ALEPH Manual

D. Grune R. Bosch L.G.L.T Meertens

Fourth printing, 1982

0 Preface

ALEPH (acronym for "A Language Encouraging Program Hierarcy") is a high-level language designed to provide the programmer with a tool that will effectively aid him in structuring his program is a hierarchical fashion. The syntactic and semantic simplicity of ALEPH leads to efficient object code [6], so that 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.

Chapter two treats the ALEPH program is general terms. 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 and example programs are given in chapters seven and eight.

An ALEPH compiler exists, which translates ALEPH programs in ALICE program in a machine-independent fashion. ALICE[1] is a simple linear code designed to aid the installation of ALEPH on new systems. The ALEPH compiler is available on both ALEPH and ALICE. An ALICE transformer to COMPASS for the Cyber 170 is also available. The compile and count pragmats have not been implemented.

This is the fourth printing of the ALEPH Manual. Many paragraphs have been rephrased to remove inconsistencies; the paragraph numbers have been kept identical throughout all printings. Since the third printing in 1977 the following modification has been made.

3.4.3: the sources in an extension are evaluated *before* the stack is extended. This prevents the extension `* st[>>st]->st *st` from pushing uninitialized data on the stack `st`.

3.5: no math in calibre is required between a formal an actual list in the former is explicitly declared with zero selectors, rather than with one selector. This criterion is clearer and prevents misapplication of some standard externals.

5.2.5: the standard externals `back char`, `back data` and `back line` are deleted since their limited usefulness in no way justifies the effort needed in their implementation.

6.1: to be effective a macro pragmat must occur *before* the pertinent rule-declaration. This modification greatly increases the efficiency of the translation process.

6.1, 6.2: all pragmats to switch off run-time checking have been deleted.

1 Informal introduction to ALEPH

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 A grammar

The problem we shall treat 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 expression will contain only integers, plus-symbols, times-symbols and parentheses: and example might be `15 * (12 + 3 * 9)`.

First we put the requirements for the input to our program in the more transparent and clearer form of a context-free 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.
empty: .

The rule for input can be read as: input is an expression followed by and input-tail, whereas the rule for primary can bead as: a primary is either

- a left-parenthesis followed by an expression followed by a right-parenthesis, or
- an integer.

This grammar shows clearly 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 Rules

We shall now write a series of rules in ALEPH, one for each rule in the grammar. For the grammar rule 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 (local) variable `r` is (we are now at the colon) a term which will yield a result in `res`, followed by *either* (we are now at the left parenthesis) by a plus-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 there paraphernalia from the ALEPH rule we obtain something very similar to the original grammar rule:

```
'action'expression1:
    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.

We now direct our attention back to the ALEPH rule `expression` and consider what happens when it is "executed". 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 separated 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.

If `is symbol+/+/` "succeeds" 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 the 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" 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 action must be performed by a specific rule and what question 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 three very basic primitives in the language, the copying of a value, the test for equality of two values and the extension of a stack by a fixed number of given values. Often, however, these two primitives are not sufficient to express the action needed; the rule is then subdivided into other rules. There are, however, cases where this is not desirable (or not possible). In such cases the rule is declared `'external'` and its actions must be specified in a different way, often 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.

We shall now pay some attention to the exact notation (syntax) of the rule `expression`. All rules have the property that when they are called they are either guaranteed to succeed or they may fail. The word `'action'` indicates that a call of this rule is guaranteed to succeed. The name of the rule is `expression`, and `res` is its only formal "affix" (parameter). The `+` serves as a separator (it *affixes* the affix to the rule). The right arrowhead (`>`) indicates that the resulting value of `res` will be passed back to 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 thus and issue a message. The `+` sign and the term *affix* stem from the theory of affix grammars of which ALEPH is based [4, 5].

The `-r` specifies `r` as a local affix (local variable) of the 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 calls of rules. The parentheses group both alternatives into one action. The `+` between slashes (indicating "absolute value") represents the integer value of the plus-symbol in the code used. The semicolon separates alternatives, which are checked in textual order. As said before, the stand-alone `+` denotes the dummy action that always succeeds. The period ends the rule.

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
```

1.3 Further rules

In view of the above the rule for term should not surprise the reader:

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

Now we are tempted to render the 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 (which is a production of input) is incorrect; we now decide that we shall not do any error recovery, so we give and 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 like one action that will always succeed: either the right parenthesis is present in the input, or an error will be signaled. `no paren` is a constant that will be specified later on.

Writing the rule for `integer` is a trickier problem than is seems to be. For a comprehensive account on how to obtain correct and incorrect versions of it the reader is referred to [3]. 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 (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 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 Input

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. Comment starts with a `$`.

```
$ 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, as specified by the programmer. 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, advance the input 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 the `n` indicates that the value of `n` (which may have been changed!) will not be passed back to the calling rule.

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 (`-`) will be executed. The rule `digit` is equivalent to

```
'predicate'digit+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 classification is analogous to case statements in ALGOL 68 and other programming languages.

