let us consider once more the set of productions:

```
IF_CLAUSE = 'IF' E 'THEN' S [ 'ELSE' S ] .
S = IF_CLAUSE | OTHER_STATEMENT .
```

The sentence:

```
IF e1 THEN IF e2 THEN s1 ELSE s2
```

is treated by the parser in the following manner:



As can be seen from this diagram, the ELSE binds to the nearest IF_CLAUSE.

IV.2) CONCEPTS OF REALIZATION:

During the realization of the general purpose parser program, the following considerations have been especially taken into account:

- The parser should be as general as possible.
- The parser should be as "safe" as possible (no "hang up's" when confronted with incorrect input).
- The parser should be easily usable and flexible.
- The parser should be able to process sample programs in as much as possible the same way as a compiler would do.
- The parser should be as portable as possible.

It must be admitted that these demands are meant as goals which are to be reached as completely as possible, but which are never fully achievable. The following presentation of the parser program shows what solutions have been found to approach these goals.

IV.3) PREDEFINED SYMBOLS:

The description elements COMMENT, IDENTIFIER, STRING, and UNSIGNED_INTEGER which are used in almost any language definition have been predefined. They may be used in any language definition by treating them as terminals. By these means, the language definition can be shortened, and a notable reduction of computing time results. This predefinition does not reduce the generality of the parser since these productions may be user redefined which then automatically overwrites the predefinition.

IV.4) CONTROL OPTIONS:

There exist a large number of control options by the help of which the user can control execution of the

parsing program. A complete syntax diagram for these control options is given in Appendix IV. They may be classified into the following groups:

- Listoption:
  This block of options allows to control amount and appearance of the output produced by the parser program.

- Setoption:
  This block of options allows to control what tests are performed by the parser (out of those tests which are not indispensable for the proper functioning of the parser program, tests which mainly check for LL(1) parsibility of the grammar).

- Singlestep:
  This block allows to define whether the parser is supposed to operate symbol by symbol (default) or character by character. The single step mode executes somewhat slower, but it enables the user to specify the syntactic rules of how identifiers are composed.

- Uppercase:
  This block allows to convert lower case letters to upper case letters. Many languages (and line printers!) do not make use of the full ASCII character set. By use of this options, the user can specify his wish to let lower- and upper case letters be undistinguishable.

- Metasymbols:
  This block of options allows to redefine the meta-symbols of the EBNF notation. This is useful since some of the default symbols (e.g. the vertical line to denote alternatives) may not be printable on the available line printer.

- Redefinition:
  This block allows to redefine the names of the predefined productions to make the standard names available for other purposes without overwriting the predefinition.

- Maxvalue:
  This block allows to reformat the output (e.g. page length) and to respecify the tolerated length of nonterminals.

- Syndia:
  This block of options allows to determine and reformat the plot output to be produced by the syntax diagram drawing program.

IV.5) HANDLING OF SYNTAX ERRORS:

There exist two different mechanisms of error detection and handling in the program which must be distinguished.

While processing the language definition block, the parser program knows the "language" in which specifications are written (being the EBNF notation). The parser can, thus, resynchronize after error detection. This means that the parser program acts here like a special purpose compiler for the meta-language EBNF.

While processing sample programs (sentences) written in the previously defined language, a resynchronization after error detection cannot be automatically performed since the structure of the language is not known in advance but has been built up dynamically during the language definition phase. For this reason, the parsing mechanism stops after the first error being detected, and the input text is skipped up to the beginning of the next sample program. In the future, it is planned to augment the EBNF notation by a mechanism to enable the user to specify actions to be taken in case of an error being detected. This will allow to simulate the error handling behaviour of a special purpose compiler for that language to a very high extent.

IV.6) PROCESSING SYNTAX DEFINITIONS:

The tests which are performed by the parser on a syntactic definition of a language can be divided into two different categories.

The first category contains those meachanisms which are indispensable for a secure handling of the syntax by the parsing algorithm. These tests are, therefore, compulsory. They include:

- Multiple definition of nonterminal symbols (consistency test).

- Missing definition of nonterminals (completeness test).

- Ambiguities resulting from the intermixed use of predefined productions (by treating them as terminals) with a redefinition of those symbols.

- Ambiguities resulting from unfavorable choices of COMMENT and STRING delimiters. (Problems could occur if e.g. the end delimiter of COMMENT is identical with the opening delimiter of string, etc..)

- Existence of left recursions.

These tests are performed by the parser under all circumstances, and the program terminates hereafter when one of the tests fails.

The second category contains the additional tests concerning LL(1) parsibility. These tests contain:

- Looking for b.p. at which several alternatives start with the same terminal symbol.

- Looking for b.p. with several parallel empty paths.

- Looking for empty paths which are not defined as the last alternative.

- Test whether conflicting FIRST's are disjoint.

This second category of tests is performed only if

the user has specified (through the control options) that he wants the FIRST's, the FOLLOW's, and the INTERSECTION's of the productions to be evaluated.

The tests are performed on several hierarchical levels to minimize the number of consequence errors to be "detected". By these means, the parser performs all tests on one hierarchical level at a time. If any of them fails, the program terminates, and all further tests are suppressed.

Error messages are never suppressed (independently of the specifications given in the list options). Error messages tend to be "verbous" to guarantee easy interpretation and correction.

IV.7) SEPARATORS:

Let us give first a definition for this term:

> Separators are characters or strings of characters which enable the scanner to compose characters to symbols for the syntactical analysis. The separators form a subset of the set of terminals (T). They must be explicitely known to the scanner.

The set of separators is built up dynamically. It will be different for different languages, and it is even different during the different phases of the parsing procedure. The scanner is adaptive in that it uses the currently active set of separators for the construction of symbols.

During the different operational phases, the following sets of separators are established:

Control option phase:
Only the SPACE is a separator.

Syntax definition phase:
Separators are the SPACE and all EBNF meta-symbols.

Parsing phase:
Separators are the SPACE and all those keywords of the language starting with a special character (neither digit nor letter) which does not appear as well within any other keyword starting with an alphanumeric.

These separator lists are generated automatically by the parser, and they are printed upon request.

If several separators start with the same special character, the rule of the longest match determines the detection of separators.

This rule creates two minor problems which can, however, be solved without restrictions:

a) The choice of the SPACE as general separator implies that keywords may not contain SPACE's. Such keywords can, however, be subdivided into parts not containing any SPACE's which solves the problem.

