stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 17/74 JUNE

D. GRUNE, R. BOSCH & L.G.L.T. MEERTENS

ALEPH MANUAL

2e boerhaavestraat 49 amsterdam

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

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

ACM - Computing Reviews - category: 4.22, 4.12
AMS (MOS) subject classification scheme (1970): 68A15

Abstract

ALEPH (acronym for "A Language Encouraging Program Hierarchy") is a high-level language designed to provide the programmer with a tool that will effectively aid him in structuring his program in a hierarchical fashion. The syntax of ALEPH enables the compiler to utilize the well-structuredness of the program for optimization purposes. Thus the loss of efficiency usually incurred in structured programming is avoided. ALEPH is suitable for any problem that suggests top-down analysis (parsers, search algorithms, combinatorial problems, artificial intelligence problems, etc.).

0. Preface.

ALEPH (acronym for "A Language Encouraging Program Hierarchy") is a high-level language designed to provide the programmer with a tool that will effectively aid him in structuring his program in a hierarchical fashion. The syntax of ALEPH enables the compiler to utilize the well-structuredness of the program for optimization purposes. Thus the loss of efficiency usually incurred in structured programming is avoided. ALEPH is suitable for any problem that suggests top-down analysis (parsers, search algorithms, combinatorial problems, artificial intelligence problems, etc.).

Chapter one of this Manual gives a tutorial introduction into the way of thinking that is used in ALEPH. It addresses itself to computer users that have some experience with algorithms and grammars, though not necessarily with high-level languages. It must not be concluded from these prerequisites that ALEPH should not be taught to the novice programmer. On the contrary, ALEPH introduces him to a discipline of thought that is lacking in many other languages. Experiments in teaching ALEPH and its parent CDL have proved to be successful (cf. C.H.A. Koster, Portable compilers and the UNCOL concept in Proc. of IFIP Working Conf. on Machine Oriented High Level Languages, Trondheim, 1973).

Chapter three through six contain a complete description of ALEPH. Chapter three treats the flow-of-control. Chapter four treats the data-types. Externals, i.e., standard-operations and communication with the outside world, are treated in chapter five. Chapter six describes the pragmats.

The representation of symbols, examples and a summary of the grammar are given in chapter seven, eight and nine, respectively.

The discipline of thought needed in ALEPH is different from that needed in most other programming languages. We are very interested in accounts of experience with ALEPH and we would be glad to receive comments, suggestions and wishes.

This report is to be followed by a report on the implementation of ALEPH. A working ALEPH compiler exists for the CYBER 73.

The text of this manual was justified by a text justifier written in ALEPH; it is available on paper tape.

1. Informal introduction.

In this chapter we shall gradually develop a small ALEPH program and intersperse it liberally with annotations and arguments. This introduction is intended to give some insight into the use of the language ALEPH and to display its main features in a very informal way.

1.1. The problem we are going to tackle is the following. We want to write a program that reads a series of arithmetic expressions separated by commas, calculate the value of each expression while reading it, and subsequently print the value. The expressions will contain only integers, plus symbols, times symbols and parentheses: an example might be 15 × (12 + 3 × 9).

First we put the requirements for the input to our program in the more transparent and clear form of a grammar. This grammar shows exactly which symbol we will accept in which position.

```
input: expression, input tail.
input tail: comma symbol, input; empty.
expression: term, plus symbol, expression; term.
term: primary, times symbol, term; primary.
primary: left parenthesis, expression, right parenthesis; integer.
integer: digit, integer; digit.
```

The rule for "input" can be read as:
"input" is
        an expression followed by an input tail,
whereas the rule for "primary" can be read as:
a "primary" is either
        a left parenthesis followed by an expression followed by a
        right parenthesis
or
        an integer.

This grammar makes it clear that for instance 15 × + 3 will not be accepted as an expression. The × can only be followed by a "term", which always starts with a "primary", which in turn either starts with an "integer" or a "left parenthesis", but never with a +.

1.2. We shall now write a series of rules in ALEPH, one for each rule in the grammar. For the grammar rule for "expression" we shall write an ALEPH rule that, when executed, reads and processes an expression and yields its result. This ALEPH rule looks as follows:

```
'action' expression + res> - r:
        term + res,
       (is symbol + /+/, expression + r, plus + res + r + res; +).
```

This can be read as: an expression, which must yield a result in res and uses a variable r is (we are now at the colon) a term which will yield a result in res, followed either (we are now at the left parenthesis) by a + symbol followed by an expression which will yield its result in r after which the result in res and the result in r will be added to form a new result in res, or (we are at the semicolon now) by nothing. We see that this is the old meaning of the grammar rule for "expression", sprinkled with some data-handling. This data-handling tells what is to be done to get the correct result: we could call it the semantics of an "expression". If we remove these paraphernalia from the ALEPH rule we obtain something very similar to the original grammar rule:

```
'action' expression1: $ to avoid confusion with "expression" $
         term, (is symbol + /+/, expression1; +).
```

This rule, while it is still correct ALEPH, does no data handling and, consequently, will not yield a result; it could for example be used to skip an "expression" in the input. Note that comments may be added between $ and $.

We now direct our attention back to the ALEPH rule "expression" and consider what happens when it is executed (called). First, "term" is executed and will yield a result in res: it does so because we shall define "term" so that it will. Then we meet a series of two alternatives seperated by a semicolon (either a this or a that). First an attempt is made to execute the first alternative by asking "is symbol + /+/". This is a question (because we shall define it so) which is answered positively if indeed the next symbol is a + (in which case the + will be discarded after reading) or negatively if the next symbol is something else. Again it does so because we shall define it that way.

If "is symbol + /+/" succeeds (is answered positively) the remainder of the first alternative is executed, "expression + r" is called (recursively), yielding its result in r and subsequently "plus + res + r + res" is called, putting the sum of res and r in res. The call of "expression + r" works because we just defined what it should do. "plus" is a name known to the compiler and has a predefined meaning. However, if we are dissatisfied with its workings we could define our own rule for it. Now this alternative is finished, so the parenthesized part is finished, which brings us to the end of the execution of the rule "expression".

If "is symbol + /+/" fails (is answered negatively) the second alternative is tried: the part after the semicolon. This alternative consists of a + which is a dummy statement that always succeeds. Without further action we reach the end of the rule "expression".

The above indicates the division of responsibility between the language and the user. The language provides a framework that controls which rules will be called depending on the answers obtained from other rules. The user must fill in this framework, by defining what actions must be performed by a specific rule and what questions must be asked. These definitions again will have the form of rules that do something (to be defined by the user) embedded in a framework that controls their order (supplied by the language). It is clear that this process must end somewhere. It can end in one of two ways.
It may appear that the action needed is supplied by ALEPH: there are two very basic primitives in the language, the copying of a value, and the test for equality of two values. Often, however, these two primitives are not sufficient to express the action needed. Normally the rule is then decomposed into other rules; however, there are cases where this is not desirable (or not possible). In such cases the rule is declared as 'external' and its actions must be specified in a different way, generally in the assembly language of the machine used. By specifying a rule as 'external' we leave the realm of machine-independent semantics. A number of external rules are predefined by the compiler, including the rule "plus" used above. This set of rules will suffice for most applications.

Now we shall pay some attention to the exact notation (syntax) of the rule "expression". The word 'action' indicates that this rule specifies an action to be performed, not a question to be asked. "expression" is the name of the rule, "res" is its only formal affix (parameter). The + serves as a separator (it "affixes" the affix to the rule). The right arrow-head (>) indicates that the resulting value of "res" will be used by the calling rule. This means that "expression" has the obligation to assign a value to "res" under all circumstances: "res" is an output parameter, guaranteed to receive a value. If the text of the rule does not support this claim, the compiler will discover so and issue a message. The +-sign and the term 'affix' stem from the theory of affix grammars on which ALEPH is based [2].
The "-r" specifies "r" as a local affix (local variable) of this rule and the colon closes the left hand side. The + in "term + res" appends the actual affix "res" to the rule "term", the comma separates subsequent calls of rules. The parentheses group both alternatives virtually into one action. The + between slashes (indicating absolute value) represents the integer value of the plus symbol in the code used. The semicolon separates (mutually exclusive) alternatives. As said before, the stand-alone + denotes the dummy action that always succeeds. The period ends the rule.

For the reader who is familiar with ALGOL-like languages the following approximate translation to ALGOL 68 might be helpful:

```
proc expression= (ref int res) void:
begin   int r; term(res);
        if is symbol("+") then expression(r); plus(res,r,res)
                          else skip fi
end     .
```

An approximate PL/I version runs as follows:

```
EXPRESSION:
    PROCEDURE(RES) RECURSIVE;
        DECLARE(RES,R) FIXED BINARY;
        CALL TERM(RES);
        IF IS_SYMBOL('+') THEN
        DO; CALL EXPRESSION(R);
            CALL PLUS(RES,R,RES);
        END; ELSE;
    END;
```

1.3. In view of the above the ALEPH rule for "term" should not surprise the reader:

```
'action' term + res> - r:
         primary + res,
           (is symbol + /x/, term + r, times + res + r + res; +).
```

Now we are tempted to render the ALEPH rule for "primary" as:

```
'action' primary + res>:
        is symbol + /(/, expression + res, is symbol + /)/;
        integer + res.
```

but here the compiler would discover that we did not specify what should be done if the second call of "is symbol" fails. If that happens, we would have recognized, processed and skipped a left parenthesis and a complete expression, to find that the corresponding right parenthesis is missing. This means that the input is incorrect; we now decide that we shall not do any error recovery, so we give an error message and stop the program. The correct version of the ALEPH rule "primary" is then:

```
'action' primary + res>;
        is symbol + /(/, expression + res,
               (is symbol + /)/;
                error + no paren);
        integer + res.
```

Here the two alternatives between parentheses behave as one action that will always succeed: either the right parenthesis is present in the input, or an error will be signalled. "no paren" is a constant that will be specified later on in this example.

Writing the rule for "integer" is a trickier problem than it seems to be. For a comprehensive account on how to obtain correct and incorrect versions of it the reader is referred to [1]. We shall confine ourselves to giving one correct version. It consists of two rules and is about as complicated as necessary.

```
'action' integer + res>:
        digit + res, integer1 + res;
        error + no int, 0 -> res.

'action' integer1 + >res> - d:
        digit + d, times + res + 10 + res,
                plus + res + d + res, integer1 + res; +.
```

The rule "integer" asks for a digit. If present, its value will serve as the initial value of "res". The value of "res" is then passed to "integer1". If no digit is present an error message will result and "res" will get the dummy value 0. This is necessary to ensure that "integer" will assign a value to "res" under all circumstances (necessary because of the right arrow-head after "res"). The right arrow in "0 -> res" indicates the assignation of the value on the left to the variable on the right, one of the primitive actions in ALEPH.
The rule "integer1" processes the tail of the integer. If there is such a tail it starts with a digit, so the first alternative asks "digit + d". If so, a new result is calculated from the previous one and the digit "d" by making "res" equal to "res × 10 + d" and "integer1" is called again (to see if there are more digits to come). If there was no digit, we have processed the whole integer and "res" contains its value.
The rule "integer1" is recursive; this should not worry the reader since the compiler will discover that this is a case of trivial right-recursion and will optimize it into a simple jump.
The right arrow-head in front of "res" means that the calling rule will have assigned a value to this affix just before calling "integer1", i.e. "res" is initialized. The right arrow-head after "res" again indicates that the resulting value will be passed back to the calling rule.

A more convenient way of reading an integer is provided by the (standard) external rule "get int".

1.4. The above forms the heart of our program. We shall now supply it with some input and output definitions. For the input we need a file to obtain the input symbols from, which we shall call "reader"; let us suppose that this file is called "SYSIN" somewhere in the surrounding operating system (e.g. on a control card). Furthermore we shall use a global variable "buff" which will contain the first symbol not yet recognized.

```
$ input $
'charfile' reader = >"SYSIN".
'variable' buff = / /.
```

The variable "buff" is initialized with the code for the space symbol (there being no uninitialized variables in ALEPH). We are now in a position to give two rule definitions that were still missing.

```
'predicate' is symbol + >n: buff = n, get next symbol.

'predicate' digit + d>:
        =buff=
        [/0/: /9/], minus + buff + /0/ + d, get next symbol;
        [   :    ], - .
```

These require some more explanation, mainly concerning the notation. The word 'predicate' indicates that "is symbol" is not an action but a question, or more precisely a "committing" question as opposed to a "non-committal" question. A non-committal question is a question that, regardless of the answer it yields, makes no global changes, does not do anything irreversible. A committing question is a question that, when answered positively, does make global (and often irreversible) changes, according to programmer specification. To give an example, "Are there plane tickets for New York for less than $ 100?" is a non-committal question, whereas "Are there plane tickets for New York for less than $ 100? If so, I want one" is a committing question.
In the case of "is symbol" the (committing) question is: "Is the symbol in "buff" equal to the one I want? If so, throw it away and put the next symbol in "buff"." The form "buff=n" is a test for equality and is one of the primitive operations in ALEPH. "get next symbol" will be defined below.

Again the right arrow-head in front of the formal affix "n" indicates that the calling rule will have assigned a value to it; the absence of a right arrow-head to the right of "n" indicates that the calling rule will not use the value of "n".

The rule for "digit" (again a 'predicate') shows another feature of ALEPH, the classification. For certain classes of values of "buff" one alternative will be chosen, for other classes a different alternative will be chosen. The classes are presented inside the square brackets. Thus, for values of "buff" that lie between the code for "0" and the code for "9" the first alternative will be chosen. For all other values the dummy question that always fails (minus-sign) will be executed. The rule "digit" is equivalent to

```
'predicate' digit1 + d >:
        between + /0/ + buff + /9/, minus + buff + /0/ + d,
                get next symbol; -.
```

assuming that "between + /0/ + buff + /9/" succeeds if and only if /0/ $\leq$ buff $\leq$ /9/. In complicated cases a "classification" is easier to write and will in general produce more efficient object code. The concept of "classification" is analogous to "case" statements in ALGOL 68 and other programming languages.

All the arithmetic used here on symbols is based on the (hopefully machine-independent) assumption that the numerical codes associated with the symbols "0" through "9" are a set of consecutive integers in ascending order. The numerical value of a digit symbol can then indeed be obtained by subtracting the code for "0" from its numerical code.

One more input rule must be supplied:

```
'action' get next symbol:
        get char + reader + buff,
                ((buff = / /; buff = newline),
                  get next symbol;
                  + );
        stop -> buff.

'constant' stop = -1.
```

"get char" is a(n external) rule known to the compiler. It tries to read the next symbol from the file indicated by its first affix (here "reader"); if there is a symbol it puts it in its second affix (here "buff"), if there is no symbol it fails. In the latter case "buff" is given the value "stop", which is defined in a constant-declaration to be -1.
If "get char" does yield a symbol and if this is a space or a new-line, "get char" is called again. We use nested parenthesizing here.
This definition of "get char" implies we have decided that spaces and new-lines are allowed in the input in all positions (a decision that was not yet present in the initial grammar).