All the arithmetic used here on symbols is based on the (possibly machine-dependent) 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 value.

One more input rule most be supplied:

```
'action'get next symbol:
   get char+reader+buff,
      ((buff=/ /; buff=newline), get next symbol;
       + );
   stop->buff.
'constant'stop=-1.
```

`get char` is an (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 next symbol`[1] implies we have decided that spaces and new-lines are allowed

[1] In the manuscript `get char`.

in the input in all positions (a decision that was not yet present in the initial grammar).

1.5 Output

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 10, the quotient is placed back to `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 `put integer`. Subsequently, the last digit is printed through a call or `put char`. This is a simple but inefficient way of printing a number. A more convenient way of printing an integer is provided by the (standard) external rule `put int`.

For the printing of error messages 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 stacks and tables 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:

```
$ Error-message printing
'action'error+>er:
   put char+printer+new line,
     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 identified 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 Starting the program

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 rule 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` contains 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 until know 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 give the rule-declarations and data-declarations in any other order and the effect would still be 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 Some details

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

```
'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 tail 1+res.
```

and

```
'action'expression tail 1+>res>-r:
   is symbol+/+/, expression1+r, plus+res+r+res; +.
```

which, according to the above recipe, would yield:

```
'action'expression 2+res>:
   term+res,
    (expression tail 2-r:
      is symbol+/+/, expression 2+r, plus+res+r+res;
      +).
```

In a sense this is a more appropriate form than the one given in 1.2: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 in 1.2 exactly one must start from:

```
'action'expression 3+res>-r:
   term+res,expression tail 3+res+r.
```

and

```
'action'expression tail 3+>res>+r>:
   is+symbol+/+/,expression 3+r,plus+res+r+res;+.
```

2 Introduction to the manual

2.1 Interface with the outside world

The solution of a problem by means of a computer implies that a sequence of actions be specified that, when executed, lead 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 complied if the need arises),
d. the programmer-defined externals (definitions of action 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 description 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 (in 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 outside world along two paths:

a. as output files,
b. as a single integer (the termination state of the program) which is made available 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.

2.2 The syntactical description

The syntax of ALEPH is given in the form of a context-free grammar. The notation in this grammar follows a well-known scheme: the part on the right hand side of a syntax rule 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 enclosed in square brackets: this indicates that the given notions may or may not be present, i.e., they are optional.

The terminal symbols of the grammar, together with the representations, are listed in 7.2; all except four end in -symbol. A notion that ends in -tag produces tag. Such a notion then contains a hint as to exactly which tags are allowed by the context conditions. A full VW-grammar incorporating all context conditions was prepared by R. Glandorf, D; Grune and J. Verhangen [2].

Constituents of the grammar are printed in bold; programs and program fragments are printed in `script`.

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 declarations and pragmats, followed by an end symbol; in this sequence exactly one root must occur." The order in which the declarations and the root appear is immaterial. The position of some pragmats is significant (6.1).
Example of a program:

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

in which 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-declarations, in such an order that no data item is used before its value has been calculated. If no such order exists an error-message is given.

Example: the data-declarations

```
'constant'p=q.
'constant'q=3.
```

are processed in reverse order, whereas the data-declarations

```
'constant'p=q.
'constant'q=2-p.
```

will result in an error-message.

A large part of the processing of the data-declarations 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 (3.5) as the sole 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 (3.6), the termination state is specified by this instruction. If the stop is due a run-time error the termination state is -1.

3.1.2 The use of tags

A tag is a sequence of letter and digits, the first of which is a letter. All tags defined by rule-declarations, pointer- initializations, constant-descriptions, variable-descriptions, table-heads (except those in field-list-packs), stack-heads (except those in field-list-packs), file-descriptions, external-rule-descriptions and external-constant-descriptions must differ from each other.

3.2 Rules

The declarations and applications of rules 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 given in this manual summarized in 3.9.2.

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.
rule tag:
    tag.

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 designated by typer, to be identified by the 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 designated 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 *either* always succeed *or* be capable of failing depending on the logical construction of the actual-rule,
b. a rule, when succeeding, may or may not have side effect, again depending on the logical construction of the actual-rule.

These criteria are elaborated upon in 3.9.

- A rule is *predicate* if it can fail and has side effect (the restriction on the structure of rules prevent these side effects from becoming effective if the rule fails).
- A rule is a *question* if it can fail and has no side effect.
- A rule is an *action* if it will always succeed and has side effects.
- A rule is a *function* 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-declaration is given, together with a summary of what the 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; `plus+res+d+res` is a member and `+` is a last-member.

When an actual-rule is executed (through a call (3.5) 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-affixes, one location of 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 alternatives applies in the present case. The applicable alternative is the (textually) first alternative whose "key" succeeds. The "key" of an alternative is its first member, or it it has no member, its terminator. Thus the key of the first alternative is executed: if it succeeds, the first alternative applies. Otherwise the key of the second alternative is executed: if it succeeds, the second alternative applies, etc. If none of the keys succeeds, the alternative-series fails.

The alternative found applicable is then elaborated further. Its key has already been executed. Now the rest of its members and last-member are executed in textual order until one of two situations is reached:

either all its membersss and its last-member have succeeded, in which which case the alternative-series succeeds as well,

or a member or last-member fails: any (textually) following members or last-member 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.

For the execution of a classification see 3.8.

After the result of the actual-rule thus has been assessed, the space for the local-affixes is removed form the run-time stack.

Restrictions.

An alternative-series must satisfy the following restrictions:

a. If the key of an alternative cannot fail (3.9.2), the alternative must be the last one. This restriction ensures that all alternatives can, in principle, be reached. Violation of the restriction causes an error message.
b. If an alternative contains a member that has side-effects (see 3.9.1) 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 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 consists in its turn 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 last 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: [formal field list pack], table tag, sub bus.
formal stack: sub bus, [formal field list pack], stack tag, sub bus.
sub bus: sub symbol, bus symbol.
formal field list pack: open symbol, [field list], close symbol.
formal file: quote image, file tag.

Example of a formal-affix-sequence:

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

The formal-affix-sequence defines the number and types of the formal-affixes of the rule it belongs to.

A formal-variable describes a 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 passed back to calling rule: it must have the attribute *initialized* at the end of each alternative of the actual-rule which does not end in a jump, exit or failure-symbol.

A formal-stack describes a stack. If the formal-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 describes a table. If the formal-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 describes a file.

All variable-, stack-, table- and file-tags 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-affixes occur in affix-forms 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; list tag; file tag.

Example of an actual affix sequence:

```
+511+/?/+alpha+beta*gamma[p]+<>list+?
```

In this example `511` is an integer-denotation, `/?/` is a character-denotation, `alpha` is a file-tag, `beta*gamma[p]` may be a stack-element, `<>list` is a calibre and `?` is a dummy-symbol.

Actual-affixes derive their exact meanings from the corresponding formal-affixes. The interrelations are discussed in 3.5 (affix-forms and 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 describes a 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. On exit from that actual-rule or compound-member this space is removed.

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.

All variable-tags in a local-affix-sequence L must be different. Furthermore, all variable-tags in L must be different from:

a. all the rule-tags, if any, and all variable-tags in the local-affix-sequences, if any, of all the compound-members, if any, in which L is contained,
b. the rule-tag and all variable-, stack-, table- and file-tags 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, variable directive sequence.

source: constant; variable.
constant: plain value; table element.
plain value:
    integral denotation; character denotation; constant tag; limit.
integral denotation: [integral denotation], digit.
character denotation: absolute symbol, character, absolute symbol.
variable: variable tag; stack element; dummy symbol.
table element:
    [selector, of symbol], table tag, sub symbol, source, bus symbol.
stack element:
    [selector, of symbol], stack tag, sub symbol, source, bus symbol.

variable directive sequence: variable directive; [variable directive sequence].
variable directive: to token, variable.
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 directive sequence.
selector directive sequence:
    selector directive, [selector directive sequence].
selector directive: to token, selector.

Example of a transport:

```
pnt->sel*list[q]->offset->ors*list[offset]
```

Example of an identity:

```
ext*list[pnt]=nil
```

Example of an extension:

```
* pnt->sel,nil->ect->ors *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 source. A source is evaluated as follows.

If the source is an integral-denotation, its value is the numeric value of the sequence of digits, 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 constant-tag or a variable-tag, its value is the value of the constant or variable identified. If a formal or local variable is identified, it must have the attribute *initialized*.

If the source is a stack-element, or a table-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. We call the stack-tag or table-tag in front of the sub-symbol T, and the (global or formal) list identified by it L. We now consider the block in L that has an address equal to P (if no such block exists, there is an error); it is called B. Subsequently a selector S is determined: if the of-symbol is present, S is the selector in front of it; if the of symbol is absent, S is T. (As en example, `list[p]` is equivalent to `list*list[p]`.) S must be a selector of L. Now the value of the stack-element or table-element is the value in the block B indicated by the selector S..

If the source is a limit, its value is described in 4.1.7.

if the source is a dummy-symbol, there is an error.

The value of the source is called V. Now the variable-directives of the transport are executed in textual order. A variable-directive is executed as follows.

If its variable is a variable-tag, V is put in the location of the variable identified. If a formal or local variable is identified, this variable has the attribute *initialized* int he rest of the alternative in this the transport appears.

If its variable is a stack-element, the source between the sub-symbol and bus-symbol is evaluated and its value is called P. We call the stack identified by the stack-tag L. We now consider the block in L that has an address equal to P (if no such block exists, there is an error); it is called B. Subsequently a selector S is determined: if the of-symbol is present, the S is the selector in front of it; if the of-symbol is absent, S is the stack-tag. S must be a selector of L. Now v is put in the location in block B identified by the selector S.

if the variable is a dummy-symbol, the variable-directive is a dummy action.