   Example:

```
GOTO_KEYWORD = 'GOTO' | 'GO TO' .
```

   can be written as:

```
GOTO_KEYWORD = 'GOTO' | ( 'GO' 'TO' ) .
```

b) Strings may not contain string delimiters since this introduces ambiguous meanings. During the syntax definition phase, the problem can be solved in analogy to the previous problem:

   Example:

```
ILLEGAL_KEYWORD = 'THAT'S' .

LEGAL_KEYWORD = 'THAT' ''' 'S' .
```

   During the parsing phase, there cannot be given any standard solution. This problem must be solved on the basis of the definition of the new language.

The adaptive nature of the scanner allows to simulate the behaviour of a special purpose compiler to a high extent since the scanner uses the same list of separators as a compiler for that language would do. Sample programs can, therefore, be coded in precisely the same manner as if the compiler would already be in existance.

IV.8) PROGRAMMING CONCEPTS:

In the development of the parser program, the following three considerations have been assigned a high priority:

- Robustness (no infinite loops or error modes)
- Generality
- Maintainability

These considerations reduce the efficiency of the program to some extent. However, since the parser program is meant to be used during the design of a new language only, and since it will not be used by large numbers of people, efficiency seemed to be of minor importance as compared to the points mentioned above.

The most important contribution to improve the robustness of the parser program are the tests which have been previously discussed. In reality, there exist even further tests to handle also very rare errors correctly.

The generality of the program has been improved by the fact that dimensioned arrays have only been used for data whose dimensions were known in advance and which (according to our judgement) would never require any modification. These arrays can, furthermore, be easily respecified during the implementation of the parser program. In all other cases, the program uses dynamically generated list structures.

The maintainability has been improved by concentrating all constants in the main program, by dividing the program into a large number of relatively small procedures, and by placing all procedures which perform system dependent activities at the beginning of the program together with an extensive description of the system dependencies.

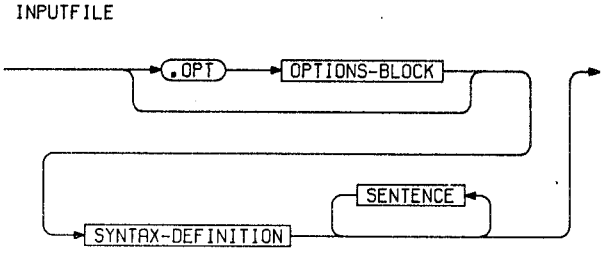
IV.9) PORTABILITY:

To obtain a high degree of portability of the parser program, we used a subset of PASCAL for its realization which is (as to our knowledge) almost system independent.

The basic version was coded for CDC 6000 series installations (or CYBER) running under SCOPE 3.4 and using the local ETH character set (!). This version contains, however, a special comment mechanism which allows to carry out the required (minor) modifications for implementation on other machines by use of a small auxiliary program. Currently, implementations exist for IBM 370 series installations (or IBM 3033) using either the Australian PASCAL compiler or the new IBM compiler, for VAX installations, for DEC-10 installations, for UNIVAC computers, and for PR1ME (which are those computers at our disposal). Installation on a PDP 11/45 running under RT-11 and using the OMSI-PASCAL compiler has also been tried, but the restricted addressable region of a 16 bit machine excludes this implementation from use for larger language definitions.

IV.10) DESCRIPTION OF INPUT DATA:

The entire text which is to be processed by the parser consists of one file which is divided into single text blocks by use of an "end-of-text" marker (default: '$$$$$'). The sequence of text blocks is illustrated by the following syntax diagram:



All blocks allow free-form input. Within the three block types, the following syntactic rules hold:

Control options:
The rules which are to be obeyed in this block are described in Appendix III.

Syntax definition:
The syntactic rules are those of the EBNF notation (cf. Appendix I).

Parsing phase (sentence):
The syntactic rules are those described in the syntax definition specified in the previous block.

IV.11) DESCRIPTION OF OUTPUT DATA:

The general purpose parser has two different output files. One is the so called MESSAGE file which contains a brief description of the execution flow of the program (version and date possibly followed by a reason for abortion). Our CDC implementation directs this MESSAGE file to the "day file" of the job. The second file is the OUTPUT file into which all further output is generated as described below.

1) OPTION ECHO:
If the user has specified a control option block, the specified options are echoed during their processing (with line numbers). The option block is searched for syntactic errors which are reported to the user by self explanatory error messages. The place where the error has been detected is marked, and it is also marked up to which place the text had to be skipped for error resynchronization.

2) SEPARATORS FOR SYNTAX DEFINITION:
The list of all separators which are active during the syntax definition phase are reported to the user. Out of those, the opening and closing delimiters for comments and terminals are specially marked.

3) SYNTAX DEFINITION ECHO:
While the syntax definition is processed, a line numbered echo of the specifications is printed. Two kind of error messages can appear in this part of the analysis. One kind concerns multiple definitions of nonterminals. The other kind concerns EBNF syntax errors. The error message indicates the type of error being detected. The place where the error has been found, and the place up to which the text had to be skipped, are both marked.

4) CROSSREFERENCES OF NONTERMINALS:
This list appears in alphabetical order. For each nonterminal, the list contains the numbers of those lines in which the symbol has been referenced. The reference denoting the definition of the nonterminal is marked by an asterisk (*). Nonterminals which have been defined but never referenced are specially marked. An error message is printed for nonterminals being referenced but not defined.

Following this crossreference table, error messages are printed which denote collisions between nonterminals and predefined productions.

5) CROSSREFERENCES OF TERMINALS:
This contains an alphabetically ordered list of all keywords of the language together with a reference to the numbers of those lines in which they have been met.

6) OUTPUT OF DETECTED LEFT RECURSIONS:
If a left recursion has been detected, the name of the corresponding production is printed together with the sequential path through which the recursion has been defined.

7) ERROR MESSAGES CONCERNING EMPTY PATHS:
These error messages specify that -- while evaluating the FIRST's, FOLLOW's and INTERSECTIONS -- parallel empty paths or empty paths which are not specified as last alternative have been detected.

8) SEQUENTIAL LIST OF FIRST'S:
This list reports in alphabetical order the structure of the set of FIRST's for each production. Each subset is specified separately. Furthermore, the sequence in which the parser looks at these subsets, and the number of empty paths met are indicated. Nonterminals are marked by '>>', empty paths are indicated as '<>'.

9) ALPHABETICAL LIST OF FIRST'S:
This list does not contain any information which could not be extracted from the previous list as well. However, the information appears compressed. This list is, therefore, easier readable, and it is sufficient for most applications. This list which contains an alphabetically ordered set of all FIRST's of a production is of importance for the construction of the special purpose compiler for the language (START and STOP set).

If a production has an empty path which goes through the whole production (empty through path), this is indicated by adding the symbol '<>' to the set of the FIRST's.

If several parallel paths of a production start with the same symbol, an error message is printed with an indication of the multiple used symbols.