1.5. The output is as follows:

```
$ output $
'charfile' printer = "SYSOUT"> .

'action' print integer + >int:
        out integer + int, put char + printer + newline.

'action' out integer + >int - rem:
        divrem + int + 10 + int + rem, plus + rem + /0/ + rem,
        (int= 0; out integer + int), put char + printer + rem.
```

The rule "put char" is known to the compiler, as is "divrem". The call of the latter has the effect that "int" is divided by ten, the quotient is placed back in "int" and the remainder in "rem". This splits the number into its last digit and its head; if this head (now in "int") is not zero it must be printed first, which is effected by the recursive call of "out integer". Subsequently, the last digit is printed through a call of "put char". This is a convenient though inefficient way of printing a number: here the recursion is essential and will not be optimized out by the compiler.

A more convenient way of printing an integer is provided by the (standard) external rule "put int".

For the printing of errormessages we shall need some string handling. Strings do not constitute a special data type in ALEPH: they are handled, like all other complicated data types, by putting them in 'stack's and 'table's and are operated upon by suitably defined rules (generally defined by the programmer but sometimes predefined in the system).

The error handler takes the following form:

```
$ errormessage printing $
'action' error + >er:
        put char + printer + newline,
                put string + printer + strings + er, 'exit' 1.

'table' strings= ("right parenthesis missing": no paren,
                  "integer missing": no int).
```

The table "strings" contains two strings, stored and packed in a way suitable to our machine; they can be reached under the names "no paren" and "no int". The call of "put string" takes the affix "er", looks in the table "strings" under the entry corresponding to "er" and transfers the string thus found to the file indicated as "printer".

When the construction "'exit' 1" is executed the program will be terminated and the 1 will be passed to the operating system as an indication of what went wrong. This is by no means the normal program termination: normal program termination ensues when all work is done.

1.6. The rule for reading an expression ("expression") and the one for printing an integer ("print integer") can now be combined into the rule "input" (see the grammar at the beginning of this chapter).

```
'action' input - int:
        expression + int, print integer + int,
               (is symbol + /,/, input; +).
```

This rule combines the grammar rules for "input" and "input tail". Instead of translating "empty" by "+", we could make a test to see whether we have indeed reached the end of the file:

```
        (buff = stop; error + no end)
```

We now remember our convention that "buff" contain the first symbol not yet recognized, and realize that "buff" must be initialized with the first non-space symbol of the input:

```
'action' initialize: get next symbol.

'action' read expressions and print results: initialize, input.
```

The reader will have noticed that up until now we have only defined rules that will do something if they are executed (called) and which will then call other rules. He may have wondered whether ALEPH contains any directly executable statements at all. The answer is yes, but only one (per program). In our example it has the following form:

```
'root' read expressions and print results.
```

We now indicate the end of our program:

```
'end'
```

When the program is run the rule "read expressions and print results" is executed. This rule calls "initialize", which through a call of "get next symbol" puts the first non-space symbol in "buff"; when "initialize" is done, "input" is called which calls "expression" which in turn executes "term", etc.. After a while "input", which is called repeatedly, will find "is symbol + /,/" to fail, it is done and so is "read expressions and print result". The call specified in the 'root' instruction is finished: this constitutes the normal program termination.

We could have given the rule-declarations and data-declaration in any other order and the effect would still have been the same. The 'end', however, must be the last item of the program.

This brings us to the end of our sample program.

1.7. Although the rule "put string" used above is known to the compiler, it is useful to see, as an additional example, how it looks when expressed in ALEPH. We first propose the preliminary version "put string 1".

```
'action' put string 1 + ""file + table[] + >string - count:
        0 -> count, next1 + file + table + string + count.

'action' next1 + ""out + tbl[] + >str + >cnt - symb:
        string elem + tbl + str + cnt + symb, put char + out + symb,
                incr + cnt, next1 + out + tbl + str + cnt; + .
```

The double set of quotation marks ("") indicates that the corresponding actual affix will be a file, the square brackets indicate that the corresponding actual affix will be a table. We see that the only thing "put string 1" does is creating an environment for "next1" to run in. "next1" starts by calling "string elem". This (standard) rule considers the string in "tbl" designated by "str" and determines whether this string has a "cnt"-th symbol. If so, it puts it in "symb"; if not, it fails. If the call fails, we know we reached the end of the string and we are done. Otherwise the symbol is transferred to the file indicated by "out", the counter "cnt" is increased by 1 (through the external rule "incr") and "next1" is called again with the same affixes. Like at the first call of "next1", the value of "cnt" is the position in the string of the symbol to be processed.
The recursive call of "next1" is again a case of trivial right-recursion; moreover all actual affixes are the same as the formal affixes (which are left of the colon). In this case the recursive call is equivalent to a straightforward jump: it does not even necessitate parameter transfers. For this case there is a shorthand notation: a name of a rule preceded by a colon denotes the re-execution of that rule with the affixes it had upon its initial call (of course this is only allowed inside that same rule and only if the recursion is trivial right-recursion). Now we can write a simplified version:

```
'action' put string 2 + ""file + table[] + >string - count:
        0 -> count, next2 + file + table + string + count.

'action' next2 + ""out + tbl[] + >str + >cnt - symb:
        string elem + tbl + str + cnt + symb,
                put char + out + symb, incr + cnt, : next2; + .
```

The gain is twofold. Firstly we no longer have to write that tail of affixes which only convey the information "same as before". Secondly, what is more important, the rule "next2" is now called only in one place (in "put string 2"), which means that we could as well have written it there explicitly. We now replace the call of "next2" in "put text2" by the definition of "next2": we parenthesize the rule, substitute for each formal affix its corresponding actual affix and then remove the formal affixes.

The final version is then:

```
'action' put string + ""file + table[] + >string - count:
        0 -> count,
       (next - symb:
                 string elem + table + string + count + symb,
                 put char + file + symb, incr + count, :next;
                 + ).
```

Note that this mechanism of replacing a call of a rule by its (slightly modified) definition is not applied here for the first time. We have been using it tacitly from the very first sample rule in 1.2.. There the rule "expression" is a contraction of:

```
'action' expression1 + res>:
        term + res, expression tail1 + res.
```

and

```
'action' expression tail1 + >res> - r:
        is symbol + /+/, expression1 + r, plus + res + r + res; +.
```

which, according to the above recipe, would yield:

```
'action' expression2 + res> :
        term + res,
       (expression tail2 - r:
        is symbol + /+/, expression2 + r, plus + res + r + res; +).
```

In a sense this is a more appropriate form than the one given in 2.2.: now the "r" occurs where it belongs, that is, in the position of a local affix of the parenthesized part only. To obtain the version of 2.2. exactly one must start from:

```
'action' expression3 + res> - r:
        term + res, expression tail3 + res + r.
```

and

```
'action' expression tail3 + >res> + r >:
        is symbol + /+/, expression3 + r, plus + res + r + res; +.
```

1.8. References.

[1] Koster, C.H.A., A Compiler Compiler, MR 127/71, Mathematical Centre, Amsterdam (1971).

[2] Koster, C.H.A., Affix-grammars, in ALGOL 68 Implementation, ed. J.E.L. Peck, North-Holland Publ. Co., Amsterdam (1971).

2. The syntactical description.

Chapter 9 contains the complete syntax of ALEPH; excerpts of it are used in this manual as a basis for the explanations. The notation in the grammar follows a usual scheme: the part on the right hand side defines the possible productions of the notion on the left hand side. The right hand side consists of one or more alternatives, separated by semicolons, of which only one alternative applies in a given case. Sometimes one or more notions in an alternative are parenthesized: this indicates that the given notions may or may not be present, i.e., they are optional.

A notion that ends in "symbol" is a terminal symbol as listed in 7.2.. A notion that ends in "tag" produces 'tag'. Such a notion then contains a hint as to exactly which 'tag's are allowed by the context conditions. This makes the syntax ambiguous, an ambiguity that is again resolved by the context conditions.

Example:

```
tag:     letter, tag tail.
tag tail: (letter or digit, tag tail).
```

The solution of a problem by means of a computer implies that a sequence of actions be specified that, when executed, leads to the desired result. In ALEPH the actions in this sequence may be obtained from four sources:

a. the framework of the language (supplied by the compiler),
b. the program (supplied by the programmer),
c. the standard externals (standard definitions of actions, to be supplied by the compiler if the need arises),
d. the programmer-defined externals (definitions of actions supplied by the programmer but not belonging to the program, for example, precompiled code or machine code).

The framework of ALEPH is treated in chapter 3., the program is treated in section 3.1. and the externals are treated in chapter 5.

The data needed in solving the problem at hand come from four sources:

a. the data descriptions in the program,
b. the input file(s),
c. the predefined constants in the compiler (e.g., the maximum value an integer can have),
d. the programmer-defined external values (for the rare case that these values cannot be normally defined in the program, as for example computer-generated binary tables of considerable size).

The data descriptions and the input files are explained in chapter 4., and the externals again in chapter 5..

The results can be passed back to the outer world along two paths:

a. as output files,
b. as a single integer (the termination state of the program) that is passed to the operating system upon termination of the program, indicating in some way the outcome of the program.

The output files are described in section 4.2.. The termination state is described in 3.1. and in 3.6.. In some operating systems it can be used to control the further course of events, in other operating systems it may only indicate whether the program proceeded satisfactorily or broke off because of some irrecoverable error.

The syntax presented in chapters 3 through 6 differs from the one in chapter 9. It gives a phenomenological description, regardless for instance of whether a given tag may be used in a given way; the syntax in chapter 9 is more elaborate and implies restrictions on the proper use of tags, although without enforcing them.

3. Program logic.

3.1. General.

3.1.1. The program.

Syntax:

```
program: (information sequence),
        root, (information sequence), end symbol.
information sequence: information, (information sequence).
information: declaration; pragmat.
root: root symbol, affix form, point symbol.
declaration:
        rule declaration; data declaration; external declaration.
```

The syntax of 'program' can be verbalized as: "A 'program' is a sequence of 'declaration's and 'pragmat's, followed by an 'end symbol'; in this sequence exactly one 'root' must occur." The order in which the 'declaration's and the 'root' appear is immaterial. The position of some 'pragmat's is significant.

Example of a program:

```
'file' output = "PRINTER">.
'root' put char + output + /3/.
'end'
```

where the first line is a 'data declaration', the second is the 'root' and the third contains the 'end symbol'. For other examples see chapter 8.

The execution of a 'program' starts with the processing of all of its 'data declaration's, in such an order that never a data item is used before its value has been calculated. If no such order exists an error-message is given.

Example: the 'data declaration's
```
'constant' p = q.
'constant' q = 3.
```
are processed in reverse order, whereas the 'data declaration's
```
'constant' p = q.
'constant' q = 2 - p.
```
will result in an error-message.

A large part of the processing of the 'data declaration's will normally be performed during compilation.

After all constants, variables, stacks, tables and files have thus been established, the 'affix form' in the 'root' is executed as the only directly executable instruction in the program. If this 'affix form' reaches its normal completion, the program finishes with a termination state of 0. If the execution of the 'affix form' stops prematurely, the program finishes, but now with a termination state possibly different from 0. If the stop is due to an 'exit' instruction, the termination state is specified by this instruction. If the stop is due to a run-time error the termination state is -1.

3.1.2. The use of 'tag's.

A 'tag' is a sequence of letters and digits, the first of which is a letter. All 'tag's defined in 'rule declaration's, 'pointer initialization's, 'constant description's, 'variable description's, 'table description's (except those in 'field list pack's), 'stack description's (except those in 'field list pack's), 'file description's, 'external rule description's, 'external table description's and 'external constant description's must differ from each other.

3.2. Rules.

The declarations and applications of 'rule's constitute the mechanism for controlling the logical flow of the program. The 'rule declaration' defines _what_ is to be done _if_ the 'rule' is called whereas the application (in an 'affix form') indicates _that_ the 'rule' is to be called.

A rule, when called, will either succeed or fail, according to criteria to be explained later.

3.2.1. Rule declarations.

Each rule in the program must be declared exactly once, either in a 'rule declaration' or in an 'external rule description' (for the latter see 5.).

```
Syntax:
rule declaration:
        typer, rule tag, (formal affix sequence),
                actual rule, point symbol.
typer:  action symbol;
        function symbol;
        predicate symbol;
        question symbol.
```

Example of a 'rule declaration':

```
'action' put string + ""file + table[] + >string - count:
        0 -> count,
        (next - symb:
                string elem + table + string + count + symb,
                put char + file + symb, incr + count, :next;
                + ).
```

Here the 'typer' is "'action'", the 'rule tag' is "put string", the 'formal affix sequence' is "+ ""file + table[] + >string" and the 'actual rule' is the rest, excluding the point but including the "-count:".

A 'rule declaration' defines the 'actual rule' to be of the type indicated by 'typer', to be known under the name 'rule tag' and to have the formal affixes given by its 'formal affix sequence'.

There are four types of rules: predicates, questions, actions and functions, each indicated by the corresponding 'typer' symbol. These four types arise from the fact that rules are differentiated on the basis of two mutually independent criteria:

a. a rule will <u>either</u> always succeed <u>or</u> be capable of failing, depending on the logical construction of the 'actual rule',
b. a rule, when succeeding, may or may not have side effects, again depending on the logical construction of the 'actual rule'.

These criteria are elaborated upon in 3.9..

A rule is a <u>predicate</u> if it can fail and has side effects (the restrictions on the construction of rules prevent these side effects from becoming effective if the rule fails).

A rule is a <u>question</u> if it can fail and has no side effects.

A rule is an <u>action</u> if it will always succeed and has side effects.

A rule is a <u>function</u> if it will always succeed and has no side effects.

The type of a rule is checked against the logical construction of the actual rule; if an action or function is found to be able to fail, an error message is given; in all other cases, if a discrepancy is found a warning is given.

Examples.
In each of the following examples the beginning of a rule is given, together with a comment indicating what the 'actual rule' does. From this explanation it follows why the rule was declared with the given type.

'predicate' digit + d>:
if the next symbol in the input file is a digit, it is delivered in 'd', the input file is advanced by one symbol (side effect) and 'digit' succeeds; otherwise it fails.

'question' is digit + >d:
if 'd' is a digit the rule succeeds, otherwise it fails.

'action' skip up to point:
the input file is advanced until the next symbol is a point.

'function' plus + >x + >y + sum>:
the sum of 'x' and 'y' is delivered in 'sum'.

3.2.2. Actual rules.

An actual rule mentions the variables local to it and specifies one or more alternatives.

Syntax:

```
actual rule: (local affix sequence), colon symbol, rule body.
rule body: alternative series; classification.
alternative series:
        alternative, (semicolon symbol, alternative series).
alternative:
        last member; member, comma symbol, alternative.
```

Example of an 'actual rule':

```
                - d:
        digit + d, times + res + 10 + res,
                plus + res + d + res, integer1 + res; +.
```

Here the 'local affix sequence' is "- d", one 'alternative' is "digit + d, times + res + 10 + res, plus + res + d + res, integer1 + res" and "+" is another. 'Member's are for example "plus + res + d + res" and "+".

When an 'actual rule' is executed (through a call (3.5.1.) of the 'rule' of which it is the 'actual rule'), the following takes place.
First space is made available on the run-time stack for the 'local affix'es, one location for each 'local affix' (see 3.3.3.). Subsequently its 'rule body' is executed.