Examples:

`0->cnt->res`	now `cnt` and `res` are both zero
`p->list[q]->q`	the value of `p` is put in the location identified by `list*list[q]` and in (the location of) `q`
`p->q->list[q]`	the value of `p` is put in (the location of) `q` and then in the location identified by `list*list[q]` which is now the same as `list*list[p]`
`list[p]->p->list[p]`	the value of `list*list[p]` is put in `p` and then put in `list*list[p]` using the new value of `p`, with the result that now `list*list[p]` contains a pointer to itself

3.4.2 Identities

An identity can be considered a question, i.e., it has no side effects and may either succeed or fail.

Both its sources 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 of 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.

Call the stack indicated by the stack-tag S. The selectors that appear in the field-transport-list must be selectors of S.

Firs the sources in the field transports are evaluated as described in 3.4.1 and their values remembered. Subsequently the stack is extended to the right with one block B of empty locations (whence the name *extension*); the number of locations in this block is equal to the calibre of S. Net the field transports are executed; a field-transport is executed by putting the value remembered for its source in the location(s) in B identified by its selectors.

No more than one value may be put in a given location in B; at the end of the extension all locations in B must have 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 `st` declared as `[](sel,ect,ors)st` then the extension

```
* 3-><ext, 5->sel->ors *st
```

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

3.5 Affix form

Syntax (see also 3.3.2::

affix form: rule tag, [actual affix sequence].

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-affixes as supplied by the affix-form and the formal-affixes 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 list-tag identifying 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 field-list the calibres of the formal and actual must be equal; the selectors may differ. If the formal has no field-list, no calibre match is required. Regardless of mismatches, the value delivered by the calibre (`<>list`) is the calibre of the global list to which the formal table corresponds, directly or indirectly.

The actual corresponding to a formal-stack must be a stack-tag identifying a (global or formal) stack. All actions performed on the formal are executed directly on the actual. If the formal has a field-list the calibres of the formal and actual must be equal; the the selectors may differ. If the formal ha no field-list, no calibre match is required. Regardless of mismatches, the value delivered by the calibre 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 file-tag identifying 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 which is a formal-variable starting with a right-symbol a transport is executed with the actual as a source and the variable-tag of the formal as it variable.

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 variable-tag of the formal as its source and the actual as its variable, in the order in which the affixes appear.

Example: Suppose the following rules are defined:

```
'question'if a: $ some question $.
'question'if b: $ another question $.
'function'give value1+n>: 1->n.
'function'give value2+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 value1+loc, use value+loc, print+loc;
      if b, give value2+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 keeps the attribute *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 has the attribute *uninitialized* (still *uninitialized*, not again *uninitialized*, since, if the beginning of the second alternative is reached, the initialization in the previous alternative will not have taken place). It keeps the attribute *uninitialized* until the call of `give value2` after (and by) which it obtains the attribute *initialized*. Its subsequent application in `use value` is correct.

The actual-rule

```
-loc:if a, use value+loc, give valu1+loc, print+loc
```

is incorrect. `loc` is still has the attribute *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 terminators:

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

3.6.1 Jumps

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

A jump to the rule-tag of a 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`, the affix-form `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`, the affix-form `c` will be executed. If the parentheses are removed, the rule is correct.

A jump to the rule-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 rule until C turns into a rule. Then the above applies.

3.6.2 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.

3.6.3 Success and failure symbols

The execution of a success-symbol always succeeds, the execution of a failure-symbol always fails. Neither has side-effects.

3.7 Compound members

Compound-members serve to turn a (composite) rule-bodys 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 of C, if present,
b. if the list is empty, the formal-affix-sequence is empty,
c. for each tag in the list, if the corresponding item
   1. is used as a source (either directly or through the affix mechanism) and is used as variable (either directly or through the affix mechanism), 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 the affix mechanism), it is entered into the formal-affix-sequence preceded by a right-symbol,
   3. is used as a variable (either directly or through the affix mechanism), 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-affix-sequence as a 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-symbols.

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 rule-tag, if any, in a compound-member C must be different from:

a. the rule-tags, if any, and all the variable-tags in the local-affix-sequences, if any, of all the compound-members, if any, in which C occurs,
b. the rule-tag and all the variable-tags, stack-tags, table-tags and file-tags in the formal-affix-sequence, if any, of the rule-declaration in which C occurs.

3.8 Classification

A classification is similar to an alternative-series in that both specify a series of alternatives only one of which will eventually apply. The difference is twofold: in a classification exactly one alternative applies (as opposed to one or zero 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 keys). Classifications allow fast selection of alternatives at the cost of less versatile selection mechanism.

Syntax:

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.

Example 1:

```
(n:get+char,
   (=char=
    [/0/:/9/], dgt->type;
    [/a/:/z/;/a/+cap:/z/+cap],ltr->type;
    [/+/;/-/;/*/;///],op->type;
    [0;127], :n;
    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 areas 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 zones. 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 X.
- 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 `mint 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 is $L \leq V \leq R$.
- If the zone is a list-tag, this list-tag must identify a global (*not* formal) list. V belongs in the zone if it is an address in the virtual address space (4.1.4) of that list.