Note that the FIRST's are currently only printed for each production, although they are internally evaluated for each b.p..

10) SEQUENTIAL LIST OF FOLLOW'S:
This list has the same structure as output type (8). Nonterminals marked by '->' indicate that, at this place, the sequential FIRST set of the nonterminal should be added. This is done automatically by the parser itself, if the keyword FULL is added to the option FOLLOWSYM SEQUENTIAL.

The representations of type (8) and (10) are very useful for the analysis of INTERSECTIONs between FIRST's and FOLLOW's.

11) ALPHABETICAL LIST OF FOLLOW'S:
This list has the same structure as output type (9). It also helps in the construction of a special purpose compiler.

12) INTERSECTION SET:
This list contains a summary of conflicts with FIRST's at b.p. and FOLLOW's of productions. This output block would e.g. report the conflict concerning the ambiguous meaning of the ELSE construct which has been previously discussed.

13) SEPARATORS FOR PARSING PHASE:
This output block contains the set of the separators which are active during the parsing of sample programs. Out of these, the COMMENT- and STRING delimiters are specially marked.

14) ECHO OF SAMPLE PROGRAMS:
While the parser processes sample programs coded in the previously defined language, it echoes the input stream by adding line numbers. Syntax errors being detected are immediately indicated, and the rest of the sample program is skipped to the next "end-of-text" marker.

15) CROSSREFERENCES OF IDENTIFIERS:
The alphabetically ordered set of the IDENTIFIER's which are collected during the processing of the sample program are listed together with references to the lines in which these IDENTIFIER's have been met. These IDENTIFIER's are the variables of the sample program.

16) MATCHING OF DELIMITERS:
If the number of opening and closing COMMENT- or STRING delimiters does not match, this is reported as an error message to the user as the last of the output blocks.

Remarks:

- The output blocks of type 1 (if options have been specified), and 14 as well as all error messages are printed under all circumstances.
- The output blocks of type 2, 3, 4, 5, 13, and 15 can be suppressed by use of the LISTOPTION specification block.
- The output blocks of type 8, 9, 10, 11, and 12 are printed only if the appropriate SETOPTION specifications have been set by the user. An exception from this rule is output type 9 in that it is possible to ask for evaluation of the involved tests without requiring the list to be printed.
- By default, all output types are printed except for blocks 8, 9, 10, 11, and 12.

An example of a complete language analysis for the language PL/0 [14] is given in Appendix IV.

V) GENERATION OF SYNTAX DIAGRAMS:

The EBNF notation is very useful for the automated processing of a language definition by a computer program since it is easily codable by use of standard computer input file specifications. However, for human beings, this notation tends to be somewhat unreadable, especially because it is not easy to find appropriate matches of parantheses. Human beings, in general, prefer a graphical representation by use of syntax diagrams. For this reason, TC3 of IMACS has accepted both the EBNF notation and its corresponding syntax diagram description as standard language description vehicles. A formal definition for syntax diagram descriptions is given in Appendix II.

Since both representations are equivalent, it is useful to have a computer program which is able to read in the EBNF specification of a language and produce on a plotting device appropriate syntax diagrams out of it.

This has been realized by using a syntax diagram plotting program [4] and a small conversion program which converts our EBNF notation to the form of input specification as it is expected by the syntax diagram plotting program.

Syntax diagrams can be generated for any (x,y) plotting device or for the line printer.

The syntax diagram plotting program produces an input file to a hypothetical high level plotter. A separate plot program interprets this plot file and emulates the hypothetical plotter on the actually available plotting device. Quite obviously, this program has to be system dependent, and needs to be adapted to the available plotter. However, even here a high degree of portability is maintained by restricting the required system dependencies to three procedures to open and close a plot file, and to draw a straight line between two points (pen either up or down). Even the fonts are generated softwarewise to minimize the system dependencies (at the cost of a somewhat reduced efficiency.

VI) PERSPECTIVES FOR DEVELOPMENT:

As we stated in the introduction to chapter III, the general purpose parser program is a valuable tool to aid in the design of new formal languages. It is foreseen to enhance this tool in the future by adding:

a) mechanisms for specification of specially marked error paths which augment the syntax definition of the language, and which will -- if specified -- allow to resynchronize after error detection also during the parsing phase of the program.

b) mechanisms for specification of the static semantics (type attributes, range tests) of the language to enable the parser program to perform precisely the same kind of error handling as a special purpose compiler for that language would do.

By adding these two mechanisms, the general purpose parser can simulate almost all activities of a special purpose parser for the defined language. Error resynchronization shall be possible to perform in precisely the same manner, as would be performed by a special purpose parser.

The next step will be to modify the general purpose parser to let it become a meta-parser. This meta-parser will be able to generate (upon request) a special purpose (non table driven) parser for the specified grammar. This special purpose parser shall then be able to parse sample programs in precisely the same way as the general purpose parser would do, but it will, of course, be much more economical to use. This special parser is the first step in coding a special purpose compiler for the new language.

Lateron we might study the possibilities to add to the language definition also a specification of the dynamic semantics (that is a description of the target language with mapping). Our hope is to be able to finally obtain a meta-compiler which can generate the special purpose compiler for any new language of type LL(1) out of some formal description. Some research projects on this topic have already been carried out. Most promising seems to be the method of attributive grammars which has been suggested by Knuth [10]. Some newer publications on this topic are [7, 9]. Unfortunately, many of the recent relevant publications in this field are written in German, and are, thus, not widely accessible.

REFERENCES:

[1] A.V.Aho, Ullman J.D.: (1972/73) "The Theory of Parsing, Translation, and Compiling -- Volume I: Parsing". Prentice-Hall, Series in Autmatic Computation.

[2] N.J.C.Baker and P.J.Smart: (1982) "The SYSMOD Language and Run Time Facilities Definition". Defensive Weapons Department, Royal Aircraft Establishment, Ministry of Defence, Farnborough Hants, United Kingdom.

[3] A.P.Bongulielmi: (1978) "Definition der allgemeinen Simulationssprache COSY". Semesterwork, Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich. To be obtained on microfiches from: The Main Library, ETH - Zentrum, CH-8092 Zurich, Switzerland. (Mikr. S637).

[4] K.J.Bucher: (1977) "Automatisches Zeichnen von Syntaxdiagrammen, welche in spezieller Backus-Naur-Form gegeben sind: Benuetzeranleitung". To be obtained from: Institute for Informatics, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[5] F.E.Cellier, Bongulielmi A.P.: (1979) "The COSY Simulation Language". Proceedings of the 9th IMACS Congress on Simulation of Systems. Published by North-Holland Publishing Company (Editor: G.Savastano).