The execution of a 'rule body' implies the execution of its 'alternative series' or of its 'classification'.

The execution of an 'alternative series' starts with a search to determine which of its 'alternative's applies in the present case. The applicable 'alternative' is the (textually) first 'alternative' that has a first 'member' that succeeds. Therefore, the first 'member' of the first 'alternative' is executed: if it succeeds, the first 'alternative' applies. Otherwise the first 'member' of the second 'alternative' is executed: if it succeeds, the second 'alternative' applies, etc. If none of the first 'member's succeeds, the 'alternative series' fails.
If the first 'member' is a 'terminator' (and then it is the only member), the 'alternative' at hand applies; if the 'terminator' succeeds, the 'alternative series' succeeds; if it fails the 'alternative series' fails and if it does neither, the question whether the 'alternative series' succeeds or fails will not arise.
The 'alternative' found applicable is then elaborated further. Its first 'member' has already been executed. Now the rest of its members are executed in textual order until one of two situations is reached:
Either all 'member's have succeeded, in which case the 'alternative series' succeeds as well,
or a member fails: the (textually) following 'member's in this 'alternative' will not be executed and the 'alternative series' fails.

If the 'alternative series' succeeded, the 'actual rule' succeeds; if it failed, the 'actual rule' fails.

After the result of the 'alternative series' has thus been assessed, the space for the 'local affix'es is removed from the stack.

Restrictions.
An 'alternative series' must satisfy the following restrictions:

a. If the first 'member' of an 'alternative' cannot fail (3.9.2.), this 'alternative' must be the last one. This restriction ensures that all 'alternative's can, in principle, be reached. Violation of this restriction causes an error message.

b. If an 'alternative' contains a 'member' that is backtrack-liable (see 3.9.3.) this 'member' may not, in the same 'alternative', be followed by a 'member' that can fail (see 3.9.2.).
This restriction ensures that the side effects of a 'member' cannot materialize if the 'member' fails; this in turn ensures that the tests necessary to determine the applicable 'alternative' in an 'alternative series' do not interfere with each other.
Violation of this restriction causes a warning. The user is urged to either reconsider the formulation of his problem or convince himself that the side effects caused have no ill consequences.

3.2.3. Members.

Members are the units of action in ALEPH. This action is a primitive operation, a call of a rule, or is again composed of other actions.

Syntax:
member: affix form; operation; compound member.
last member: member; terminator.

Example of a 'member':

```
(declaration sequence option - type - idf:
        declaration + type + idf, enter + type + idf,
                : declaration sequence option; +)
```

This 'member' is a 'compound member', "declaration + type + idf" is an 'affix form', ": declaration sequence option" is a 'last member', as is "+".

The notion 'last member' has been introduced in the syntax to ensure that a 'terminator' will occur only as the last 'member' in an 'alternative'.

3.3. Affixes.

Formal and actual affixes constitute the communication between the caller of a 'rule' and the 'rule' called. Local affixes are a means for creating variables that are local to a given 'rule body'.

3.3.1. Formal affixes.

Syntax:
formal affix sequence:
        formal affix, (formal affix sequence).
formal affix: formal affix symbol, formal.
formal: formal variable; formal stack; formal table; formal file.
formal variable: (right symbol), variable tag, (right symbol).
formal table: (field list pack),table tag, sub bus.
formal stack: sub bus, (field list pack), stack tag, sub bus.
sub bus: sub symbol, bus symbol.
formal file: quote symbol, quote symbol, file tag.

Example of a 'formal affix sequence':

```
+ ""file + table[] + >string
```

The 'formal affix sequence' defines the number and types of the 'formal affix'es of the 'rule' it belongs to.

A 'formal variable' is considered as data of type 'variable'. If the 'formal variable' starts with a 'right symbol' the variable has obtained a value from the calling rule; it is "initialized". Otherwise it has the attribute "uninitialized" at the beginning of each alternative in the 'actual rule'.
If the 'formal variable' ends in a 'right symbol' its value will be used by the calling rule: it must be "initialized" at the end of the 'actual rule'.

A 'formal stack' is considered as data of type 'stack'. If the 'field list pack' is absent, the 'formal stack' is supposed to have one 'selector': the tag of this 'selector' is the same as the tag of the 'formal stack' itself. For example, the 'formal affix' "[]list[]" has the same meaning as "[](list)list[]".

A 'formal table' is considered as data of type 'table'. If the 'field list pack' is absent, the 'formal table' is supposed to have one 'selector': the tag of this 'selector' is the same as the tag of the 'formal table' itself.

A 'formal file' is considered as data of type 'file'.

All 'tag's in a 'formal affix sequence' must be different. They must also be different from the 'rule tag' that precedes the 'formal affix sequence'.

3.3.2. Actual affixes.

'Actual affix'es occur in 'affix form's which cause the call of a 'rule'. Each 'actual affix' corresponds to a 'formal affix' of that 'rule'.

Syntax:
actual affix sequence: actual affix, (actual affix sequence).
actual affix: actual affix symbol, actual.
actual: source.

Example of an 'actual affix sequence':

```
+ 511 + /?/ + alpha + betaxgamma[p] + <>list
```

In this example "511" is an 'integral denotation', "/?/" is a 'character denotation', "alpha" is a 'tag', "betaxgamma[p]" is an 'element' and "<>list" is a 'calibre'.

'Actual affix'es derive their exact meanings from the corresponding 'formal affix'es. The interrelations are discussed in 3.5. (affix forms) and in 3.4. (transports).

3.3.3. Local affixes.

Syntax:
local affix sequence: local affix, (local affix sequence).
local affix: local affix symbol, local variable.
local variable: variable tag.

Example of a 'local affix sequence':

- count

A 'local variable' is considered as data of type 'variable'. Space for this 'variable' is reserved on the run-time stack upon entry of the 'actual rule' or 'compound member' of which it is part. This space is removed on exit from that 'actual rule' or 'compound member'.
A 'local variable' has the attribute "uninitialized" at the beginning of each 'alternative' of the 'actual rule' or 'compound member'. Its attribute must be "initialized" at the end of at least one 'alternative' that does not end in a 'jump'.

All 'tag's in a 'local affix sequence' must be different. Furthermore, all 'tag's in a 'local affix sequence' L must be different from:

a. all the 'label's, if any, and all 'tag's in the 'local affix sequence's, if any, of all the 'compound member's, if any, in which L is contained,
b. the 'rule tag' and all 'tag's in the 'formal affix sequence', if any, of the 'rule declaration' in which L occurs.

3.4. Operations.

Syntax:
operation: transport; identity; extension.

transport: source, destination sequence.
source: integral denotation; character denotation; tag;
        element; limit.
integral denotation: (integral denotation), digit.
character denotation: absolute symbol, character, absolute symbol.
element: (tag, of symbol), tag, sub symbol, source, bus symbol.
destination sequence: destination, (destination sequence).
destination: to token, tag; to token, element.
to token: minus symbol, right symbol.

identity: source, equals symbol, source.

extension: of symbol, field transport list, of symbol, tag.
field transport list: field transport,
             (comma symbol, field transport list).
field transport: source, selector destination sequence.
selector destination sequence: selector destination,
             (selector destination sequence).

selector destination: to token, selector.

Example of a 'transport':

```
pnt -> selxlist[q] -> offset -> orsxlist[offset]
```

Example of an 'identity':

```
ectxlist[pnt] = nil
```

Example of an 'extension':

```
x pnt -> sel, nil -> ect -> ors x list
```

3.4.1. Transports.

A 'transport' can be considered as a 'function', i.e., it has no (inherent) side effects and will always succeed.

Its execution starts with the evaluation of its textually first 'source'.

A 'source' is evaluated as follows.

If the 'source' is an 'integral denotation', its value is the numerical value of the sequence of 'digit's, considered as a number in decimal notation.

If the 'source' is a 'character denotation', its value is the numerical value of the 'character' in the code used.

If the 'source' is a 'tag', it must be the 'tag' of a global constant or of a (global, formal or local) variable. In each case the value of the 'source' is the value of the constant or variable. Moreover, in case of a formal or local variable it must have the attribute "initialized".

If the 'source' is an 'element', its value is determined as follows (see also 4.1.5. and 4.1.6.). The 'source' between the 'sub symbol' and the 'bus symbol' is evaluated and its value is called P. The 'tag' in front of the 'sub symbol' must be the tag of a (global or formal) stack or table. We now consider the block in this stack or table that has an address equal to P (if no such block exists, there is an error); it is called B. Subsequently a selector is determined: if the 'of symbol' is present, the selector is the 'tag' in front of it; if the 'of symbol' is absent, the selector is the 'tag' in front of the 'sub symbol' (in other words: it is the textually first 'tag' in the 'element'). Now, the value of the 'element' is the value in the block B indicated by the selector.

If the 'source' is a 'limit', the tag of the 'limit' must be the 'tag' of a (global or formal) stack or table. The value of the 'min limit' ('max limit', 'calibre') is the value of the "min limit" ("max limit", "calibre") of the corresponding actual stack or table. For these values see 4.1.7..

The value of the textually first 'source' is called V. Now the first 'destination' of the 'transport' is determined and V is put in the location indicated. Next its second 'destination', if present, is determined and V is put in the location indicated, and so forth.

A 'destination' is determined as follows.
If the 'destination' is a 'tag', this 'tag' must identify a global, formal or local variable. In this case the value is put in the location of the variable. If the 'destination' is the 'tag' of a formal or local variable, this variable has the attribute "initialized" in the rest of the 'member's in the 'alternative' in which the 'transport' appears.

If the 'destination' is an 'element', the 'source' between the 'sub symbol' and 'bus symbol' is evaluated and its value is called P. The 'tag' in front of he 'sub symbol' must be the tag of a global or formal stack. We now consider the block in this stack that has an address equal to P (if no such block exists, there is an error); it is called B. Subsequently a selector is determined: if the 'of symbol' is present, the selector is the 'tag' in front of it. Otherwise it is the 'tag' in front of the 'sub symbol'. (As an example, list[p] is equivalent to list x list[p].) The value is now put in the location in block B that is identified by the selector.

Examples:

```
0 -> cnt -> res           $ now "cnt" and "res" are both zero

p -> list[q] -> q         $ the value of "p" is put in the location
                          $ indicated by "listxlist[q]" and in
                          $ (the location of) q

p -> q -> list[q]         $ the value of "p" is put in (the location
                          $ of) "q" and in the location indicated
                          $ by "listxlist[q]" which is now the same
                          $ as "listxlist[p]"

list[p] -> p -> list[p] $ the value of "listxlist[p]" is put in
                          $ "p" and then put in "listxlist[p]" using
                          $ the new value of "p", with the result
                          $ that now "listxlist[p]" contains a
                          $ pointer to itself
```

3.4.2. Identities.

An 'identity' can be considered as a 'question', i.e., it has no side effects and may either succeed or fail.
Both its 'source's are evaluated as described above. If the two values are numerically equal the 'identity' succeeds, otherwise it fails.
If the values represent numerical results the 'identity' tests equality. If the values represent pointers to blocks in lists or tables, the 'identity' tests whether the two blocks pointed at are the same, not whether they are equal (as this might imply complicated comparison criteria).

3.4.3. Extensions.

An 'extension' can be considered as an 'action', i.e., it has side effects and will always succeed.
The 'tag' after the second 'of symbol' must be that of a (global or formal) stack, and the 'tag's that appear as 'selector destination' in the 'field transport list' must be selectors of that stack.
First the stack is extended to the right with one block of empty locations (whence the name "extension"); the number of locations in this block is equal to the calibre of the stack.
Subsequently the first 'source' in the 'field transport list' is evaluated as described in 3.4.1.. Its value is put in the location in the block just added that is identified by the first selector in the 'selector destination sequence', then that value is put in the location identified by the second selector (if present), and so on. Next, the second 'source' is evaluated (if present) and its value put in the location(s) indicated by its 'selector destination sequence', and so on.
No more than one value may be put in a given location; at the end of the 'extension' all locations in the added block must have been given a value; if the stack is formal, the calibre of the actual stack must be equal to that of the formal stack.

Example: given a stack "lst" declared as [](sel,ect,ors)lst:
then the 'extension'

```
        x 3 -> ect, 5 -> sel -> ors x lst
```

would add the block (5, 3, 5) to "lst" and ">>lst" would be 3 higher than it was before.

3.5. Affix forms.

Syntax:
affix form: rule tag, (actual affix sequence). $ see 3.3.2.

Example:

string elem + tbl + str + cnt + symb

When an 'affix form' is executed, the rule identified by the 'rule tag' in the 'affix form' is called, as follows.

Relationships are set up between the 'actual affix'es as supplied by the 'affix form' and the 'formal affix'es as supplied by the 'rule declaration'. The correspondence between actual and formal affixes is decided from their order: the first actual corresponds to the first formal, the second actual to the second formal, and so on. The number of actuals must be equal to the number of formals.

The 'actual' corresponding to a 'formal table' must be a 'tag' which is a (global or formal) stack or a (global or formal) table. All actions performed on the 'formal' are executed directly on the 'actual'. If the 'formal' has a calibre larger than 1 the calibre of the 'formal' and 'actual' must be equal; the names of the selectors may differ. If the calibre of the 'formal' is 1, no match is required. Regardless of mismatches, the value delivered by the 'calibre' ("<>tag") is the calibre of the global stack or table to which the 'formal table' corresponds, directly or indirectly.

The 'actual' corresponding to a 'formal stack' must be a 'tag' that is a (global or formal) stack. All actions performed on the 'formal' are executed directly on the 'actual'. If the 'formal' has a calibre larger than 1 the calibre of the 'formal' and 'actual' must be equal; the names of the selectors may differ. If the calibre of the 'formal' is 1, no match is required. Regardless of mismatches, the value delivered by the 'calibre' ("<>tag") is the calibre of the global stack to which the 'formal stack' corresponds, directly or indirectly.

The 'actual' corresponding to a 'formal file' must be a 'tag' that is a (global or formal) file. All actions performed on the 'formal' are executed directly on the 'actual'.

First the copying part of the affix mechanism is put into operation: for each 'formal' that is a 'formal variable' starting with a 'right symbol', a 'transport' is executed with the 'actual' as a 'source' and the 'formal' as a 'destination'.

Subsequently, the 'actual rule' in the rule identified above is executed (see 3.2.2.). If this 'actual rule' succeeds, the 'affix form' succeeds; if it fails, the 'affix form' fails.

If the 'affix form' succeeds the restoring part of the affix mechanism will be executed: for each 'formal' that is a 'formal variable' ending in a 'right symbol', a 'transport' is executed with the 'formal' as a 'source' and the 'actual' as a 'destination', in the order in which the affixes appear.