Areas may coincide partially or totally; the textually first area takes precedence.

The exact size and location of all zones is known at compile time; this information can be utilized by the compiler.

A classification can fail if at least on of its alternatives can fail, it has side-effects if at least one of its alternatives has side-effects.

3.9 Criteria for side effects and failing

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 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 an action or predicate or
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 variable(s) identifies a global variable or is a stack-element.

3.9.2 Criteria for failure

A member can fail if

1. it is an affix-form the rule of which is a predicate or question,
2. it is an identity,
3. it is a compound-member the rule-body of which can fail.

a terminator can fail if

1. it is a failure-symbol (–) or
2. it is a jump to a rule or compound-member that can fail.

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

An alternative series can fail if

1. the key of its last alternative can fail,
2. it contains an alternative that contains a member or terminator, other than its key, that can fail.

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 location 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 type 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, identityes and extensions. Additional integer handling can be done through externals.

There are no primitive action of 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, plusminus, term.
term: [term, times by], base.
base: plain value; expression pack.
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` $\neq$ `0`) is a value `n` such that `p-n*q` is non-negative and minimal (so, e.g., `7/3=2`, `7/(-3)=-2`, `(-7)/3=-3` and `(-7)/(-3)=3`).

A constant-tag defined in a user=defined external-constant-declaration cannot be used in an expression.

The list-tag in a min-limit or max-limit (see 4.1.7 used in an expression must identify a (global) table, i.e., limits of stacks cannot be used in expressions.

4.1.2 Constants

A *constant* consists of a constant-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: constant tag, equals symbol, expression.
constant tag: tag.

Example:

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

The value of the expression must not depend on the constant-tag being declared. That is,

```
'constant' p=q, q=2-p.
```

is not allowed.

Constants can be used in expressions and in sources.

4.1.3 Variables

A *variable* consists of a variable-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: variable tag, equals symbol, expression.
variable tag: tag.

Examples:

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

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

Variables can be used in sources and in destinations. They cannot be used in expressions.

4.1.4 The address space

In addition to constants and variables lists of constants (*tables*) and lists of variables (*stacks*) exist. 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 ranges from a (large) negative number to a (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 though 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 exists 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, and 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 if

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. *Deleted*; see 5.2.4 for `nil` and `nil table`.
b. For each table of stack without size-estimate L the size of the actual address space is calculated from its filling-list and L is given a virtual address space of exactly the same size.
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 sizes.

For each list L 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 L is called *virtual min limit*; the size of its actual address space is calculated form its filling list and the actual address space is positioned at the left end in the virtual address space. The `max limit` of L is made equal to the right-most address in the actual address space; the `min limit` of L is made equal to the *virtual min limit*.

If the actual address space has length zero, the `max limit` of L is equal to the `min limit` minus the `calibre` of L.

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

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 =
                ((365,113):pi, (191,71):e).
```

the virtual address space could have the following layout:

address	contents	belongs to	selector	pointer
-16	—	—	—	nil
-15	1	powers	powers	<<powers
-14	10	"	"	
-13	100	"	"	
-12	1000	"	"	>>powers
-11	0	digits	digits	<<digits, >>digits
-10	—	"	"	
-9	—	"	"	
-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	—	"	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 toward the right (raising the `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 form 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 a stack fashion (*add to the right end / remove from the right end*) or in queue fashion (*add to the right end / remove from the left end*). Queue-operation consumes virtual address space but in most implementations virtual address space will be virtually unlimited.

Usually an actual address space correspond 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, part of it may be on background memory.

4.1.5 Tables

Tables originate from table declarations. 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], 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.

Examples:

```
'table'messages =
   ("tag undefined": bad tag,
    "wrong number of parameters": wrong  parameters,
    "quote "" where not allowed"" bad quote).
'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 The table head

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 ks 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 same is that of the table-head, 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 tags in a field list pack must differ one from another.

The `calibre` C of a list is the number of fields 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 fields 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, 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 in it is defined as having the value of the address of the block.

If the filling is a compound-block, the number of expressions 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 in it 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 ion a machine-dependent way. If a pointer-initialization is present the constant-tag in

it is defined as having the value of the largest address in the generated list of blocks.

The string denoted by a string-denotation consists of the characters which are the representation of its string-items, if any, except that for each quote-image the representation of the quote-symbol is taken. Spaces are considered string items, new-line control characters are not, since the dividing into lines is done through the charfile-handling externals (see 5.2.5).

Example 1: The table-declaration for `for 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 or file-description.

4.1.6 Stacks

Stacks original from stacks declarations.
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], 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.

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,  sun   ): 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 is 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 compiles and are unique to the given stack.