[6] F.E.Cellier, Moebius P.J.: (1979) "Towards Robust General Purpose Simulation Software". Proceedings of the ACM SIGNUM Conference on Numerical Ordinary Differential Equations. To be obtained from: Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana IL 61801, U.S.A. (Editor: R.D.Skeel).

[7] H.Ganzinger: (1978) "Optimierende Erzeugung von Uebersetzerteilen aus implementierungsorientierten Sprachbeschreibungen". Ph.D. Thesis, Technical University of Munich, FRG. Form: TUM-INFO-04-78-09-0300/1-FBMA.

[8] D.Heppner: (1977) "Beschreibung der Simulationssprache SCALE/F". To be obtained from: Computing Center, Technical University of Braunschweig, FRG.

[9] U.Kastens: (1976) "Ein uebersetzer-erzeugendes System auf der Basis attributierter Grammatiken". Ph.D. Thesis, University of Karlsruhe, FRG.

[10] D.E.Knuth: (1968) "Semantics of Context-Free Languages". Journal of Mathematical System Theory, no. 2, 1968, pp. 127 - 145.
Corrections:
Journal of Mathematical System Theory, no. 5, 1971, p. 95.

[11] G.A.Korn, Wait J.V.: (1978) "Digital Continuous-System Simulation". Prentice-Hall, Series in Automatic Computation.

[12] T.I.Oren: (1975) "Syntactic Errors of the Original Formal Definition of CSSL 1967". Technical Report: TR75-01 (IEEE Computer Society Repository no. R75-78), Computer Science Dept., University of Ottawa, Ottawa, Canada.

[13] N.Wirth: (1972) "Systematisches Programmieren". Teubner Studienbuecher, Informatik, Vol. 17.

[14] N.Wirth: (1976) "Algorithms + Data Structures = Programs -- Chapter V". Prentice-Hall, Series in Automatic Computation.
or:
N.Wirth: (1977) "Compilerbau". Teubner Studienbuecher, Informatik, vol. 36.

[15] (1967) "The SCi Continuous System Simulation Language (CSSL)". Simulation, vol. 9, no. 6, December 1967; pp. 281 - 303.

APPENDIX I: FORMAL DEFINITION OF THE EXTENDED BACKUS-NAUR-FORM (EBNF)

A) The Alphabet:

The following keywords are used in EBNF:

TERMINAL SYMBOLS VERSION 3-MAY-83 15:29:49.

SYMBOL		CROSSREFERENCES
$	22	23
'	32	32
(	20	
(*	34	
)	20	
*)	34	
.	12	
=	12	
END_SYNTAX_DEFINITION	9	
[	21	
]	21	
{	22	
|	14	
}	24	

Assignment:
Productions are assigned to nonterminals by use of the symbol '='.

Concatenation:
There is no special keyword required for concatenation of symbols. Symbols which are supposed to follow each other are separated by SPACE's.

Alternatives:
Alternatives are separated by a vertical line '|'.

Grouping:
Groups of symbols may be defined by enclosing them into pairs of parentheses '(' and ')'. Groups have a higher priority than concatenation or alternatives.

Option:
A symbol or a sequence of symbols may optionally be present (but need not to be present) in the sample program if the syntactical construct in the language definition which is associated with it is surrounded by a pair of square brackets '[' and ']'.

Repetition:
A symbol or a sequence of symbols may appear one or several times in the sample program if the syntactical construct in the language definition which is associated with it is embraced by a pair of curly braces '{' and '}'.

Exit:
A repetition loop can have two different parts which are separated by the exit symbol '$'. The meaning of the structure { A $ B } is defined as:

$$\{ A \$ B \} = A \mid ABA \mid ABABA \mid \ldots .$$

If B is the empty symbol (ε), B must be omitted. In this case, also the exit symbol '$' can be omitted which then denotes a repetition (as defined above) which can also be interpreted as a transitive closure:

$$\{ A \} := A+ .$$

The transitive closure (A+) of a set (A) is equivalent to the closure (A*) of that set with the exception that the empty string (ε) does not belong to it. Therefore:

$$A* = A+ \cup \varepsilon .$$

If the symbol A of the structure { A $ B } is the empty string, we simply omit it, whereas the exit symbol '$' must be specified in this case. The meaning of the construct {$ B } is consequently that of an optional repetition or of a closure.

$$\{\$ B \} := B* .$$

Strings:
Terminal symbols are placed between two string delimiters ''' and ''' to make them distinguishable from nonterminals which are not marked. This rule replaces an explicite specification of the sets N and T.

Comments:
Comments can be freely intermixed with statements of the EBNF notation, as long as they do not split up any symbols. Comments are placed between pairs of '(*' and '*)'. Comments may be specified hierarchically (which differs from their definition in PASCAL which looks formally the same).

B) The Set of Nonterminals:

The following nonterminals are used for the EBNF definition:

```
NON-TERMINAL SYMBOLS  VERSION   3-MAY-83 15:29:49.
================================================

SYMBOL                      CROSSREFERENCES

CHARACTER                28    32    34    36#
COMMENT                  34#  <<<<
DIGIT                 *** UNDEFINED ***    36    <<<<
EXPRESSION               12    14#   20    21     22
                         23    24
FACTOR                   16    18#
IDENTIFIER               26    28#
LETTER                *** UNDEFINED ***    28     36
NON_TERMINAL_SYMBOL      11    18    26#
PRODUCTION                8    11#
SPECIAL_CHARACTER     *** UNDEFINED ***    36    <<<<
STRING                   30    32#
SYNTAX_DEFINITION         8#  <<<<
TERM                     14    16#
TERMINAL_SYMBOL          19    30#
```

The nonterminals: CHARACTER, EXPRESSION, FACTOR, IDENTIFIER, NON_TERMINAL_SYMBOL, PRODUCTION, STRING, TERMINAL_SYMBOL, and TERM shall be explained subsequently in the discussion of productions.

COMMENT determines a production which has been added for documentation only. It is not referenced at all. Since comments can be freely intermixed with processed text, it is impossible to include the description of comments into the EBNF notation.

DIGIT is a nonterminal which requires no further explanation. It denotes the set of the digits 1..9 and 0.

LETTER is a nonterminal which requires no further explanation either. It denotes the set of the letters A..Z.

SPECIAL-CHARACTER denotes the set of all those characters which do not belong to DIGIT or LETTER.

C) THE STARTING SYMBOL:

The nonterminal SYNTAX_DEFINITION is used as starting symbol for the generation of syntax trees.