Example:

Suppose the following 'rule's are defined:

```
'question' if a: $ some question $.
'question' if b: $ another question $.
'function' give value 1 + n>: 1 -> n.
'function' give value 2 + n>: 2 -> n.
'action' use value + >n: print + n.
'action' print + >n:
        $ some actual rule that prints the value of n $.
```

In the 'actual rule'

```
- loc: if a, give value 1 + loc, use value + loc, print + loc;
       if b, give value 2 + loc, use value + loc.
```

"loc" is "uninitialized" at the colon and likewise at the first comma, "initialized" at the second comma because of the restoring done by the call of "give value1", and remains "initialized" until the end of the 'alternative'. Its value can be copied over to 'use value' and 'print'. At the beginning of the second alternative it is still "uninitialized" (<u>still</u> "uninitialized", not <u>again</u> "uninitialized", since, if the beginning of the second 'alternative' is reached, the initialization in the previous 'alternative' will not have taken place). It remains "uninitialized" until the call of "give value 2" after (and by) which it is "initialized". Its subsequent application in "use value" is correct.

The 'actual rule'

```
- loc: if a, use value + loc, give value 1 + loc, print + loc.
```

is incorrect. "loc" is still "uninitialized" at the first comma and is then used as a source in the copying done by the call of "use value".

3.6. Terminators.

Syntax:

```
terminator: jump; exit; success symbol; failure symbol.
jump: repeat symbol, rule tag.
exit: exit symbol, expression.
```

Examples of 'terminator's:

```
: order
'exit' 16
+
-
```

Jumps.
The 'tag' after the 'repeat symbol' may be the 'tag' of the 'actual rule' in which this 'jump' occurs or the 'tag' of (one of) the 'compound member'(s) in which this 'jump' occurs.

A 'jump' to the 'tag' of an 'actual rule' is an abbreviated notation of a call to that rule, with actual affixes that correspond to the original actual affixes. The abbreviation is only allowed if, after the execution of the call, no more members in the rule can be executed. This condition ensures that there will be no need for the "recursive call" mechanism to be invoked.

Example:

The rule:

```
'action' bad1: a, (b; :bad1), c; +.
```

is incorrect: after returning from ":bad1" "c" will be executed. If the ",c" is removed, the rule is correct. Likewise the rule:

```
'question' bad2: (a, b, :bad2); c.
```

is incorrect: after unsuccessful returning from ":bad2" "c" will be executed. If the parentheses are removed, the rule is correct.

A 'jump' to the 'tag' of a 'compound member' C causes this 'compound member' to be re-executed. The precise meaning can be assessed by decomposing (see 3.7.) the 'actual rule' until C turns into an 'actual rule'. Then the above applies.

Exits.
The execution of an 'exit' causes the entire program to be terminated. The termination state is equal to the value of the 'expression' in the 'exit'. An 'exit' is a 'function'.

Success and failure symbols.
The execution of a 'success symbol' always succeeds, the execution of a 'failure symbol' always fails. They have no additional effects.

3.7. Compound members.

'Compound member's serve to group a series of 'alternative's into a single 'member'.

```
Syntax:
compound member:
        open symbol, (local part, colon symbol),
                rule body, close symbol.
local part: rule tag, (local affix sequence); local affix sequence.
```

Example:

```
(order - n: less + y + x, x -> n, y -> x, n -> y;
            x = y, get next int + x, : order; +)
```

A 'compound member' is an abbreviated notation for the call of a rule. Loosely speaking, the rule that is called has the same meaning as the 'rule body' of the 'compound member' and has all its non-globals as formal affixes. The call then calls that rule with these non-globals as actual affixes. The following statement expresses this more precisely.

A 'rule declaration' for the rule that is called can be derived from the 'compound member' in the following way.

a. the 'open symbol' and 'close symbol' are removed,
b. a 'point symbol' is placed after the 'rule body',
c. if the 'local part, colon symbol' is absent, a 'colon symbol' is placed in front of the 'rule body',
d. if the 'rule tag' is missing, a 'rule tag' is placed in front that produces a 'tag' that is different from any other tag in the program,
e. a 'formal affix sequence', if necessary, is constructed (see below) and inserted after the 'rule tag',
f. the "type" of the 'rule body' is determined (see 3.9.) and the corresponding 'typer' (see 3.2.1.) is placed in front of the 'rule tag'.

The 'formal affix sequence' mentioned in e. above is constructed as follows:

a. a list is made of all tags in the 'rule body' that do not refer to global items and do not occur in the 'local affix sequence', if present,
b. if the list is empty the 'formal affix sequence' is not necessary,
c. for each tag in the list, if the corresponding item
1. is used as a "source" (either directly or through a call) and is used as a "destination" (either directly or through a call), it is entered into the 'formal affix sequence' preceded and followed by a 'right symbol',
2. is used as a "source" (either directly or through a call), it is entered into the 'formal affix sequence' preceded by a 'right symbol',

3. is used as a "destination" (either directly or through a call), it is entered into the 'formal affix sequence' followed by a 'right symbol',
4. is used as a 'stack tag' (or 'table tag'), it is entered into the 'formal stack' (or 'formal table') with the same 'field list pack' as that of the corresponding (formal or actual) stack (or table),
5. is used as an 'actual affix' where a file is required, it is entered into the 'formal affix sequence' as a 'formal file',

d. the items in the 'formal affix sequence' are preceded by 'formal affix symbol's.

Example:

For the 'compound member'

```
        (a[p] = 0, 0 -> a[q]; plus + m + p + q)
```

where "m" is global, the 'rule declaration' runs:

```
'action' zzgrzl + [](a)a[] + >p + >q>:
        a[p] = 0, 0 -> a[q]; plus + m + p + q.
```

and the call is:

```
        zzgrzl + a + p + q
```

This also implies that, if a 'compound member' fails, the changes it made to formal and local variables do not become effective. Compare

```
        0 -> n, ((1 -> n, -);
                 n = 0, do something)
with
        0 -> n, (spoil and fail + n;
                 n = 0, do something)
where
        'question' spoil and fail + n>: 1 -> n, -.
```

Both cases behave in exactly the same way: the rule "do something" will be called.

The 'tag' in the 'label' in a 'compound member' C must be different from:

a. the 'rule tag's, if any, and all the 'tag's in the 'local affix sequence's, if any, of all the 'compound member's, if any, in which C occurs,

b. the 'rule tag' and all the 'tag's in the 'formal affix sequence', if any, of the 'rule declaration' in which C occurs.

3.8. Classifications.

A 'classification' is similar to an 'alternative series' in that both specify a series of 'alternative's only one of which will eventually apply. The difference is twofold: in a 'classification' exactly 1 'alternative' applies (as opposed to 1 or 0 in an 'alternative series'), and the choice of the pertinent 'alternative' is based on a single runtime value (as opposed to the successive execution of first members). 'Classification's allow fast selection of 'alternative's at the cost of a less versatile selection mechanism.

Syntax:

```
classification: classifier box, class chain.
classifier box: box symbol, source, box symbol.
class chain: class, semicolon symbol, class chain; last class.
class: area, comma symbol, alternative.
area: sub symbol, zone series, bus symbol.
zone series: zone, (semicolon symbol, zone series).
zone: (expression), up to symbol, (expression); expression; tag.
last class: class; alternative.
```

Example 1:

```
(n: get + char,
        (= char =
         [/0/ : /9/], dgt -> type;
         [/a/ : /z/; /a/ + cap : /z/ + cap], ltr -> type;
         [/+/; /-/; /x/; ///], op -> type;
         [0; 127], : n;
         [ 0 : 127 ], err -> type))
```

Example 2:

```
= tag =
[var decl],   handle variable + tag;
[macro decl], handle macro call + tag;
[rout  decl], handle routine call + tag;
              handle bad tag + tag
```

The execution of a 'classification' starts with the evaluation of the 'source' in its 'classifier box'. The resulting value is called V. Now the 'area's in the 'classification' are searched in textual order for an 'area' in which V belongs. If such an 'area' is found, the 'alternative' following it applies and is executed (see 3.2.2.). If there is no such 'area', the 'last class' must be an 'alternative', which then applies and is executed. Otherwise there is an error.

V belongs in a given 'area' if it belongs in any of its constituent 'zone's. Whether V belongs in a given 'zone' is determined as follows.

If the 'zone' is an 'expression' E then V belongs in that 'zone' if it is equal to the value of E.
If the 'zone' contains an 'up to symbol' it is designated by two boundaries. The left boundary L is the value of the 'expression' in front of the 'up to symbol' or, if it is missing, the value of "min int". The right boundary R is the value of the 'expression' after the 'up to symbol' or, if it is missing, the value of "max int". V belongs to the given 'zone' if $L \leq V \leq R$.

If the 'zone' is a 'tag', this 'tag' must correspond to a global (not formal) stack or table. V belongs in this 'zone' if it is an address in the indicated stack or table.

'Area's may coincide partially or totally; the textually first 'area' takes precedence. A warning is issued if the total of the 'area's do not cover the complete range from "min int" to "max int".

When all 'zone's consist of 'expression's the exact size and location of all 'zone's is known at compile time; this information will be utilized by the compiler. When all 'zone's consist of 'tag's (identifying 'list's and 'table's) the relative positions of all 'zone's are known at compile time (although their exact sizes are not); again this information is utilized by the compiler. In mixed cases slightly less efficient code may result.

Example:
If it is known that "ind" takes only values between 0 and 4, very efficient code can be obtained by:

```
= ind =
[0], zero; [1], one; [2], two; [3], three; [4], four
```

as the applicable 'alternative' is selected directly; slightly less efficient code would be generated for:

```
= ind =
[0], zero; [1], one; [2], two; [3], three; four
```

as this necessitates two comparisons to check for the last 'alternative'. On the other hand, the latter is defined for all values of "ind" whereas the former is only defined for $0 \leq ind \leq 4$.
Again, less efficient code is obtained by:

```
        ind = 0, zero;
        ind = 1, one;
        ind = 2, two;
        ind = 3, three;
        ind = 4, four
```

as this implies a sequence of comparisons. In contrast to the first example the above 'rule body' is defined for all values of "ind" and as opposed to the second example it fails when "ind" does not have a value in the range 0 through 4.

A 'classification' can fail if at least one of its 'alternative's can fail, it has side-effects if at least one of its 'alternative's has side-effects.

3.9. Criteria for side effects, failing and backtrack.

Where a list of conditions is given in this paragraph, the requirements for this list are fulfilled if at least one of the conditions is fulfilled.

3.9.1. Criteria for side effects.

In essence a 'rule' "has side effects" if it changes global information.

A 'rule' has side effects if its 'rule body' has side effects.

A 'rule body' (i.e., an 'alternative series' or a 'classification') has side effects if it contains at least one 'member' that has side effects.

A 'member' has side effects if

1. it is an 'affix form' that has side effects,
2. it is a 'transport' that has side effects,
3. it is an 'extension',
4. it is a 'compound member' the 'rule body' of which has side effects.

An 'affix form' has side effects if

1. the 'rule' called is of type 'action' or 'predicate',
2. the restoring part of the affix mechanism (see 3.5.) causes a 'transport' that has side effects.

A 'transport' has side effects if (one of) its destination(s) is a global variable or a (stack) element.

3.9.2. Criteria to determine whether a 'member' or 'rule body' can fail.

A 'member' can fail if

1. it is an 'affix form' the 'rule' of which is of type 'predicate' or 'question',

2. it is an 'identity',
3. it is a 'compound member' the 'rule body' of which can fail,
4. it is a 'failure symbol' (-).

A 'rule body' can fail if its 'alternative series' or 'classification' can fail.

An 'alternative series' can fail if
1. the first 'member' of its last 'alternative' can fail,
2. it contains an 'alternative' consisting of more than one 'member', in which a 'member' other than the first can fail.

A 'classification' can fail if it contains a 'member' that can fail.

3.9.3. Criteria for backtrack-liability.

A 'member' is backtrack-liable if it has side effects.

4. Data.

The basic way of representing information in ALEPH is through integers. There are four integer-based data types:
integers (constants),
locations that contain integers (variables),
ordered lists of integers (tables), and
ordered lists of locations that contain integers (stacks).
Integers used in data declarations can be given in the form of expressions.

The basic way of routing information into and out of the program is through files. There are two types of files:
charfiles, files containing only integers that correspond to characters, and
datafiles, files containing pointers to prescribed stacks and tables and/or integers in a prescribed range.

There are three primitive actions on integer-based data: transports, identities and extensions. Additional integer handling can be done through externals.

There are no primitive actions on files: all file handling is done through externals.

Syntax of 'data declaration':

```
data declaration:
        constant declaration; variable declaration;
        stack declaration; table declaration;
        file declaration.
```

4.1. Integer-based data.

Since all integer-based data can be initialized through expressions, these will be treated first.

4.1.1. Expressions.

```
Syntax:
expression:
        (plus minus), term; expression, plus minus, term.
term: (term, times by), base.
base: plain value; expression pack.
plain value:
        integral denotation; character denotation; tag; limit.
integral denotation: (integral denotation), digit.
character denotation:
        absolute symbol, character, absolute symbol.
expression pack: open symbol, expression, close symbol.
plus minus: plus symbol; minus symbol.
times by: times symbol; by symbol.
```

Examples:
-3 + 5 × byte size
line width/2
((/e/ + 1) × char size + /n/ + 1) × char size + /d/ + 1

The value of an 'expression' is the integral value that results from evaluating the 'expression' according to the standard rules of algebra. The result of an integer division n = p/q (q ≠ 0) is a value n such that p ≥ n × q and p - n × q is as small as possible (so, 7/3=2, 7/(-3)=-2, (-7)/3=-3 and (-7)/(-3)=3). The value of an 'integral denotation' is the numerical value of the sequence of 'digit's, considered as a number in decimal notation. The value of a 'character denotation' is the numerical value of the 'character' in the code used. If a 'plain value' is a 'tag', it must be a 'constant tag', defined in a 'constant declaration' or in a 'pointer initialization' (a 'tag' defined in a user-defined 'external constant declaration' cannot be used as a 'plain value'). Its value may not depend on the 'expression' in which the 'constant tag' occurs. That is,

'constant' p = q, q = 2 - p.
is not allowed.

The 'tag' of a 'limit' (see 4.1.7.) in an 'expression' must be the 'tag' of a (global) table, i.e., limits of stacks cannot be used in expressions.

4.1.2. Constants.

A constant consists of a 'tag' and an integral value. The relation between tag and value is set up through a 'constant declaration' and cannot be changed afterwards.

```
Syntax:
constant declaration:
        constant symbol, constant description list,
        point symbol.
constant description list:
        constant description,
        (comma symbol, constant description list).
constant description: tag, equals symbol, expression.
```

Example:
'constant' mid page = line width/2, line width = 144.

Constants can be used in 'expression's and in 'source's.

4.1.3. Variables.

A variable consists of a 'tag' and a location, the location may or may not contain a value. If it contains a value the variable 'has' that value. The contents of a location may be changed. Once a location has obtained a value it can never become empty again.

A global variable is declared in a 'variable declaration'.

A formal variable originates from a 'formal affix sequence'.

A local variable originates from a 'local affix sequence'.

Syntax of 'variable declaration':

```
variable declaration:
        variable symbol, variable description list, point symbol.
variable description list:
        variable description,
        (comma symbol, variable description list).
variable description: tag, equals symbol, expression.
```

Examples:

```
'variable' tag pnt = nil, median code = (<<code + >>code)/2.
'variable' line cnt = 0, page cnt = 0.
```

For each 'variable description' a location is made available, tagged with the 'tag' and filled with the value of the 'expression'.

Variables can be used in 'source's and in 'destination's. They cannot be used in 'expression's.