The value of the expression in the size-estimate must not depend, directly or indirectly, on the value of any constant-tag defined in a pointer-initialization.

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 the 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 designate a zone in a classification or file-description.

4.1.7 Limits

Syntax:

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.

Examples:

```
<<stack, >>table, <>blocked
```

A min-limit (max-limit, calibre) has the value of the `min limit` (`max limit`, `calibre`) or the list identified by the list-tag. The value of a limit is a constant in that it cannot be changed by a transport. 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 values of the min-limit and the max-limit of tables and the calibre of all lists are invariable.

4.2 Files

File originate form file declarations. 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 description: 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.

Examples:

```
'charfile'printer="output">, backward cards=>"qelet,invert".
'datafile'tagfile[tag;link;0:]= >"systags">,
     bin card[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 is the desired way. It might for example contain: the external file name, 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 `[:]` is assumed.

Files are read and written sequentially. The can be reset to the beginning of the file and be reread or rewritten. The file ends after the last item written or else after the last item produced by the preprocessing.

4.2.1 Charfiles

label4.2.1 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 printer over the previous line and those of type `new page` will be printed on the first line of the next page; `rest line` serves administrative 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 consists of two lines:

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

The standard externals allow two ways of processing a charfile.

a. linewise: each call of `'predicate'get line+""charfile+[]stack[]+cint>` puts the next line on `stack` (the last character on the line is the rightmost item in the stack) and yields the control integer in `cint`. It will fail if there is no next line.
b. characterwise: each call of `'predicate'get char+""charfile+char>` yields the next item form 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.q. `[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 indications 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 of the file in the following way.

If the `type` is `numerical`, there must be a zone in the area which is not a tag identifying a list, such that the value of `item` 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 be an address in the virtual address space of a list whose list-tag is a zone in the area. tThe 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 `file`, 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 was 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 was 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:
    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:
    table head, 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, equal symbol, string denotation.

Example:

```
'external''function'converto to hash+t[]+>p+h>="subtr, convertt".
'external''table' conv 2 ebcdic="adde, 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 rule-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; it is the implementer's responsibility to see to it that these actions are in accordance with the type of the rule and that no-sideeffect will occur when a call of the rule fails.

An external-table-description defines a table to be known internally under the name given by the table-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 constant-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 name can be redeclared by the user.

5.2.1 Integers

- `'constant' zero, one, max int, min int, int size.`
  `zero` has the value 0, `one` has 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`: $\texttt{a} + \texttt{b} = \texttt{head} \times (\texttt{max int} + 1) + \texttt{tail}$, such that $|\texttt{head}|$ is minimal.
- `'function'subtr+>a+>b+head>+tail>.`
  The double-length difference of `a` and `b` is given in `head` and `tail`: $\texttt{a} - \texttt{b} = \texttt{head} \times (\texttt{max int} + 1) + \texttt{tail}$, such that $|\texttt{head}|$ is minimal.
- `'function'mult+>a+>b+head>+tail>.`
  The double-length product of `a` and `b` is given in `head` and `tail`: $\texttt{a} \times \texttt{b} = \texttt{head} \times (\texttt{max int} + 1) + \texttt{tail}$, such that $|\texttt{head}|$ 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`: $\texttt{a} = \texttt{b} \times \texttt{quot} + \texttt{rem}$, such that `rem` is non-negative and minimal. `b` must not be zero.
- `'function'plus+>a+>b+c>.`
  The sum of `a` and `b` is given in `c`.
- `'function'minus+>a+>b+c>.`
  The difference of `a` and `b` (i.e., `a-b`) is given in `c`.
- `'function'times+>a+>b+c>.`
  The product of `a` and `b` is given in `c`.
- `'function'incr+>x>.`
  The value of `x` is increased by 1.
- `'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.`
  Succeeds 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`, $\texttt{p} \leq \texttt{r} \leq \texttt{q}$. The value of `r` is derived from an element in a uniformly distributed 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 $\texttt{a} = \texttt{root} \times \texttt{root} + \texttt{rem}$, 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 `amount1-amount2` 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) form `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` constants a 0, and 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'boor 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.
- `'function'left circ+>x>+>n.`
  The bit-array in `x` is shifted `n` position 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`.
- `'function'left clear+>x>+>n.`
  The bit-array in `x` is shifted `n` position 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`.
- `'function'right circ+>x>+>n.`
  The bit-array in `x` is shifted `n` position 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`.
- `'function'right clear+>x>+>n.`
  The bit-array in `x` is shifted `n` position 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`.
- `'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 $\leq$ 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.
- 'function'set elem+>x>+>n.
  The n-th bit in x is made equal to 1. It is required that $0 \leq$ n $\leq$ word size.
- 'function'clear elem+>x>+>n.
  The n-th bit in x is made equal to 0. It is required that $0 \leq$ n $\leq$ word size.
- '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, 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.
- '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 $<$ word size.