D) PRODUCTIONS:

The following productions describe the syntactic rules of the EBNF notation:

```
SYNTAX DEFINITION  VERSION   3-MAY-83 15:29:49.
================================================

 1  (*********************************************)
 2  (*                                           *)
 3  (*          EBNF DEFINITION IN EBNF          *)
 4  (*                                           *)
 5  (*********************************************)
 6
 7
 8  SYNTAX_DEFINITION = { PRODUCTION }
 9                      'END_SYNTAX_DEFINITION' .
10
11  PRODUCTION = NON_TERMINAL_SYMBOL
12               '=' EXPRESSION '.' .
13
14  EXPRESSION = { TERM $ '|' } .
15
16  TERM = { FACTOR } .
17
18  FACTOR = NON_TERMINAL_SYMBOL
19         | TERMINAL_SYMBOL
20         | ( '(' EXPRESSION ')' )
21         | ( '[' EXPRESSION ']' )
22         | ( '{' ( ( '$' EXPRESSION )
23                  | ( EXPRESSION [ '$'
24                      EXPRESSION ] ) ) '}' ) .
25
26  NON_TERMINAL_SYMBOL = IDENTIFIER .
27
28  IDENTIFIER = LETTER { $ CHARACTER } .
29
30  TERMINAL_SYMBOL = STRING .
31
32  STRING = ''' { CHARACTER } ''' .
33
34  COMMENT = '(*' { $ CHARACTER } '*)' .
35
36  CHARACTER = LETTER | DIGIT | SPECIAL_CHARACTER .
37
38  $$$$$
```

Each syntax definition is embraced by the start symbol SYNTAX_DEFINITION and the "end-of-text" marker END_SYNTAX_DEFINITION. Inbetween, there may be specified as many productions as needed to describe the language to be defined.

Each production assigns an expression to a nonterminal. (Note: This is the standard notation rule for all CFG's.)

An expression consists of one or several alternative terms.

A term is a sequence of one or several factors.

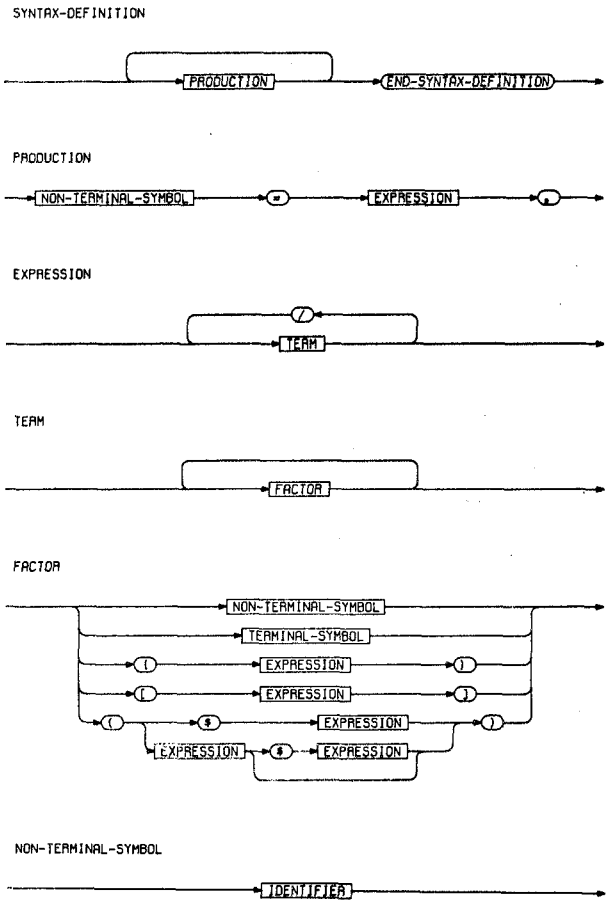
A factor is either a nonterminal symbol or a terminal symbol or a group or an option or a repetition with or without exit.

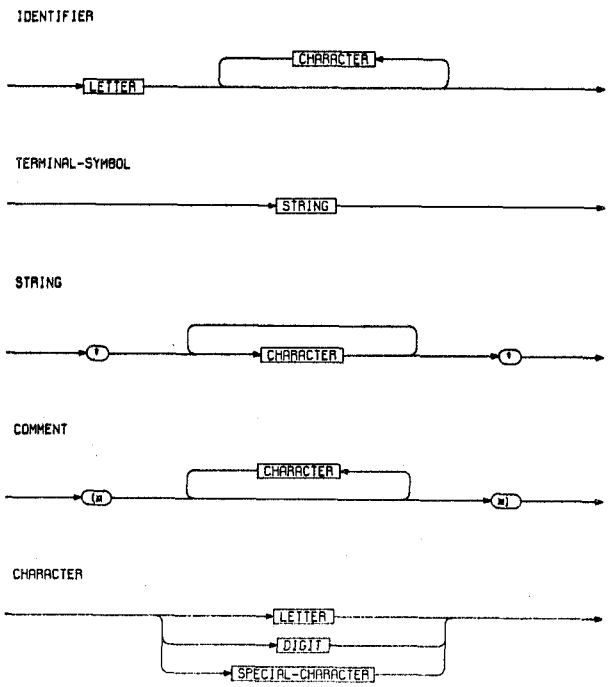
Nonterminals are identifiers. They all start with a letter, and may be continued by up to 39 characters (this number can be respecified in the implementation of the parser program). Nonterminals may, therefore, also contain special characters. Excluded

from use are only those characters which are used as the first character of any separator. (The underscore symbol which is not used by most languages directly, is often used in this report to denote compounds.) In the production for IDENTIFIER, the nonterminal CHARACTER must, therefore, be interpreted somewhat restrictive. Since the meta-symbols can be redefined in the control options block of the parser program, there is no way to include this restriction into the EBNF notation of EBNF. This rule must, therefore, be specified in the description of the static semantics of the language EBNF.

Terminal symbols are strings of characters. They may start with any character, and they may take eventually any form with the exception that the SPACE may not appear in a terminal, and that the string delimiters (') may not appear within terminals. The string delimiter may, however, be used as a terminal for itself (''''). The length of terminals is currently also restricted to 40 characters.

For better illustration, the representation of the syntax of the EBNF notation by use of syntax diagrams is added.





APPENDIX II: FORMAL DEFINITION OF SYNTAX DIAGRAMS

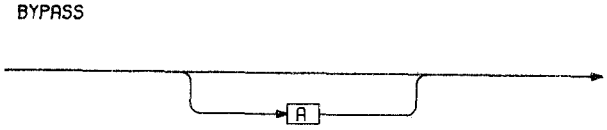
Since a description of a grammar by use of syntax diagrams is isomorphic to the description by use of an EBNF notation, it is sufficient to give here the mapping between an EBNF representation, and the representation by use of syntax diagrams.

Assignment:
Each production is represented by a separate syntax diagram which is labeled by the name of the nonterminal to which the production is assigned.