4.1.4. The address space.

In addition to constants and variables there exist lists of constants ("tables") and lists of variables ("stacks"). Stacks and tables together are called "lists". The items in these lists are identified by unique addresses which are represented by integral values. These values range from a very large negative number to a very large positive number: this range is called the "address space".

The "lists" are described as running from left to right.

Example:
On a 16-bit machine the address space could be thought of as a list of 2↑16 (65536) locations, the addresses of which run from -2↑15 (-32768) at the left to 2↑15-1 (32767) at the right. The question whether all these locations actually exist in memory is at this point immaterial: it is only the addressability of a location that is secured here.

For a given program the address space is divided into chunks, one for each list. Consequently, an address uniquely identifies not only a location but also the list it belongs to. A chunk of address space belonging to a list is called its "virtual address space". Generally only a part of the virtual address space is in use: this part is called the "actual address space". From the language specifications it follows that an actual address space is always a contiguous list of locations or values.

The user has no direct control over the way in which the address space is divided and addresses are assigned. This is done as follows:

a. One address is set aside and the 'external constant' "nil" is given its value. Consequently "nil" will never address any user item.
b. For each table T the size of its actual address space is calculated from its 'filling list' and T is given a virtual address space of exactly the same size. The right-most address is called the "max limit" of T, the left-most address minus one plus the "calibre" of T is called the "min limit" of T.
c. For each stack with an 'absolute size' a virtual address space of that size is reserved.
d. The remainder of the virtual address space is distributed over the rest of the stacks, proportionally to their 'relative size's.
e. For each stack S the right-most address in its virtual address space is called "virtual max limit", the left-most address in its virtual address space minus one plus the "calibre" of S is called "virtual min limit"; the size of its actual address space is calculated from its 'filling list' and the actual address space is positioned at the left end in the virtual address space. The "max limit" of S is made equal to the right-most address in the actual address space; the "min limit" of S is made equal to the "virtual min limit".
   If the actual address space has length zero, the "max limit" of S is equal to the "min limit" minus the "calibre" of S.

example:
Suppose a virtual address space of 5 bits, i.e. the addresses range from -16 to 15. If the following declarations (see 4.1.5. and 4.1.6.) occur in the program:

```
'table' powers = (1, 10, 100, 1000).
'stack' [= 5 =] digits = (0),
        [ 30  ] stack,
        [ 50  ] (num, denom) rationals =
                ((355, 113):pi, (191, 71):e).
```

the virtual address space could have the following layout:

address	contents	belongs to	selector	pointer
-16	not actual	- -	- -	nil
-15	1	powers	powers	<<powers
-14	10	"	"	
-13	100	"	"	
-12	1000	"	"	>>powers
-11	0	digits	digits	<<digits, >>digits
-10	not actual	"	"	
-9	" "	"	"	
-8	" "	"	"	
-7	" "	"	"	>>stack
-6	" "	stack	stack	<<stack
-5	" "	"	"	
-4	" "	"	"	
-3	" "	"	"	
-2	" "	"	"	
-1	" "	"	"	
0	" "	"	"	
1	" "	"	"	
2	355	rationals	num	
3	113	"	denom	<<rationals, pi
4	191	"	num	
5	71	"	denom	>>rationals, e
6	not actual	"	num	
7	" "	"	denom	
8	" "	"	num	
9	" "	"	denom	
10	" "	"	num	
11	" "	"	denom	
12	" "	"	num	
13	" "	"	denom	
14	" "	"	num	
15	" "	"	denom	

(For the notation used see 4.1.5. through 4.1.7).

ALEPH allows the user to extend a stack towards the right (raising "max limit") through an 'extension' (3.4.3.); to remove items from the right of a stack through a call of "unstack", "unstack n", "scratch" or "delete" (5.2.4.) after which the discarded address space can be reclaimed (but not the values in it) through an 'extension'; and to remove items from the left of a stack through a call of "unqueue" or "unqueue n" (5.2.4.) after which the discarded address space is irrevocably lost.

Through the use of these features a stack can be operated in stack fashion (add to "top"/remove from "top") or in queue fashion (add to "top"/remove from "bottom"). Queue-operation consumes virtual address space but in most implementations virtual address space will be virtually unlimited.

The virtual and actual address space of a table are fixed (and equal) for the duration of the program.

Usually an actual address space corresponds to a physical space that is in the physical memory of the computer used. The physical space is completely invisible to the user except perhaps in efficiency considerations. Parts of it may be in main memory, managed by some re-allotment scheme, parts of it may be on background memory.

If the 'tag' of a table is not used in other constructs than 'limit' and 'zone', the values in the table will never be accessed, and no physical space needs to be assigned to this table.

Example:
The 'table declaration':

```
'table' dummy = (0:nil1, 0:nil2, 0:nil3).
```

where the tag "dummy" does not occur anywhere else in the program, declares 3 more nil-like constants and no physical space needs to be reserved.

4.1.5. Tables.

Tables originate from 'table declaration's.

Syntax:

```
table declaration:
        table symbol, table description list, point symbol.
table description list:
        table description, (comma symbol, table description list).
table description: table head, equals symbol, filling list pack.
table head: (field list pack), tag.
field list pack: open symbol, field list, close symbol.
field list: field, (comma symbol, field list).
field: selector chain.
selector chain: selector, (equals symbol, selector chain).
selector: tag.

filling list pack: open symbol, filling list, close symbol.
filling list: filling, (comma symbol, filling list).
filling: single block; compound block; string filling.
single block: expression, (pointer initialization).
compound block:
        expression list proper pack, (pointer initialization).
```

```
pointer initialization: colon symbol, tag.
expression list proper pack:
        open symbol, expression list proper, close symbol.
expression list proper:
        expression, comma symbol, expression list.
expression list: expression, (comma symbol, expression list).
string filling: string denotation, (pointer initialization).
string denotation:
        quote symbol, (string item sequence), quote symbol.
string item sequence: string item, (string item sequence).
string item: non quote item; quote image.
quote image: quote symbol, quote symbol.

Examples:
'table' messages =
        ("tag undefined": bad tag,
         "wrong number of parameters": wr par,
         "quote "" where not allowed":bad qu).
'table' hexadec =
        (/0/,/1/,/2/,/3/,/4/,/5/,/6/,/7/,
         /8/,/9/,/a/,/b/,/c/,/d/,/e/,/f/).
'table' (wind, next) four winds =
        ((north wind, east): north,
         (east wind, south): east,
         (south wind, west): south,
         (west wind, north): west).
```

4.1.5.1.

A "table" is a sequential list of integral values. For referencing purposes these values are numbered sequentially. The numbers which can be used as addresses are chosen by the compiler and are unique for the given table, i.e., no two integral values in tables have the same address. The right-most item in the table has the largest address, which is known as the "max limit" of the table. The left-most item has the smallest address, the smallest address minus one plus the calibre is known as the "min limit" of the table. Consequently the number of values in the table is "max limit" - "min limit" + "calibre".

If the 'field list pack' is missing, a 'field list pack' of the form:

open symbol, table tag, close symbol

where 'table tag' is the 'tag' of the table, is supposed to be present. For example:

"'table' messages" means "'table' (messages) messages".

4.1.5.2.

The 'field list pack' and the 'filling list'.

The following applies to tables and stacks alike.

All 'tag's in a 'field list pack' must differ one from another.

The "calibre" C of a list is the number of 'field's in the 'field list pack'. The list is considered to be subdivided into blocks of length C; this implies that "max limit" - "min limit" is an integral multiple of C. The address of the right-most item in a block is considered the address of that block. Each value in a block can be referenced through a 'selector': the 'field's in the 'field list pack' correspond, in that order, to the values in the block. A 'field' is indicated by one of its 'selectors'.

The values in the list are specified in the 'filling list pack'. Each 'filling' in the 'filling list pack' corresponds to one or more blocks in the list: the first block produced by the 'filling list pack' corresponds to the left-most block in the list (the most primitive block if the list is a stack), and so on.

If the 'filling' is a 'single block', the calibre of the list must be 1. It gives rise to one block; the value in the block is the value of the expression. If a 'pointer initialization' is present the 'constant tag' herein is defined as having the value of the address of the block.

If the 'filling' is a 'compound block', the number of 'expression's in it must be equal to the calibre of the list. The values in the block are the values of the expressions. If a 'pointer initialization' is present the 'constant tag' herein is defined as having the value of the address of the block.

If the 'filling' is a 'string denotation', the calibre of the list must be 1. It gives rise to one or more blocks of one value each that describe the given string in a machine-dependent way. If a 'pointer initialization' is present the 'constant tag' herein is defined as having the value of the largest address in the generated list of blocks.

The string denoted by a 'string denotation' is the 'string item sequence' in which each 'quote image' is replaced by a 'quote symbol'. A 'non quote item' is any symbol with the exception of 'quote symbol'. Spaces are considered symbols, new-line controls are not, since the dividing into lines is done through the charfile-handling externals (see 5.2.5.).

Example 1:
The 'table declaration' for "four winds" (example 3 above) gives rise to the following list:

address	selector	value
	wind	north wind
north	next	east
	wind	east wind
east	next	south
	wind	south wind
south	next	west
	wind	west wind
west	next	north

and "wind × four winds [next × four winds [west]]" has the value "north wind".

Example 2:
The 'table declaration'

'table' strings = ("abcdefg": letters, "01234": digits)

could in some version on some computer generate:

address	selector	value			
	strings	13	14	15	16
	"	17	20	21	00
letters	"	00	07	00	02
	"	01	02	03	04
	"	05	00	00	00
digits	"	00	05	00	02

A 'table tag' can be used in a 'table element' or a 'limit', or as an 'actual' in an 'affix form', or to indicate a 'zone' in a 'classification'.

4.1.6. Stacks.

Stacks originate from 'stack declaration's.

Syntax:
```
stack declaration:
        stack symbol, stack description list, point symbol.
stack description list:
        stack description, (comma symbol, stack description list).
stack description: stack head, (equals symbol, filling list pack).
stack head: size estimate, (field list pack), tag.
size estimate: relative size; absolute size.
relative size: sub symbol, expression, bus symbol.
absolute size:
        sub symbol, box symbol, expression, box symbol, bus symbol.
```

Examples:

```
'stack' [= line width =] (char) print line.

'stack' [40] (tag pnt, left, right) idf list =

$ the following 'filling list pack' describes a binary tree
$ containing the standard identifiers of ALGOL 60.

        ((exp    st, cos, sign  ): exp,
         (abs    st, nil, arctan): abs,
         (arctan st, nil, nil   ): arctan,
         (cos    st, abs, entier): cos,
         (entier st, nil, nil   ): entier,
         (ln     st, nil, nil   ): ln,
         (sign   st, ln , sin   ): sign,
         (sin    st, nil, sqrt  ): sin,
         (sqrt   st, nil, nil   ): sqrt).
```

A "stack" is a (possibly empty) sequential list of locations that contain integral values. The structure of this list and its addressing scheme is parallel to that of a table. The initial values in the locations are determined by the 'filling list pack' in a way analogous to that used for tables. The "max limit" is equal to the address of the right-most location, the "min limit" is equal to the address of the left-most location minus one plus the "calibre" of the stack. Again these values are chosen by the compiler and are unique to the given stack.

The values in the locations in a stack can be altered by transporting (3.4.) a value into an 'element' of that stack. For ways of changing the size of a stack, see 4.1.4..

A 'stack tag' can be used in a 'stack element', a 'limit' or an 'extension', or as an 'actual' in an 'affix form', or to indicate a 'zone' in a 'classification'.

4.1.7. Limits.

Syntax:
```
limit: min limit; max limit; calibre.
min limit: min token, tag.
max limit: max token, tag.
calibre: calibre token, tag.
min token: left symbol, left symbol
max token: right symbol, right symbol.
calibre token: left symbol, right symbol.
```

Examples:
```
<<stack, >>table, <>blocked
```

A 'min limit' ('max limit', 'calibre') has the value of the "min limit" ("max limit", "calibre") of the list identified by the 'tag'.
The value of a 'limit' is a constant in that it cannot be changed by an assignment. However, the 'min limit' and 'max limit' of a stack may change as a consequence of actions that change the size of that stack. The 'min limit' and 'max limit' of tables and the 'calibre's of all lists are invariable.

4.2. Files.

Files originate from 'file declaration's. They can be prefilled by the operating system (input files) or postprocessed by the operating system (output files) or both (I/O files) or neither (scratch files).

Syntax:
```
file declaration:
        file typer, file description list, point symbol.
file typer: charfile symbol; datafile symbol.
file description list:
        file description, (comma symbol, file description list).
file description:
        tag, (area), equals symbol,
        (right symbol), string denotation, (right symbol).
```

Examples:

```
'charfile' printer = "output">, backward cards = >"qelet,invert".

'datafile' tagfile[tag; link; 0:] = >"systags">,
        bin cards[0:4095] = "12row,bin">, overflow[:] = "‡??qxz".
```

A 'file description' declares a "file" of the type indicated by the 'file typer'. If the first 'right symbol' is present, the file is prefilled by the operating system (but it may still be empty); if the second 'right symbol' is present, the file will be postprocessed by the operating system (but it may be empty).
The (implementation-dependent) 'string denotation' must contain enough information to enable the operating system to manipulate the file in the desired way. It might for example contain: the name of a file control card, allocation information, the names of routines to do the prefilling and postprocessing, etc..

ALEPH contains no explicit file handling statements: all file handling is done through (standard) externals (see 5.2.5.). When a file is used for writing, each item offered must belong in the 'area' given in the 'file description'; when a file is used for reading, each item delivered will belong in the given 'area'. If no 'area' is supplied, the 'area' "[0:max char]" is assumed.
Files are read and written sequentially. They can be reset to the beginning of the file and be reread or rewritten. Some files allow "back-spacing", "back-lining" or "back-data-ing". The file ends after the last item written or else after the last item produced by the preprocessing.

4.2.1. Charfiles.

A "charfile" is a list of "line"s. A line consists of a control integer and a (possibly empty) sequence of characters. Characters are values in the 'area' "[0:max char]", control integers are values outside that 'area'. Four control integers are predefined in the compiler (see 5.2.5.): "new line", "same line", "rest line" and "new page". These control integers can be used by the pre- and post-processing to reconcile the system requirements with the ALEPH requirements. If the file is eventually postprocessed towards a printer, lines of the type "new line" will be printed on new lines, those of the type "same line" will be printed over the previous line and those of type "new page" will be printed on the first line of the next page; "rest line" serves administration purposes only. Analogous effects should be defined for other devices, as far as the analogy will stretch.

Example:
A file containing "a^b=b^a" would consist of two lines:

```
new line,  /a/,/^/,/b/,/=/,/b/,/^/,/a/
same line, / /,/ /,/ /,/_/.
```

The standard externals allow two ways of processing a charfile.

a. linewise: e.g., each call of "get line + char file + stack + cint" puts the next line on "stack" (the last item in the line is the last item in the stack) and yields the control integer in "cint". It will fail if there is no next line.
b. characterwise: e.g., each call of "get char + charfile + char" yields the next item from the "charfile" (control integers and characters equally). It will fail if there is no next item.