5.2.3 Strings

For those data 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 ≤ 127.
- 'action'pack string+from[]+>n+[]to[].
  The right-most n elements of from must be values that correspond the characters. These characters are packed, in same 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 strings 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.

- `'action'unpack string+from[]+>p+[]to[].`
  The pointer `p` most 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 on one location each, containing these values in left-to-right order.
- `'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.
- `'function'string length+text[]+>p+n>.`
  The pointer `p` must point into `text` and be the address of a packed string. The number of character in this string is yielded in `n`.
- `'function'compare string+t1[]+>p1+t2[]+>p2+trit>.`
  The pointer `p1` must point into `t1` and be the address of a packed string, s_1. The pointer `p2` must point into `t2` and be the address of a packed string, s_2. These two strings are compared in some way: if s_1 is smaller than (lexicographically comes before) s_2, `trit` is set to `-1`; if they are equal, `trit` is set to `0`; otherwise `trit` is set to `1`.
- `'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`.
- `'action'previous string+t[]+>pnt>`
  The pointer `pnt` most point into `t` and be the address of a packed string; it is made to point to the (possibly non-existing) block just preceding the string.
- `'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 fails.

5.2.4 Lists

For lists the following externals are available.

- `'constant'nil.`
  `nil` is a value that points into the standard table `nil table`.
- `'table'nil table.`
  Contains one entry, `nil`, pointed at by `nil`.
- `'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 location 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` is `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 block 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 block in `st` are removed. Their locations can be reclaimed through extensions, their contents are lost.
- `'action'delete+[]st[].`
  All block in `st` are removed, as in a call of `scratch`. Moreover, the runrime 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 charfile. If the file is exhausted, `get line` fails. Otherwise the next item in `file` is read; 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 line. The stack `st` is extended with these character 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 the file `file`, containing the character 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.
- `'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 i either reached the end of 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 if 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, and 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 character are: zero or more spaces, the sign of `int` and the character of the decimal representation of the absolute value in `int` without leading zeroes.
- `'constant'numerical, pointer.`
  These constants are predefined values that can be used as type indications in datafiles. For their meaning 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.

- `'predicate'back file+""file.`
  If there is not yet a last item read, `back file` fails. Otherwise it succeeds and the file is repositioned to beginning of the file.

6 Pragmats

Pragmats are used to control certain aspect of the compilation (compiler-pragmats and to supply implementation-dependent information to the machine-dependent part of the compiler (user-pragmats). The exact position of a compiler 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;
    tag, equals symbol, pragmat value;
    tag, equals symbol, pragmat value list pack.
pragmat value:
    tag;
    integral denotation;
    string denotation.
pragmat value list pack:
    open symbol, pragmat value list, close symbol.
pragmat value list:
    pragmat value, [comma symbol, pragmat value list].

Example:

```
'pragmat'title="aleph compiler",
    background=(numb adm, history),
    macro=(convert 1 to 2 compl, set all bits).
```

Before the meaning of a pragmat is determined, it is preprocessed: all pragmat-value-list-packs are removed in the following way.

For every pragmat-value-list-pack which is preceded by an equals-symbol preceded by a tag, the equals-symbol and tag are removed and inserted in front of each pragmat-value in the pragmat-value-list-pack. Subsequently all open-symbols and close-symbols are removed.

This the pragmat-item `background=(numb adm,history)` has the same meaning as `background=number adm, background=history`.

6.1 Compiler-pragmats

The tags `background`, `compile`, `count`, `dump`, `first col`, `last col`, `macro` and `title` identify compiler-pragmats.

- `background`=list-tag
  The identified list will be kept on background memory if possible and necessary. The position of this pragmat is immaterial.
- `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 applications of this pragmat can render a correct program incorrect.
  `on`: normal compilation is resumed.
  `all`: subsequent pragmat-items 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`, expect 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. The standard option is `member`.
- `first col`=integral-denotation
  Call the value of the integral-denotation i. The first $i-1$ character 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`=rule-tag
  The rule-tag must identify a non-recursive rule. Calls of this rule will be implemented through textual substitution rather than by subroutine call. The rule-tag may not be the rule-tag of the affix-form of the root. This pragmat must occur before the declaration of the affected rule.
- `title`=string-denotation
  The string-denotation is the title of the program. The default title is empty.

6.2 External-pragmats

Deleted.

6.3 User-pragmats

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

7 The representation of program

7.1 The program

The program produced by the notion program consists of a series of terminal symbols. Into this program comments may be inserted in the following way. The program is considered as a sequence of the following units:

tags,
integral denotations,
character denotations,
symbols not occurring in one of the above.

Spaces may be added in from of all these units and inside tags and integral denotations. Long comments may be added in form of all these units. A long comment consist 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 from of all units except tags and integral denotations. 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 as described in 7.2 (e.g., root-symbol turns into `'root'`). 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 an 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 programs `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

Almost all terminal symbols of the ALEPH grammar are notions that end in -symbol. The exceptions are tag, character and non-quote-item. A tag is represented by a non-empty sequence of small letters and / or digits, the first of which is a small letter: two tags are equal if their representation consist of equal sequences. A digit is represented by one of the digits `0` ... `9`. A character is represented by any character in the available character set except the new-line control character. A non-quote-item has as its representation any representation of character with the exception of the representation of the quote-symbol.