Concatenation:
A concatenation of factors is represented by a series connection of the diagrams of those factors.

Alternatives:
Alternatives are represented by a parallel connection of involved terms.

Option:
An option can be thought of as to be an alternative between a non-empty and an empty path. The empty path is depicted as a connecting line.

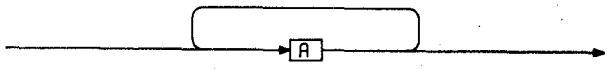


Note: Empty paths are not necessarily drawn as the "last alternative" as we demanded. This is, however, just a particularity of the syntax diagram drawing program, and not of the underlying EBNF notation

which is used by the parser program, and which is used as input to the syntax diagram plotting program as well.

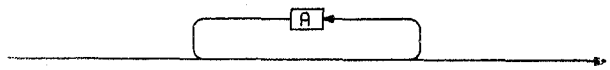
Repetition:
A repetition is represented by a feedback loop in the diagram. The construct { A } is depicted as:

REPETITION1



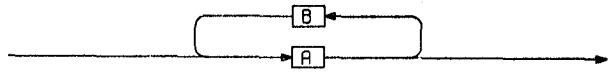
The construct {$ A } is depicted as:

REPETITIONØ



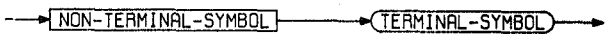
The general construct { A $ B } can be viewed as:

REPETITION



Terminals and Nonterminals:
The two sets are distinguished by the shape of the box by which they are represented. Nonterminals appear as a square box, whereas boxes for terminals have rounded corners.

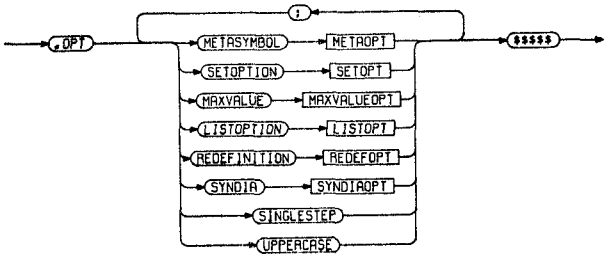
DUMMY



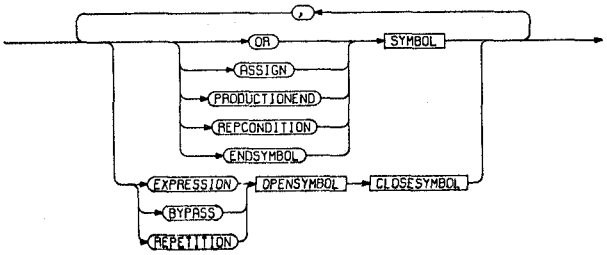
APPENDIX III: CONTROL OPTIONS OF THE GENERAL PURPOSE PARSER

This appendix describes the syntactical rules for the specification of control options for the general purpose parser. A syntax diagram has been used for representation.