The 'area' in the 'file description' of a charfile pertains to the values of the characters only. If present, the 'area' must only specify values that belong in "[0:max char]", e.g. [0:1].

4.2.2. Datafiles.

A "datafile" is a list of "data-item"s. A data-item consists of an integer value and an indication about its meaning. This indication is either "numerical" in which case the integer value stands for itself, or is the name of a list in which case the integer value is an offset from the left end of that list.
A data-item is written on a datafile by a call of "'action' put data + ""file + >item + >type". The data-item is constructed from the item- and type-parameters and from the 'area' in the 'file description' in the following way.
If the type is "numerical", there must be a 'zone' in the area which is not the name of a list, such that the value belongs in that 'zone'. The data-item then consists of the value of "item" and the indication "numerical".
If the type is "pointer", the value of "item" must point into a list that is a 'zone' in the area. The data-item then consists of the offset from the left end of that list and the name of the list.

A data-item is read from a datafile by a call of "'predicate' get data + ""file + item> + type>". If there is still a data-item on the datafile, it is read and the "item" and "type" are reconstructed from it (see above). If there are no more data-items on the datafile, the predicate fails.

Datafiles can be used to transfer information from one ALEPH-program to another. Pointers to lists that are in different positions in both programs are adjusted automatically during the transfer.

Note: in practice it is not necessary to record the list name with every item. It is enough to have one bit per item and one translation table for the whole file.

Example:
Suppose the 'file declaration':

```
        'datafile' tag file[tag; list; 0:] = >"systags">.
```

Then "put data" for this file can be visualized as:

```
'action' put data + ""file + >item + >type:
$ for file = tagfile only $
type = pointer,
    (= item =
     [tag],  minus + item + <<tag + item,
                   write data item + item + tag name;
     [list], minus + item + <<list + item,
                   write data item + item + list name;
             error + bad item);
type = numerical,
    (= item =
     [0:], write data item + item + numerical;
             error + bad item);
error + bad type.
```

Here the (imaginary) "write data item + >val + >ind" would write a data-item consisting of "val" and "ind" on the file "tagfile".

5. Externals.

External rules, tables and constants can be used in the same way as internally declared rules, tables and constants. An external rule differs from an "internal" rule in that its body is not given in the program but is instead obtained from external sources. In the same way the values of external tables and constants are obtained from external sources. The necessary information can be supplied by the user through external means ("user" externals, section 5.1.) in which case the name of the item and some of its properties must be declared in the program, or it is supplied automatically by the compiler ("standard" externals, section 5.2.) in which case there is no explicit declaration at all.

5.1. User externals.

```
Syntax:
external declaration:
        external rule declaration;
        external table declaration;
        external constant declaration.
external rule declaration:
        external symbol, typer,
                external rule description list, point symbol.
external rule description list:
        external rule description,
                (comma symbol, external rule description list).
external rule description:
        tag, (formal affix sequence),
                equals symbol, string denotation.
external table declaration:
        external symbol, table symbol,
                external table description list, point symbol.
external table description list:
        external table description,
                (comma symbol, external table description list).
external table description:
        (field list pack), tag, equals symbol, string denotation.
external constant declaration:
        external symbol, constant symbol,
                external constant description list, point symbol.
external constant description list:
        external constant description,
                (comma symbol, external constant description list).
external constant description:
        tag, equals symbol, string denotation.
```

Example:

```
'external' 'function' convert to hash + t[] + >p + h> =
       "subr, convertt".
'external' 'table' conv 2 ebcdic = "addr, conv2ebc".
'external' 'constant' max ebcdic = "cons, maxebcdi".
```

An 'external rule description' defines a rule to be of the type given by the preceding 'typer', to be known internally under the name given by the 'tag' and externally by the 'string denotation', and to have affixes as shown by the 'formal affix sequence'. A call to such a rule will result in implementation-dependent actions (see "ALEPH Implementation", to be published by the Mathematical Centre, Amsterdam).

An 'external table description' defines a table to be known internally under the name given by the 'tag' and externally by the 'string denotation', and to have the selectors given by the 'field list pack'. An application of this table will result in implementation-dependent actions.

An 'external constant description' defines a constant to be known internally under the name given by the 'tag' and externally by the 'string denotation'. An application of this constant will result in implementation-dependent actions.

5.2. Standard externals.

Standard externals can be used in all programs without further notice. Their names can be redeclared by the user.
The workings of rules marked with a P are affected by pragmats (see chapter 6.), whereas rules marked with a . are not.

5.2.1. Integers.

For those data that are considered to be integers, the following standard externals are available.

. 'constant' zero, one, max int, min int, int size.
"zero" has the value 0, "one" has the value 1. "max int" has the value of the largest integer in the given implementation, and "min int" has the value of the smallest (most negative) integer in the given implementation. "int size" is the number of decimal digits necessary to represent "max int".

. 'function' add + >a + >b + head> + tail>.
The double-length sum of "a" and "b" is given in "head" and "tail":
a + b = head × (max int + 1) + tail, such that |head| is minimal.

. 'function' subtr + >a + >b + head> + tail>.
  The double-length difference of "a" and "b" is given in "head" and "tail": $a - b = head \times (max\ int + 1) + tail$, such that $|head|$ is minimal.

. 'function' mult + >a + >b + head> + tail>.
  The double-length product of "a" and "b" is given in "head" and "tail": $a \times b = head \times (max\ int + 1) + tail$, such that $|head|$ is minimal.

. 'function' divrem + >a + >b + quot> + rem>.
  The quotient and remainder of the integer division of "a" by "b" is given in "quot" and "rem": $a = b \times quot + rem$, such that "rem" is non-negative and minimal. "b" must not be zero.

P 'function' plus + >a + >b + c>.
  The sum of "a" and "b" is given in "c".

P 'function' minus + >a + >b + c>.
  The difference of "a" and "b" (i.e., a - b) is given in "c".

P 'function' times + >a + >b + c>.
  The product of "a" and "b" is given in "c".

P 'function' incr + >x>.
  The value of "x" is increased by 1.

P 'function' decr + >x>.
  The value of "x" is decreased by 1.

. 'question' less + >p + >q.
  Succeeds if "p" is less than "q", fails otherwise.

. 'question' lseq + >p + >q.
  Succeeds if "p" is less than or equal to "q", fails otherwise.

. 'question' more + >p + >q.
  Succeeds if "p" is more than "q", fails otherwise.

. 'question' mreq + >p + >q.
  Succeeds if "p" is more than or equal to "q", fails otherwise.

. 'question' equal + >p + >q.
  Succees if "p" is equal to "q", fails otherwise. It is identical to "p = q".

. 'question' noteq + >p + >q.
Succeeds if "p" is not equal to "q", fails otherwise.

. 'action' random + >p + >q + r>.
A pseudo-random number between "p" and "q" is given in "r": $p \leq r \leq q$.
The value of "r" is derived from an element in a circular sequence of random numbers. The next call of "random" will derive its output value from the next number in that sequence, etc..

. 'action' set random + >n.
"n" determines in some way the position in the circular sequence of random numbers mentioned above, from which the next call of "random" will obtain its output value.

. 'action' set real random.
The position in the circular sequence of random numbers used by "random" is determined in an unpredictable way.

. 'question' sqrt + >a + root> + rem>.
If "a" is non-negative, "sqrt" succeeds; the square root and remainder of "a" are yielded such that $a = root \times root + rem$, and "rem" is non-negative and minimal. Otherwise it fails.

. 'function' pack int + from[] + >n + int>.
The right-most "n" elements in the list "from" must be integer values corresponding to characters that indicate digits. The digits thus indicated are considered as the decimal notation of an integer, and the value of this integer is yielded in "int".
A check on integer overflow is performed.
Example: if the 4 right-most elements of "st" are:

/0/,/2/,/7/,/3/

then a call of "pack int + st + 4 + res" will assign the value 273 to "res".

. 'action' unpack int + >int + []st[].
The absolute value of "int" is written in decimal notation in "int size" digits, and "st" is extended with the integer values of the digits thus obtained, in left-to-right order.

The following externals are recommended.

. 'function' date + year> + month> + day>.
The year, month and day are yielded in "year", "month" and "day".

. 'function' time + amount>.
If two calls of "time" yield "amount1" and "amount2" respectively, then amount2 - amount1 is in some way indicative for the time spent by the program between these two calls.

5.2.2. Words.

For those data that are considered to be arrays of bits (words), the following standard externals are available.

. 'constant' word size.
The bits in a word are numbered (from left to right) from "word size" - 1 to 0.

. 'constant' false, true.
The value of "false" is 0, that of "true" is 1.

. 'function' bool invert + >a + b>.
A word is yielded in "b" that contains a 1 in those positions where "a" contains a 0, and a 0 otherwise.

. 'function' bool and + >a + >b + c>.
A word is yielded in "c" that contains a 1 in those positions where both "a" and "b" contain a 1, and a 0 otherwise.

. 'function' bool or + >a + >b + c>.
A word is yielded in "c" that contains a 1 in those positions where either "a" or "b" or both contain a 1, and a 0 otherwise.

. 'function' bool xor + >a + >b + c>.
A word is yielded in "c" that contains a 1 in those positions where "a" and "b" differ, and a 0 otherwise.

P 'function' left circ + >x> + >n.
The bit-array in "x" is shifted "n" positions to the left; bits leaving the word on the left are re-introduced on the right. It is required that $0 \leq n \leq$ word size.

P 'function' left clear + >x> + >n.
The bit-array in "x" is shifted "n" positions to the left; bits leaving the word on the left are discarded and 0-s are introduced on the right. It is required that $0 \leq n \leq$ word size.

P 'function' right circ + >x> + >n.
The bit-array in "x" is shifted "n" positions to the right; bits leaving the word on the right are re-introduced on the left. It is required that $0 \leq n \leq$ word size.

P 'function' right clear + >x> + >n.
The bit-array in "x" is shifted "n" positions to the right; bits leaving the word on the right are discarded and 0-s are introduced on the left. It is required that $0 \leq n \leq$ word size.

P 'question' is elem + >x + >n.
Succeeds if the "n"-th bit in "x" is a 1, fails otherwise. It is required that $0 \leq n <$ word size.

. 'question' is true + >x.
Succeeds if "x" contains at least one 1, fails otherwise.

. 'question' is false + >x.
Succeeds if "x" contains only 0-s, fails otherwise.

P 'function' set elem + >x> + >n.
The "n"-th bit in "x" is made equal to 1. It is required that $0 \leq n <$ word size.

P 'function' clear elem + >x> + >n.
The "n"-th bit in "x" is made equal to 0. It is required that $0 \leq n <$ word size.

P 'function' extract bits + >x + >n + y>.
A word is yielded in "y" that contains copies of the right-most "n" bits in "x" in the corresponding positions, and 0-s in the remaining positions, if any. It is required that $0 \leq n \leq$ word size.

. 'question' first true + >x + n>.
If "x" contains at least one 1, "first true" succeeds and yields the position of the left-most 1 in "n". Otherwise it fails.

. 'function' pack bool + from[] + >n + word>.
The right-most "n" bits of "word" are filled as follows. If the element in "from" with address ">>from - i" contains at least one 1, bit "i" of "word" is set to 1, and otherwise to 0, for $0 \leq i < n$. The remaining bits in "word", if any, are 0. It is required that $0 \leq n <$ word size, regardless of pragmats.

. 'action' unpack bool + >word + []st[].
The stack "st" is extended with "word size" blocks of one location each, the location with address ">>st - i" containing a copy of the "i"-th bit in "word", for $0 \leq i <$ wordsize.

5.2.3. Strings.

For those data in stacks and tables that are considered to be strings, the following externals are available.

. 'constant' max char.
"max char" has the maximum integer value that corresponds to a character.

. 'function' to ascii + >c + d>.
"d" is given the integer value that corresponds in ASCII-code to the character that corresponds to "c" in the code used. It is required that $0 \leq c \leq$ max char.

. 'function' from ascii + >c + d>.
"d" is given the integer value that corresponds in the code used to the character that corresponds to "c" in ASCII. It is required that $0 \leq c \leq 127$.

. 'action' pack string + from[] + >n + []to[].
The right-most "n" elements of "from" must be values that correspond to characters. These characters are packed, in some way, into some number "m" of values, and the stack "to" is extended with "m" blocks of one location each, containing these values. The packed format thus obtained is the same as that used for storing 'string's in lists (see 4.1.5). The "pointer" to the string is the address of the right-most element. So, after a call of "pack string", the 'limit' >>to is the pointer to the resulting packed string.

P 'action' unpack string + from[] + >p + []to[].
The pointer "p" must point into the list "from" and be the address of a packed string. This string is unpacked yielding a sequence of "m" character values, and the stack "to" is extended with "m" blocks of one location each, containing these values in left-to-right order.

P 'question' string elem + text[] + >p + >n + c>.
The pointer "p" must point into "text" and be the address of a packed string. If this string has an "n"-th character (counting from 0), its value is yielded in "c" and "string elem" succeeds; otherwise it fails.

P 'function' string length + text[] + >p + n>.
The pointer "p" must point into "text" and be the address of a packed string. The number of characters in this string is yielded in "n".

P 'function' compare string + t1[] + >p1 + t2[] + >p2 + trit>.
The pointer "p1" must point into "t1" and be the address of a packed string, s1. The pointer "p2" must point into "t2" and be the address of a packed string, s2. These two strings are compared in some way: if s1 is smaller than (lexicografically before) s2, "trit" is set to -1; if they are equal, "trit" is set to 0; otherwise "trit" is set to 1.

P 'action' unstack string + []st[].
The "max limit" of "st" must point into "st" and be the address of a packed string. The blocks containing this string are removed from "st".

. 'question' may be string pointer + text[] + >p.
Succeeds if "p" points into "text" and can be interpreted as the address of a packed string. Otherwise it fails.

5.2.4. Lists.

For lists the following externals are available.

. 'constant' nil.
"nil" is a value that does not point into any list.

. 'question' was + a[] + >p.
Succeeds if "p" points into "a", fails otherwise.

. 'function' next + a[] + >p>.
The calibre of "a" is added to "p".

. 'function' previous + a[] + >p>.
The calibre of "a" is subtracted from "p".

. 'function' list length + a[] + l>.
The number of elements in "a" is yielded in "l".

. 'action' unstack + []st[].
The stack "st" must contain at least one block. The right-most block of "st" is removed. Its locations can be reclaimed by an 'extension', its contents are lost.

. 'action' unstack to + []st[] + >pnt.
Zero or more blocks are removed from the right hand side of "st", so that the "max limit" of "st" becomes equal to "pnt". If this cannot be done, an error message follows.

. 'action' unqueue + []st[].
The stack "st" must contain at least one block. The left-most block of "st" is removed. Its (virtual) locations and its contents are lost.

. 'action' unqueue to + []st[] + >pnt.
Zero or more blocks are removed from the left hand side of "st", so that the "min limit" of "st" becomes equal to "pnt". If this cannot be done, an error message follows.

. 'action' scratch + []st[].
All blocks in "st" are removed. Their locations can be reclaimed through 'extension's, their contents are lost.