The representations of the other terminal symbols can found in the following table.

symbol	representation
absolute symbol	`/`
action symbol	`'action'` or `'act'`
actual affix symbol	`+`
box symbol	`=`
bus symbol	`]`
by symbol	`/`
charfile symbol	`'charfile'`
close symbol	`)`
colon symbol	`:`
comma symbol	`,`
constant symbol	`'constant'` or `'cst'`
datafile symbol	`'datafile'`
dummy symbol	`?`
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	[
success symbol	+
table symbol	'table'
times symbol	*
up to symbol	:
variable symbol	'variable' or 'var'

8 Examples

8.1 Towers of Hanoi

```
$ 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:
   putchar+print+s1,put char+print+s2,put char+print+/ /.
'root'move tower+6+/a/+/b/+/c/.
'end'
```

8.2 Printing Towers of Hanoi

```
$ 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,printdisc+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+/*/,repeat+1+/*/,
   repeat+nmb+/*/,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, * n->a *a, :fill a),
   print towers,move tower+size+a+b+c.
'root'play towers.
'end'
```

8.3 Symbolic differentiation

```
$ 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,"*":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'[1]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;
   left*expr[e]->f, right*expr[e]->g,
   derivative+f+df, derivative+g+dg,
     (=op*expr[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;
   op*expr[e]->zz,
   (= zz =
     [plus op; min op; tim op; div op],
        put char+out+/(/,print expr+left*expr[e],
        put char+out+/)/,put string+out+operator+zz,
        put char+out+/(/,print expr+right*expr[e],
        put char+out+/)/;
     put string+out+operator+zz,put char+out+/(/,
        (equal+zz+pow op, print expr+left*expr[e],
             put char+out+/,/; +),
        print expr+right*expr[e], put char+out+/)/
   ).
'action'test-e1-e2-e3:
   gen node+pow op+x var+x var+e1, $ pow(x,x) $
      print expr+e1, nl,
      derivative+e1+e2, print expr+e2, nl,
      derivative+e2+e3, print expr+e3, nl,
   gen node+div op+x var+x var+e1, $ x/x $
      print expr+e1, nl,
      derivative+e1+e2, print expr+e2, nl,
      derivative+e2+e3, print expr+e3, nl.
'action'gen node+>op+>left+>right+res>:
   * op->op, left->left, right->right *expr, >>expr->res.
'action' nl: put char+out+new line.
'root'test.
'end'
```

8.4 Quicksort

```
'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", C.A.R.Hoare, 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]->amiddle,
  (split:
     (push right: more+left+to;
                  more+a[left]+amiddle;
                  incr+left,:push right),
     (push left:  more+from+right;
                  more+amiddle+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],amiddle->a[right],decr+right;
      more+middle+left,
         a[left]->a[middle],amiddle->a[left],incr+left;
     +)
  ),
  quicksort+from+right+a, quicksort+left+to+a.
```

8.5 Permutations

```
$ "next perm" considers the rightmost "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.
'predicate'next perm+>i+[]st[]-p:
  less+i+>>st,plus+i+1+p,
    (next perm+p+st;
     less+st[i]+st[p],simple perm+i+st
    ).
'action'simple perm+>i+[]st[]-p-q:
  $ the rightmost "i" elements of "st" do have a next permutation
  $ but the rightmost "i-1" don't
  >>st->q,
   (find new ith elem:
     less+st[q]+st[i],decr+q,:find new ith elem;+),
   swap+st+i+q,
   plus+i+1+p,>>st->q,
   (invert perm tail:
     mreq+p+q;swap+st+p+q,incr+p,decr+q,:invert perm tail).
'action'swap+[]st[]+>i1+>i2-elem:
  st[i1]->elem,st[i2]->st[i1],elem->st[i2].
```

```
'stack' st=(/1/,/2/,/3/,/4/).
'root'display perms+st.
'action'display perms+[]st[]:
  put line+output+st+new line,
    (next perm+<<st+st,:display perms;+).
'charfile'output="output">.
'end'
```

9 References in the manual

[1] A. P. W. Böhm, ALICE: An Exercise in Program Portability, IW 91/77, *Mathematical Centre*, Amsterdam, 1977

[2] R. Glandorf, D. Grune, J. Verhagen, A W-grammar of ALEPH,IW 100/78, *Mathematical Centre*, Amsterdam, 1978

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

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

[5] D. A. Watt, The parsing problem for affix grammars, *Acta Inf.* **8**, pp 1–20, 1977

[6] B. A. Wichmann, How to call procedures, or second thoughts on Ackermann's function, *Software – Practica & Experience*, **7** pp 317–329 (1977)