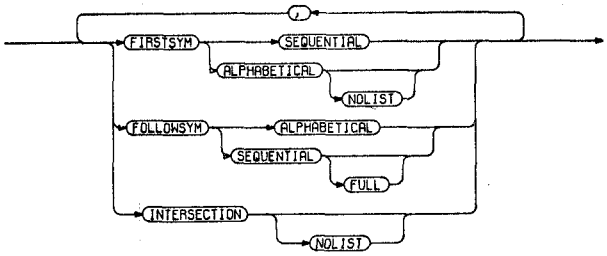
OPTIONS-DEFINITION



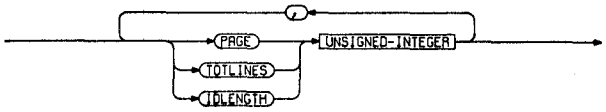
METAOPT



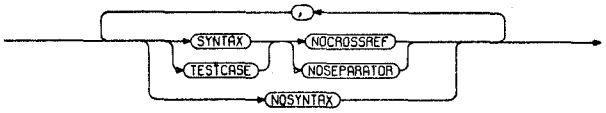
SETOPT



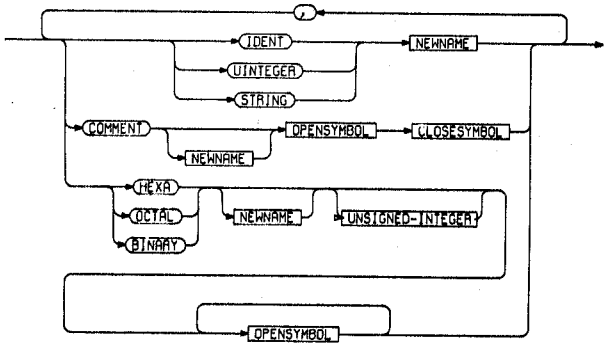
MAXVALUEOPT



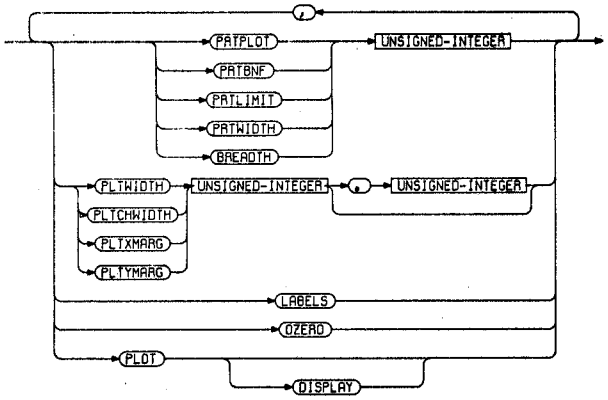
LISTOPT



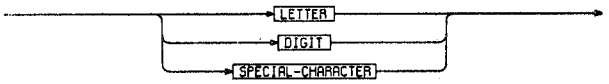
REDEFOPT



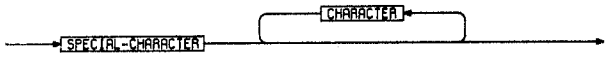
SYNDIAOPT



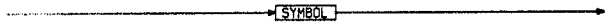
CHARACTER



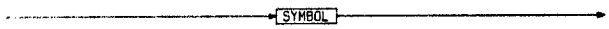
SYMBOL



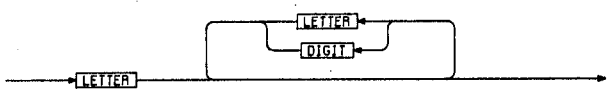
OPENSYMBOL



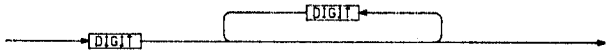
CLOSESYMBOL



NEWNAME



UNSIGNED-INTEGER



APPENDIX IV: EXAMPLE OF A CORRECT PARSER OUTPUT -- THE PL/0 LANGUAGE

```
OPTIONS DEFINITION  VERSION   3-MAY-83 15:30:03.
*************************************************

 1  .OPT
 2      SETOPTION
 3        FIRSTSYM ALPHABETICAL ,
 4        FOLLOWSYM ALPHABETICAL ,
 5        FIRSTSYM SEQUENTIAL   ,
 6        FOLLOWSYM SEQUENTIAL ,
 7        INTERSECTION ;
 8
 9
10      SYNDIA
11        PRTPLOT 100 ,
12        PRTBNF 70 ,
13        PLOT DISPLAY ,
14        PLTWIDTH 20.0 ,
15        PLTCHWIDTH 0.25 ,
16        LABELS ,
17        OZERO ;
18
19  $$$$$

SEPARATORS: SYNTAX DEFINITION   3-MAY-83 15:30:03.
*************************************************

        BLANK  $  $$$$$  '  (  (*  )  *)  .  =  [
        ]  {  |  }

COMMENT    DELIMITERS

        (*  *)

TERMINAL SYMBOL DELIMITERS

        '  '
```

SYNTAX DEFINITION VERSION 3-MAY-83 15:30:03.

```
 1  (*********************************************)
 2  (*                                           *)
 3  (*            PL0 SYNTAX DEFINITION          *)
 4  (*                                           *)
 5  (*********************************************)
 6
 7
 8  PROGRAM = IDENTIFIER BLOCK '.' .
 9
10  BLOCK = ['CONST' { IDENTIFIER
11        '=' UNSIGNED_INTEGER $ ',' } ';']
12        ['VAR' { IDENTIFIER $ ',' } ';']
13        { $ 'PROCEDURE' IDENTIFIER ';' BLOCK ';' }
14        STATEMENT .
15
16  STATEMENT = [(IDENTIFIER ':=' EXPRESSION)
17            |('CALL' IDENTIFIER)
18            |('BEGIN' { STATEMENT $ ';' } 'END')
19            |('IF' CONDITION 'THEN' STATEMENT)
20            |('WHILE' CONDITION 'DO' STATEMENT)] .
21
22  CONDITION = ('ODD' EXPRESSION)
23            |(EXPRESSION
24            ('=' | '<>' | '<' | '<=' | '>' | '>=')
25            EXPRESSION) .
26
27  EXPRESSION = ['+' | '-'] { TERM $ ('+' | '-')} .
28
29  TERM = { FACTOR $ ('*' | '/')} .
30
31  FACTOR = IDENTIFIER | UNSIGNED_INTEGER
32         | ('(' EXPRESSION ')') .
33
34  IDENTIFIER = 'IDENT' .
35
36  UNSIGNED_INTEGER = 'UINTEGER' .
37  $$$$$
```

NON-TERMINAL SYMBOLS VERSION 3-MAY-83 15:30:03.

SYMBOL	CROSSREFERENCES					
BLOCK	8	10#	13			
CONDITION	19	20	22#			
EXPRESSION	15	22	23	25	27#	32
FACTOR	29	31#				
IDENTIFIER	8	10	12	13	16	17
	31	34#				
PROGRAM	8#	<<<<				
STATEMENT	14	16#	18	19	20	
TERM	27	29#				
UNSIGNED_INTEGER	11	31	36#			

TERMINAL SYMBOLS VERSION 3-MAY-83 15:30:03.

SYMBOL	CROSSREFERENCES				
(	32				
)	32				
*	29				
+	27	27			
,	11	12			
-	27	27			
.	8				
/	29				
:=	16				
;	11	12	13	13	18
<	24				
<=	24				
<>	24				
=	11	24			
>	24				
>=	24				
BEGIN	18				
CALL	17				
CONST	10				
DO	20				
END	18				
IDENT	34				
IF	19				
ODD	22				
PROCEDURE	13				
THEN	19				
UINTEGER	36				
VAR	12				
WHILE	20				

'FIRST' SYMBOL SETS (SEQUENTIAL) 3-MAY-83 15:30:03.

```
BLOCK :
     1   CONST
<>   2   VAR
<>   3   PROCEDURE
<>   4 >>STATEMENT  >>IDENTIFIER
         IDENT  CALL  BEGIN  IF  WHILE
<>   5


CONDITION :
     1   ODD
       >>EXPRESSION
         +  -
<>   2 >>TERM  >>FACTOR  >>IDENTIFIER
         IDENT
       >>UNSIGNED_INTEGER
         UINTEGER  (


EXPRESSION :
     1   +  -
<>   2 >>TERM  >>FACTOR  >>IDENTIFIER
         IDENT
       >>UNSIGNED_INTEGER
         UINTEGER  (
```

```
FACTOR :
     1 >>IDENTIFIER
          IDENT
       >>UNSIGNED_INTEGER
          UINTEGER  (


IDENTIFIER :
     1    IDENT


PROGRAM :
     1 >>IDENTIFIER
          IDENT


STATEMENT :
     1 >>IDENTIFIER
          IDENT  CALL  BEGIN  IF  WHILE
<>   2


TERM :
     1 >>FACTOR  >>IDENTIFIER
          IDENT
       >>UNSIGNED_INTEGER
          UINTEGER  (


UNSIGNED_INTEGER :
     1    UINTEGER
```

'FIRST' SYMBOL SETS (ALPHABETICAL) 3-MAY-83 15:30:03

```
BLOCK :
<>        BEGIN  CALL  CONST  IDENT  IF
          PROCEDURE  VAR  WHILE


CONDITION :
          (  +  -  IDENT  ODD  UINTEGER


EXPRESSION :
          (  +  -  IDENT  UINTEGER


FACTOR :
          (  IDENT  UINTEGER


IDENTIFIER :
          IDENT


PROGRAM :
          IDENT


STATEMENT :
<>        BEGIN  CALL  IDENT  IF  WHILE


TERM :
          (  IDENT  UINTEGER


UNSIGNED_INTEGER :
          UINTEGER
```