. 'action' delete + []st[].
All blocks in "st" are removed, as in a call of "scratch". Moreover, the run-time system will disregard "st" until a possible subsequent 'extension' on "st". Consequently, the remaining stacks will get better service, but reactivating "st" will be expensive.

5.2.5. Files.

The following standard externals on files are available.

. 'constant' new line, same line, new page.
These constants are predefined values to be used as control integers for "charfiles". Their intended meanings are "print on new line", "print again on same line" and "print on first line of next page" respectively, as far as meaningful for the 'charfile' and as far as implementable in the system.

. 'constant' rest line.
"rest line" acts as a dummy control integer and is used by "get line", "put line" and "put char".

. 'predicate' get line + ""file + []st[] + cint>.
The file "file" must be a charfile. If the file is exhausted, "get line" fails. Otherwise the next item in "file" is read; if it is a control integer, it is assigned to "cint", otherwise "cint" is set to "rest line". Then zero or more characters are read from "file" until the end of the line. The stack "st" is extended with these characters in left-to-right order.

. 'action' put line + ""file + a[] + >cint.
The file "file" must be a charfile; "a" must only contain values that correspond to characters. If "cint" is not "rest line", a line with control integer "cint" is written on file "file", containing the characters in "a" in left-to-right order. Otherwise the characters in "a" are appended to the last line written on "file".

. 'predicate' get char + ""file + char>.
The file "file" must be a charfile. If the file is not exhausted, the next character or control integer is read and delivered in "char". Otherwise "get char" fails.

. 'action' put char + ""file + >char.
The file "file" must be a charfile. The value of "char" must either correspond to a character or be a control integer. This character or control integer is written on file "file", except the control integer "rest line", which is ignored.

P 'action' put string + ""file + text[] + >p.
The file "file" must be a charfile; the pointer "p" must point into "text" and be the address of a packed string. This string is written on the file "file".

. 'predicate' get int + ""file + int>.
The file "file" must be a charfile. A call of "get int" will read and skip any number of spaces and control integers on "file" until it either reaches the end of the file, in which case it fails, or finds a digit, plus-sign or minus-sign. It will then read and collect one or more digits until a non-digit is found: this non-digit is not read. The value of this stream of digits considered as a signed decimal number is given in "int".
A subsequent call of "get char" will yield the non-digit mentioned.
If the above cannot be performed, an error message is given.
This rule involves backtrack. It is not intended for use in programs that handle input very carefully; it is meant to provide an easy means for reading numbers.

. 'action' put int + ""file + >int.
"intsize" + 1 characters are appended to the last line on "file", which must be a charfile. These characters are: zero or more spaces, the sign of "int" and the characters of the decimal representation of the absolute value of "int" without leading zeroes.

. 'constant' numerical, pointer.
These constants are predefined values that can be used as type indications in datafiles. For their meanings see 4.2.2..

. 'predicate' get data + ""file + data> + type>.
The file "file" must be a datafile. If the file is not exhausted, the next data-item is read, its value delivered in "data" and its type in "type". Otherwise it fails. For a more detailed description see 4.2.2..

. 'action' put data + ""file + >data + >type.
The file "file" must be a datafile. A data-item is written on the file, consisting of the value "data" and the type "type". For a more detailed description see 4.2.2..

The following four externals are recommended.

. 'predicate' back char + ""file.
The file "file" must be a charfile. If the last item read was a control integer, or if there is no last item read, "back char" fails. Otherwise it succeeds and the file is backspaced over one character, i.e., a subsequent call of "get char" will yield the last item read.

. 'predicate' back data + ""file.
The file "file" must be a datafile. If there is not yet a last item read, "back data" fails. Otherwise it succeeds and the file is backspaced over one data-item, i.e., a subsequent call of "get data" will yield the last item read.

. 'predicate' back line + ""file.
The file "file" must be a charfile. If there is not yet a last item read, "back line" fails. Otherwise it succeeds and the file is backspaced until a subsequent call of "get char" would read the last control integer read.

. 'predicate' back file + ""file.
If there is not yet a last item read, "back file" fails. Otherwise it succeeds and the file is backspaced or "back-data-ed" until the beginning of the file.

6. Pragmats.

Pragmats are used for three purposes:

a. to control certain aspects of the compilation,

b. to control the functioning of certain standard externals, and

c. to supply implementation-dependent information to the machine-dependent part of the compiler.

The exact position of the pragmat in the program may be significant.

Syntax:

```
pragmat: pragmat symbol, pragmat item list, point symbol.
pragmat item list: pragmat item, (comma symbol, pragmat item list).
pragmat item:
        tag, equals symbol, integral denotation;
        tag, equals symbol, string denotation;
        tag, equals symbol, pragmat item;
        pragmat item list pack.
pragmat item list pack:
        open symbol, pragmat item list, close symbol.
```

Example:

```
'pragmat' title = "aleph compiler",
          bounds = (taglist = off, numb adm = on),
          macro = (convert 1 to 2 compl, set all bits).
```

Before the meaning of a 'pragmat' is determined, it is preprocessed: all 'pragmat item list pack's are removed in the following way.
For every 'pragmat item list pack' that is preceded by an 'equals symbol' preceded by a 'tag', the 'equals symbol' and 'tag' are removed and inserted in front of each 'pragmat item' in the 'pragmat item list pack'.
Subsequently all 'open symbol's and 'close symbol's are removed.

Thus the 'pragmat item':

```
algol = (apl = (lisp, 2, pl = 1), snobol)
```

has the same meaning as:

```
algol = apl = lisp, algol = apl = 2,
algol = apl = pl = 1, algol = snobol.
```

All 'pragmat item's are now of the form 'tag, equals symbol, pragmat item or int or string'. They are divided into three groups according to the 'tag': compiler-pragmats, affecting the compiler; external-pragmats, affecting the standard externals; and user-pragmats.

6.1. Compiler-pragmats.

The 'tag's "background", "bounds", "class", "compile", "count", "dump", "first col", "last col", "macro" and "title" identify compiler-pragmats.

. background = 'tag'
The 'tag' must identify a list. The identified list will be kept on background memory if possible and necessary.
The position of this 'pragmat' is immaterial.

. bounds = 'tag1' = 'tag2'
The 'tag1' must identify a list. The 'tag2' can be:
"on":
subsequent 'element's of 'tag1' will be compiled with a dynamic bound check (if necessary).
"off":
no such check is made.
The standard option is "on".
An 'element' of a formal table or stack will be compiled with a bound check, unless at that point in the program no bound checks would be compiled for any 'element'.

. class = 'tag'
The 'tag' can be:
"on":
subsequent 'classification's are compiled with a dynamic check that determines if the value of its 'classifier' belongs in any area (if necessary).
"off":
no such check is compiled.
The standard option is "on".

. compile = 'tag'
The 'tag' can be:
"off":
subsequent program text will be interpreted in the following sense:
a. the 'rule body' of a 'rule declaration', the 'rule tag' of which is used in normally compiled text will be interpreted as dummy,
b. a 'rule declaration' the 'rule tag' of which is not used in normally compiled text will be ignored,
c. a 'data declaration' will be ignored,
d. a 'pragmat item' other than "compile = on" will be ignored;
injudicious application of this pragmat can render a correct program incorrect.

"on":
normal compilation is resumed.
"all":
subsequent 'pragmat item's of the form "compile = off" will have no effect.
The standard option is "on".

. count = 'tag'
The 'tag' can be:
"rule":
a counter is kept for each subsequent "rule" and "compound member". The initial value of the counter is 0; it is incremented by 1 for every entrance to its "rule" or "compound member". The counters are printed at program termination.
"member":
same as for "rule", except that a counter is kept for every member.
"off":
no counters are kept for subsequent program text.
The standard option is "off".

. dump = 'tag'
The 'tag' can be:
"global":
upon error termination a symbolic dump of all global variables and stacks will be printed.
"rule":
upon error termination a symbolic dump of the run-time stack will be printed.
"member":
upon error termination the number of the current member (as determined by the compiler) will be printed.
The position of this pragmat in the program is immaterial.

. first col = 'integral denotation'
Call the value of the 'integral denotation' i. The first i-1 characters on subsequent program lines are ignored. This alignment can be revoked in another "first col" pragmat. An initial pragmat "first col = 1" is assumed.

. last col = 'integral denotation'
Call the value of the 'integral denotation' i. All characters beyond the i-th position on subsequent program lines are ignored. This alignment can be revoked in another "last col" pragmat. An initial pragmat "last col = 72" is assumed.

. macro = 'tag'
The 'tag' must identify a non-recursive rule. Calls of this rule will be implemented through textual substitution rather than by subroutine call.

. title = 'string denotation'
The 'string denotation' is the title of the program.
The standard title is empty.

6.2. External-pragmats.

The 'tag's "overflow", "wrong bit" and "wrong string" identify external-pragmats. They can be used to control those tests in the standard externals that, in the average implementation, will need considerable time.

. overflow = 'tag'
The 'tag' can be:
"on":
a run-time test is made to check that a resulting integral value lies between "min int" and "max int".
"off":
no such test is made.
The standard option is "on".

This pragmat affects the standard externals "plus", "minus", "times", "incr" and "decr".

. wrong bit = 'tag'
The 'tag' can be:
"on":
a run-time test is made to check that no non-existing bits in a word are addressed.
"off":
no such test is made.
The standard option is "on".

This pragmat affects the standard externals "left circ", "left clear", "right circ", "right clear", "is elem", "set elem", "clear elem" and "extract bits".

. wrong string = 'tag'
The 'tag' can be:
"on":
a run-time test is made to check that a pointer, purported to point to a string in a given list, indeed does so.
"off":
no such test is made.
The standard option is "on".

This pragmat affects the standard externals "unpack string", "string elem", "string length", "compare string", "unstack string" and "put string".

6.3. User-pragmats.

Pragmats not identified in 6.1 or 6.2 are considered user-pragmats and are transferred to the machine-dependent part of the compiler.

7. The representation of programs.

7.1. The program.

The program produced by the notion 'program' consists of a series of symbols. Into this program comments may be inserted in the following way.
The program is considered as a sequence of the following units:
'tag's,
'integral denotation's,
'character denotation's,
'string denotation's and
'symbol's not occurring in one of the above.
Spaces may be added in front of all these units and inside 'tag's and 'integral denotation's.
Long comments may be added in front of all these units. A long comment consists of a dollar-sign ($), followed by zero or more characters which are not dollar-signs, followed by a dollar-sign.
Short comments may be added in front of all units except 'tag's and 'integral denotation's. A short comment consists of a sharp-sign (#) followed by zero or more letters, digits and spaces.

In the program thus obtained all symbols are expanded into characters (e.g., 'variable symbol' turns into "'var'").
The program text is then divided into lines in such a way that no comment is spread over two or more lines. If a line ends with a dollar-sign from a long comment, this dollar-sign may be omitted. In other words: long comments start with a dollar-sign and end at a dollar-sign or at the end of the line; short comments start with a sharp-sign and end at the first character that is not a letter, a digit or a space, or at the end of the line.

Depending on the pragmats "first col" and "last col" (see 6.1.) a number of characters must be added before each line or may be added behind each line.

7.2. The characters.

symbol	representation
absolute symbol	/
action symbol	'action' or 'act'
actual affix symbol	+
box symbol	=
bus symbol	] or)
by symbol	/
charfile symbol	'charfile'
close symbol	)
colon symbol	:
comma symbol	,
constant symbol	'constant' or 'cst'

datafile symbol	'datafile'
end symbol	'end'
equals symbol	=
exit symbol	'exit'
external symbol	'external'
failure symbol	-
formal affix symbol	+
function symbol	'function' or 'fct'
left symbol	<
local affix symbol	-
minus symbol	-
of symbol	×
open symbol	(
plus symbol	+
point symbol	.
pragmat symbol	'pragmat'
predicate symbol	'predicate' or 'pred'
question symbol	'question' or 'qu'
quote symbol	"
repeat symbol	:
right symbol	>
root symbol	'root'
semicolon symbol	;
stack symbol	'stack'
sub symbol	[or (
success symbol	+
table symbol	'table'
times symbol	×
up to symbol	:
variable symbol	'variable' or 'var'

8. Examples.

Example 1:

```
$  towers of hanoi  $
'charfile' print = "output">.

'action' move tower + >length + >from + >via + >to:
    length = 0;
    decr + length, move tower + length + from + to + via,
    move disc + from + to, move tower + length + via + from + to.

'action' move disc + >s1 + >s2:
    put char + print + s1, put char + print + s2,
    put char + print + / /.

'action' root: move tower + 6 + /a/ + /b/ + /c/.

'root' root.

'end'
```

Example 2:

```
$ towers of hanoi, full printing of the towers $
'charfile' print = "output">.

'stack' [1] a, [1] b, [1] c.

'constant' size = 5.

'action' move tower + >length + []from[] + []via[] + []to[]:
    length = 0;
    decr + length, move tower + length + from + to + via,
    move disc + from + to, print towers,
    move tower + length + via + from + to.

'action' move disc + []st1[] + []st2[]:
    × st1[>>st1] -> st2 × st2, unstack + st1.

'action' print towers - ln:
    size -> ln,
   (lines:
        ln = 0;
        print disc + a + ln, print disc + b + ln,
        print disc + c + ln, put char + print + new line,
        decr + ln, :lines).

'action' print disc + []st[] + >line - index:
    minus + line + 1 + index, plus + index + <<st + index,
   (was + st + index, print actual disc + st[index];
    print blank disc).

'action' print actual disc + >nmb - spc:
    minus + size + nmb + spc,
    repeat + spc + / /, repeat + nmb + /x/, repeat + 1 + /x/,
    repeat + nmb + /x/, repeat + spc + / /.

'action' print blank disc:
    repeat + size + / /, repeat + 1 + / /, repeat + size + / /.

'action' repeat + >cnt + >sb:
   cnt = 0; put char + print + sb, decr + cnt, :repeat.

'action' play towers - n:
    size -> n,
   (fill a: n = 0; decr + n, x n->a x a, :fill a),
    print towers, move tower + size + a + b + c.

'root' play towers.

'end'
```