'FOLLOW' SYMBOL SETS (SEQUENTIAL) 3-MAY-83 15:30:03.

```
BLOCK :
     1    .
     1    ;


CONDITION :
     1    THEN
     1    DO


EXPRESSION :
     1    =  <>  <  <=  >  >=
     1    )
->   2 >>STATEMENT
->   2 >>CONDITION


FACTOR :
     1    *  /
<>   2
->   2 >>TERM


IDENTIFIER :
     1 >>BLOCK
     1    CONST
<>   2    VAR
<>   3    PROCEDURE
<>   4 >>STATEMENT  >>IDENTIFIER
     4    IDENT  CALL  BEGIN  IF  WHILE
<>   5    .
     1    =
     1    ,
<>   2    ;
     1    ;
     1    :=
->   2 >>PROGRAM
->   2 >>STATEMENT
->   2 >>FACTOR


PROGRAM :
     1


STATEMENT :
     1    ;
<>   2    END
->   2 >>BLOCK


TERM :
     1    +  -
<>   2
->   2 >>EXPRESSION


UNSIGNED_INTEGER :
     1    ,
<>   2    ;
->   2 >>FACTOR
```

```
'FOLLOW' SYMBOL SET (ALPHABETICAL) 3-MAY-83 15:30:03
====================================================


        BLOCK :
                .  ;


        CONDITION :
                DO  THEN


        EXPRESSION :
                )  .  ;  <  <=  <>  =  >  >=  DO
                END  THEN


        FACTOR :
                )  *  +  -  .  /  ;  <  <=  <>  =
                >  >=  DO  END  THEN


        IDENTIFIER :
                )  *  +  ,  -  .  /  :=  ;  <  <=
                <>  =  >  >=  BEGIN  CALL  CONST
                DO  IDENT  IF  PROCEDURE  THEN
                VAR  WHILE


        PROGRAM :
              >>>>> EMPTY 'FOLLOW' SYMBOL SET <<<<<


        STATEMENT :
                .  ;  END


        TERM :
                )  +  -  .  ;  <  <=  <>  =  >  >=
                DO  END  THEN


        UNSIGNED_INTEGER :
                )  *  +  ,  -  .  /  ;  <  <=  <>
                =  >  >=  DO  END  THEN


'INTERSECTIONS' ('FIRS'-'FOLL') 3-MAY-83 15:30:03.
==================================================


        NO 'INTERSECTION' SET FOUND


SEPARATORS: SYNTAX CHECK   3-MAY-83 15:30:03.
=============================================

                BLANK  $$$$$  (  (*  )  *  *)  +  ,
                -  .  /  :=  ;  <  <=  <>  =  >  >=


        COMMENT    DELIMITERS

                (*  *)
```

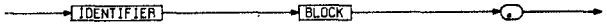
```
                VERSION   3-MAY-83 15:30:03.

 1  (*********************************************)
 2  (*                                           *)
 3  (*            PL/0 SAMPLE PROGRAM            *)
 4  (*                                           *)
 5  (*********************************************)
 6
 7
 8  PROGRAM EXAMPLE
 9
10  CONST M=7 , N=85  ;
11  VAR  X,Y,Z,Q,R ;
12
13
14  PROCEDURE MULTIPLY ;
15  VAR  A,B  ;
16
17  BEGIN  A:=X  ;  B:=Y  ;  Z:=0  ;
18    WHILE  B>0  DO
19      BEGIN  IF ODD B  THEN  Z:=Z+A  ;
20        A:=2*A  ;  B:=B/2
21      END
22  END (* MULTIPLY *)  ;
23
24
25  PROCEDURE DIVIDE  ;
26  VAR  W  ;
27
28  BEGIN  R:=X  ;  Q:=0  ;  W:=Y  ;
29    WHILE  W<=R  DO  W:=2*W  ;
30    WHILE  W>Y  DO
31      BEGIN  Q:=2*Q  ;  W:=W/2  ;
32        IF  W<=R  THEN
33          BEGIN  R:=R-W  ;  Q:=Q+1
34          END
35      END
36  END (* DIVIDE *)  ;
37
38
39  (*----- MAIN PROGRAM -----*)
40  BEGIN
41    X:=M  ; Y:=N ; CALL MULTIPLY  ;
42    X:=25 ; Y:=3 ; CALL DIVIDE
43  END .
44  $$$$$
                    CORRECT
```

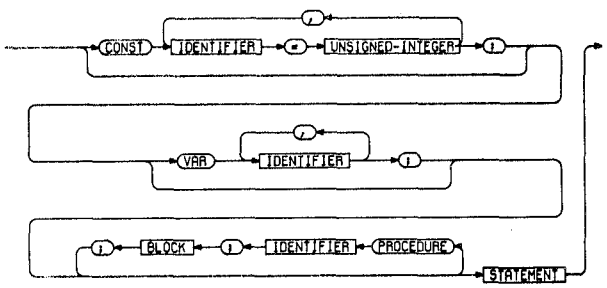
IDENTIFIER LIST VERSION 3-MAY-83 15:30:03.

SYMBOL	CROSSREFERENCES					
A	15	17	19	20	20	
B	15	17	18	19	20	20
DIVIDE	25	42				
EXAMPLE	8					
M	10	41				
MULTIPLY	14	41				
N	10	41				
Q	11	28	31	31	33	33
R	11	28	29	32	33	33
W	26	28	29	29	29	30
	31	31	32	33		
X	11	17	28	41	42	
Y	11	17	28	30	41	42
Z	11	17	19	19		

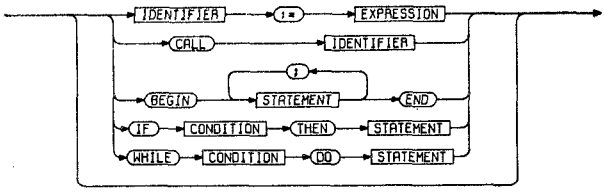
PROGRAM



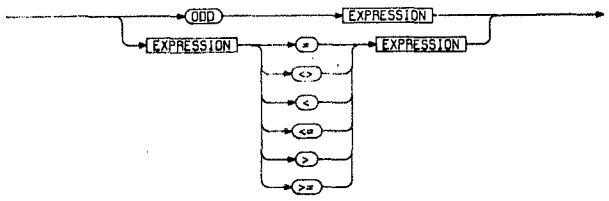
BLOCK



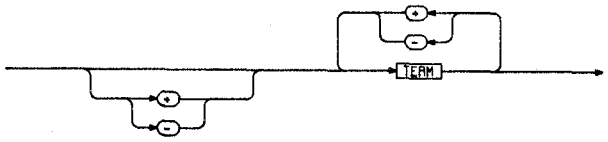
STATEMENT



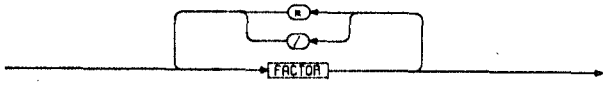
CONDITION



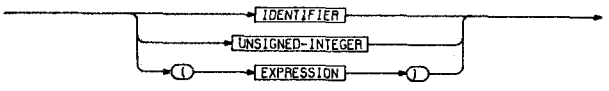
EXPRESSION



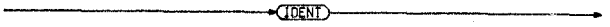
TERM



FACTOR



IDENTIFIER



UNSIGNED-INTEGER