Example 3:

```
$ symbolic differentiation, problem iii in "machine oriented
$ languages bulletin", molb 3.1.2., (1973). $
'charfile' out = "output">.

'stack' [100] (op, left, right) expr.

'table' operator = ("+":plus op, "-":min op,"x": tim op, "/":div op,
    "ln": ln op $ ln(f) is represented as 0 "ln" f $,
    "pow": pow op $ pow(f, g) is represented as f "pow" g $).

'stack' [10] const = (0: c zero, 1: c one, 2: c two).

'stack' [1] var = ("x": x var).

'action' derivative + >e + de> - f - df - g - dg - n1 - n2 - n3:
    was + const + e, c zero -> de;
    was + var + e, c one -> de;
    leftxexpr[e] -> f, rightxexpr[e] -> g,
    derivative + f + df, derivative + g + dg,
       ( = opxexpr[e] =
          [plus op], gen node + plus op + df + dg + de;
          [min  op], gen node + min  op + df + dg + de;
          [tim  op],
            gen node + tim op + f + dg + n1,
            gen node + tim op + df + g + n2,
            gen node + plus op + n1 + n2 + de;
          [div  op],
                  gen node + tim op + df + g + n1,
                  gen node + tim op + f + dg + n2,
                  gen node + min op + n1 + n2 + n1,
                  gen node + pow op + g + c two + n2,
                  gen node + div op + n1 + n2 + de;
                [ln   op], gen node + div op + dg + g + de;
                [pow  op],
                  gen node + min op + g + c one + n1,
                  gen node + pow op + f + n1 + n1,
                  gen node + tim op + df + g + n2,
                  gen node + tim op + n2 + n1 + n1,
                  gen node + ln op + c zero + f + n2,
                  gen node + tim op + n2 + dg + n2,
                  gen node + pow op + f + g + n3,
                  gen node + tim op + n2 + n3 + n2,
                  gen node + plus op + n1 + n2 + de;
                +).

'action' print expr + >e - zz:
    was + const + e, put int + out + const[e];
    was + var + e, put string + out + var + e;
    opxexpr[e] -> zz,
       ( = zz =
          [plus op; min op; tim op; div op],
            put char + out + /(/, print expr + left xexpr[e],
            put char + out + /)/, put string + out + operator + zz,
            put char + out + /(/, print expr + rightxexpr[e],
            put char + out + /)/;
          put string + out + operator + zz, put char + out + /(/,
           (equal + zz + pow op, print expr + leftxexpr[e],
            put char + out + /,/; +), print expr + rightxexpr[e],
            put char + out + /)/
       ).

'action' test - e1 - e2 - e3:
    gen node + pow op + x var + x var + e1, $ pow(x, x) $
        print expr + e1,
        put char + out + new line,
    derivative + e1 + e2, print expr + e2,
        put char + out + new line,
    derivative + e2 + e3, print expr + e3,
        put char + out + new line,
    gen node + div op + x var + x var + e1, $ x/x $
        print expr + e1,
        put char + out + new line,
    derivative + e1 + e2, print expr + e2,
        put char + out + new line,
    derivative + e2 + e3, print expr + e3,
        put char + out + new line.

'action' gen node + >op + >left + >right + res>:
    x op -> op, left -> left, right -> rightx expr,
    >>expr -> res.
```

```
'root' test.

'end'
```

Example 4:

```
'action' quicksort + >from + >to + a[]
    - left - middle - right - amiddle:
$
$ this rule sorts the elements in the stack "a" from "from" to
$ "to" in ascending order. the algorithm used is a variation of
$ "quicksort", computer j. 5 (1), 10-15 (1962)
$
    mreq + from + to;
    $ the area to be sorted is not empty:
    $ it is split into three parts, left, middle and right.
    $ the middle contains one or more equal elements
    from -> left, random + from + to + middle, to -> right,
        a[middle] -> a middle,
   (split:
       (push right:
            more + left + to;
            more + a[left] + a middle;
            incr + left, : push right),
       (push left:
            more + from + right;
            more + a middle + a[right];
            decr + right, : push left),
       (less + left + right,
           ( - elem:
            a[left] -> elem, a[right] -> a[left], elem -> a[right]),
            incr + left, decr + right, : split;
        less + middle + right,
            a[right] -> a[middle], a middle -> a[right],
            decr + right;
        more + middle + left,
            a[left] -> a[middle], a middle -> a[left], incr + left;
        +)
       ),
    quicksort + from + right + a, quicksort + left + to + a.
```

Example 5:

```
$  "next perm" considers the right-most "n" elements of "st"
$  as a permutation and replaces them by the elements of the next
$  permutation in lexicographical order. If there is no next
$  permutation, "next perm" fails.

'question' next perm + >n + []st[] - br - item - p - q:
  $ find break point or fail:
```

```
 min int -> item, >>st -> br, minus + >>st + n + p,
(breakpoint:
   lseq +  br + p, -;
   less + st[br] + item; st[br] -> item, decr + br, :breakpoint),
 $ invert part after break point:
 plus + br + 1 + p, >>st -> q,
(invert:
   lseq + q + p;
   st[p] -> item, st[q] -> st[p], item -> st[q],
     incr + p, decr + q, :invert),
 $ find the value of the first element which is
 $   larger than that at the breakpoint:
 st[br] -> item, br -> p,
(first: incr + p, (more + st[p] + item; :first)),
 $ and swap them:
 st[p] -> st[br], item -> st[p].
```

9. The grammar.

```
[ syntax of aleph, 28-05-1974 ]

[ 7.2. symbols ]

open symbol              [ ( ];          [ 3.7., 4.1.1., 4.1.5., 6]
close symbol             [ ) ];          [ 3.7., 4.1.1., 4.1.5., 6 ]
absolute symbol          [ / ];          [ 4.1.1. ]
plus symbol              [ + ];          [ 4.1.1. ]
minus symbol             [ - ];          [ 4.1.1. ]
times symbol             [ × ];          [ 4.1.1. ]
by symbol                [ / ];          [ 4.1.1. ]
left symbol              [ < ];          [ 4.1.7. ]
right symbol             [ > ];          [ 3.1.1., 4.1.7., 4.2. ]
equals symbol            [ = ];          [ 3.4., 4.1.5., e.a. ]
point symbol             [ . ];
colon symbol             [ : ];          [ 3.2.2., 4.1.5. ]
semicolon symbol         [ ; ];          [ 3.2.2., 3.8. ]
comma symbol             [ , ];          [ 3.2.2., 3.8., e.a. ]
sub symbol               [ [ ];          [ 3.2.1., 3.5., 3.8., 4.1.6. ]
bus symbol               [ ]  ;          [ 3.2.1., 3.5., 3.8., 4.1.6. ]
quote symbol             [ " ];          [ 4.1.5. ]
formal affix symbol      [ + ];          [ 3.2.1. ]
local affix symbol       [ - ];          [ 3.2.2. ]
repeat symbol            [ : ];          [ 3.6. ]
success symbol           [ + ];          [ 3.6. ]
failure symbol           [ - ];          [ 3.6. ]
box symbol               [ = ];          [ 3.8., 4.1.6. ]
up to symbol             [ : ];          [ 3.8. ]
actual affix symbol      [ + ];          [ 3.5. ]
of symbol                [ × ];          [ 3.2.3., 3.5. ]

[ bold face symbols ]
constant symbol          [ 'constant',  'cst' ];  [ 4.1.2., 5.1. ]
variable symbol          [ 'variable',  'var' ];  [ 4.1.3. ]
stack symbol             [ 'stack' ];             [ 4.1.6. ]
table symbol             [ 'table' ];             [ 4.1.5., 5.1. ]
charfile symbol          [ 'charfile' ];          [ 4.2. ]
datafile symbol          [ 'datafile' ];          [ 4.2. ]
predicate symbol         [ 'predicate', 'pred' ]; [ 3.2.1. ]
question symbol          [ 'question',  'qu' ];   [ 3.2.1. ]
action symbol            [ 'action',    'act' ];  [ 3.2.1. ]
function symbol          [ 'function',  'fct' ];  [ 3.2.1. ]
external symbol          [ 'external' ];          [ 5.1. ]
pragmat symbol           [ 'pragmat' ];           [ 6. ]
exit symbol              [ 'exit' ];              [ 3.6. ]
root symbol              [ 'root' ];              [ 3.1. ]
end symbol               [ 'end' ];               [ 3.1. ]

[ other primitives ]
tag;                                              [ 4.1.5., 6., e.a. ]
```

```
digit;                                              [ 4.1.1. ]
character;                                          [ 4.1.1. ]
non quote item.                                     [ 4.1.5. ]


[ 3.1. the program ]
program:
    (information sequence), root, (information sequence), end symbol.
information sequence:
    information, (information sequence).
information:
    declaration; pragmat.
root:
    root symbol, affix form, point symbol.
declaration:
    rule     declaration;
    data     declaration;
    external declaration.

[ 3.2.1. rule declarations ]
rule declaration:
    typer, rule tag, (formal affix sequence), actual rule, point symbol.
typer:
    action symbol; function symbol; predicate symbol; question symbol.
rule tag:
    tag.
formal affix sequence:
    formal affix, (formal affix sequence).
formal affix:
    formal affix symbol, formal.
formal:
    formal variable; formal stack; formal table; formal file.
formal variable:
    (right symbol), variable tag, (right symbol).
formal table:
    (field list pack), table tag, sub bus.
formal stack:
    sub bus, (field list pack), stack tag, sub bus.
sub bus:
    sub symbol, bus symbol.
formal file:
    quote image, file tag.

[ 3.2.2. actual rules ]
actual rule:
    (local affix sequence), colon symbol, rule body.
local affix sequence:
    local affix, (local affix sequence).
local affix:
    local affix symbol, local variable.
local variable:
    variable tag.
rule body:
    alternative series; classification.
alternative series:
    alternative, (semicolon symbol, alternative series).
alternative:
    last member; member, comma symbol, alternative.

[ 3.2.3. members ]
last member:
    member; terminator.
member:
    affix form; operation; compound member.

[ 3.4. operations ]
operation:
    transport; identity; extension.
transport:
    source, destination sequence.
destination sequence:
    destination, (destination sequence).
destination:
    to  token, variable.
to token:
    minus symbol, right symbol.
identity:
    source, equals symbol, source.
extension:
    of symbol, field transport list, of symbol, stack tag.
field transport list:
    field transport, (comma symbol, field transport list).
field transport:
    source, selector destination sequence.
selector destination sequence:
    selector destination, (selector destination sequence).
selector destination:
    to token, selector.

[ 3.5. affix forms ]
affix form:
    rule tag, (actual affix sequence).
actual affix sequence:
    actual affix, (actual affix sequence).
actual affix:
    actual affix symbol, actual.
actual:
    source; list tag; file tag.
source:
    constant; variable.
constant:
    plain value; table element.
variable:
    variable tag; stack element.
table element:
    (selector, of symbol), table tag, sub symbol, source, bus symbol.
```

```
stack element:
    (selector, of symbol), stack tag, sub symbol, source, bus symbol.

[ 3.6. terminators ]
terminator:
    jump; exit; success symbol; failure symbol.
jump:
    repeat symbol, rule tag.
exit:
    exit symbol, expression.

[ 3.7. compound members ]
compound member:
    open symbol, (local part, colon symbol), rule body, close symbol.
local part:
    rule tag, (local affix sequence); local affix sequence.

[ 3.8. classifications ]
classification:
    classifier box, class chain.
classifier box:
    box symbol, classifier, box symbol.
classifier:
    source.
class chain:
    class, semicolon symbol, class chain; last class.
class:
    area, comma symbol, alternative.
area:
    sub symbol, zone series, bus symbol.
zone series:
    zone, (semicolon symbol, zone series).
zone:
    (expression), up to symbol, (expression); expression; list tag.
last class:
    class; alternative.

[ 4. data declarations ]
data declaration:
    constant declaration;
    variable declaration;
    stack    declaration;
    table    declaration;
    file     declaration.

[ 4.1.1. expressions ]
expression:
    (plus minus), term; expression, plus minus, term.
term:
    (term, times by), base.
base:
    plain value; expression pack.
plain value:
    integral denotation; character denotation; constant tag; limit.
integral denotation:
    (integral denotation), digit.
character denotation:
    absolute symbol, character, absolute symbol.
expression pack:
    open symbol, expression, close symbol.
plus minus:
    plus symbol; minus symbol.
times by:
    times symbol; by symbol.

[ 4.1.2. constant declarations ]
constant declaration:
    constant symbol, constant description list, point symbol.
constant description list:
    constant description, (comma symbol, constant description list).
constant description:
    constant tag, equals symbol, expression.
constant tag:
    tag.

[ 4.1.3. variable declarations ]
variable declaration:
    variable symbol, variable  description list, point symbol.
variable description list:
    variable description, (comma symbol, variable description list).
variable description:
    variable tag, equals symbol, expression.
variable tag:
    tag.

[ 4.1.5. table declarations ]
table declaration:
    table symbol, table description list, point symbol.
table description list:
    table description, (comma symbol, table description list).
table description:
    table head, equals symbol, filling list pack.
table head:
    (field list pack), table tag.
table tag:
    tag.
field list pack:
    open symbol, field list, close symbol.
field list:
    field, (comma symbol, field list).
field:
    selector chain.
selector chain:
    selector, (equals symbol, selector chain).
selector:
    tag.
```

```
filling list pack:
    open symbol, filling list, close symbol.
filling list:
    filling, (comma symbol, filling list).
filling:
    single block; compound block; string filling.
single block:
    expression, (pointer initialization).
compound block:
    expression list proper pack, (pointer initialization).
pointer initialization:
    colon symbol, constant tag.
expression list proper pack:
    open symbol, expression list proper, close symbol.
expression list proper:
    expression, comma symbol, expression list.
expression list:
    expression, (comma symbol, expression list).
string filling:
    string denotation, (pointer initialization).
string denotation:
    quote symbol, (string item sequence), quote symbol.
string item sequence:
    string item, (string item sequence).
string item:
    non quote item; quote image.
quote image:
    quote symbol, quote symbol.

[ 4.1.6. stack declarations ]
stack declaration:
    stack symbol, stack description list, point symbol.
stack description list:
    stack description, (comma symbol, stack description list).
stack description:
    stack head, (equals symbol, filling list pack).
stack head:
    size estimate, (field list pack), stack tag.
size estimate:
    relative size; absolute size.
relative size:
    sub symbol, expression, bus symbol.
absolute size:
    sub symbol, box symbol, expression, box symbol, bus symbol.
stack tag:
    tag.

[ 4.1.7. limits ]
limit:
    min limit; max limit; calibre.
min limit:
    min token, list tag.
max limit:
    max token, list tag.
calibre:
    calibre token, list tag.
list tag:
    stack tag; table tag.
min token:
    left symbol, left symbol.
max token:
    right symbol, right symbol.
calibre token:
    left symbol, right symbol.

[ 4.2. file declarations ]
file declaration:
    file typer, file description list, point symbol.
file typer:
    charfile symbol; datafile symbol.
file description list:
    file description, (comma symbol, file description list).
file description:
    file tag, (area),
        equals symbol, (right symbol), string denotation, (right symbol).
file tag:
    tag.

[ 5.1. external declarations ]
external declaration:
    external rule declaration;
    external table declaration;
    external constant declaration.
external rule declaration:
    external symbol, typer, external rule description list, point symbol.
external rule description list:
    external rule description,
       (comma symbol, external rule description list).
external rule description:
    rule tag, (formal affix sequence), equals symbol, string denotation.
external table declaration:
    external symbol, table symbol,
        external table description list, point symbol.
external table description list:
    external table description,
       (comma symbol, external table description list).
external table description:
    (field list pack), table tag, equals symbol, string denotation.
external constant declaration:
    external symbol, constant symbol,
        external constant description list, point symbol.
external constant description list:
    external constant description,
       (comma symbol, external constant description list).
external constant description:
    constant tag, equals symbol, string denotation.
```

```
[ 6. pragmats ]
pragmat:
    pragmat symbol, pragmat item list, point symbol.
pragmat item list:
    pragmat item, (comma symbol, pragmat item list).
pragmat item:
    tag, equals symbol, integral denotation;
    tag, equals symbol, string  denotation;
    tag, equals symbol, pragmat item;
    pragmat item list pack.
pragmat item list pack:
    open symbol, pragmat item list, close symbol.
```

10. Index.