Aleph Manual

Laszlo Csirmaz

April 29, 2025

Preface

ALEPH – an acronym for "A Language Encouraging Program Hierarchy" – is a high-level programming language designed to provide a tool that effectively helps programmers to structure their programs in a hierarchical fashion. ALEPH is suitable for problems that suggest top-down analysis, like parsers, search algorithms, combinatorial problems, artificial intelligence problems, etc. Probably the most interesting feature of the ALEPH language is its unique "copy, execute, copy" parameter passing method. This method prevents modifying parameter values when the called routine fails as well as it postpones side effects caused by global parameters until a successful call is completed. Another ALEPH feature is modeling the complete computer memory as a single (virtual) list of computer words indexed by integers. Continuous segments of this virtual memory are populated by stacks and tables so that an index (pointer) determines the stack or table it points to. Stacks can grow and shrink within their virtual limits which are assigned during compilation. This memory model allows transferring pointers seamlessly from one ALEPH program to another.

Chapter 1 of this Manual gives a tutorial introduction into the basics of ALEPH. It is written in a style which can be followed easily by novice programmes. Chapters 2 to 7 contain a semi-formal definition of the language. Chapter 3 treats *rules*, and the control flow in them. Chapter 4 is about data types, while *externals* are treated in Chapter 5. Chapter 6 describes *pragmats*. The global structure of an ALEPH program, including how the source can be split into files and modules are discussed in Chapter 7. Elements of the Standard Library in Chapter 8, program representation in Chapter 9, and sample programs in Chapter 10 close the Manual.

This version describes the language as it has been implemented in [1]. The Manual is based on, and repeats large parts of the fourth printing from 1982 [2]. Many paragraphs have been kept identical, while many others were rewritten to reflect the implemented language.

Contents

1 Informal introduction to Aleph

This chapter gradually develops a small ALEPH program, interspersing it liberally with annotations and arguments. This introduction is intended to give some insight into the use of the language ALEPH and to display its main features in a very informal way.

1.1 The problem and its grammar

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 unsigned integers, plus symbols, multiplication symbols and parentheses; and example might be `15 * (12 + 3 * 9)`.

First we put the requirements for the input to our program in a more formal 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 be read 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 now write a series of ALEPH rules, one for each rule in the grammar. For the grammar rule expression we 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,add+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 these 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 will 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 will 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, meaning `expression+r` is called (recursively), yielding its result in `r`, and subsequently `add+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. `add` 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 next alternative after the semicolon is tried. This alternative consists of a single `+` 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 programmer. The language provides a framework that controls which rules are called depending on the answers obtained from other rules. The programmer 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 (as defined by the programmer) embedded in a framework that controls their order (which is 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 some basic primitives in the language, such as copying of a value, comparing

two values, and extending a stack by a fixed number of given values. Often, however, these primitives are not sufficient. In such cases the rule is declared to be `'external'`, and its actions are specified in a different way. A number of such external rules are predefined, including the rule `add` used above. This predefined set of rules probably suffice for most applications.

We 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 as 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 since `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 it and issue an error message. The `+` sign and the term *affix* stem from the theory of affix grammars of which ALEPH is based on [3, 4].

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. The semicolon separates alternatives, which are checked in textual order. As said before, the stand-alone `+` denotes the dummy action that always succeeds. Finally, the period ends the rule.

1.3 Further rules

In view of the above, the rule for term should not be a surprise:

```
'action'term+res>-r:
   primary+res,
    (is symbol+/*/, term+r, mult+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 is incorrect; we now decide that we do not do any error recovery, instead give an error message and stop the program. The modified version of the `primary` rule 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 a rule for reading an integer is trickier than it seems to be. Our version consists of two rules and is about as complicated as necessary.

```
'action'integer+res>:
   digit+res,integer1+res;
   error+no int.
'action'integer1+>res>-d:
   digit+d,mult+res+10+res,add+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 is issued. The rule `integer` must assign a value to `res` under all circumstances. This requirement is not violated here as `error` will be specified as a non-returning, `'exit'` rule.

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 rule `get int` from the Standard Library.

1.4 Input

The above forms the heart of our program. We now supply some input and output definitions. For input we will use the predefined ALEPH character file `STDIN`, which provides the characters entered at the console. Additionally, the global variable `buff` will contain the first character not yet processed. Comments in ALEPH start with a `$` and end with another `$` or at the end of the line, whichever comes first.

```
$ Input
'variable'buff=/ /.
```

The variable `buff` is initialized with the code for the space character (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/], subtr+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. To give an example, "*Are there plane tickets to New York for less than $1000?*" is a non-committal question, whereas "*Are there plane tickets for New York for less than $1000? 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 character into* `buff`. The form `buff=n` is a test for equality, it is one of the primitive operations in ALEPH. The rule `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. Classes are specified by the values presented in square brackets, for each possible value of `buff` they specify which of the alternatives will be chosen. Thus for values of `buff` that lie between the code for `0` and the code for `9` the first alternative is chosen. For all other values the next alternative—consisting of the dummy question that always fails, denoted by the dash—will be chosen. The rule `digit` is equivalent to

```
'predicate'digit+d>:
   between+/0/+buff+/9/,subtr+buff+/0/+d,get next symbol.
```

assuming that `between+/0/+buff+/9/` succeeds if and only if the value of `buff` is between `/0/` and `/9/`. Classification is analogous to the case statement in other programming languages.

One more input rule must be supplied:

```
'action'get next symbol:
   get char+STDIN+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 character from the ALEPH file indicated by its first affix, which is `STDIN`. If

there is a character, it puts it in its second affix (here `buff`); if there are no more characters, the rule fails. In the latter case `buff` is given the value `stop`, which is defined in a constant-declaration to be `-1`.

If `get char` returns a character and if this is a space or a newline, `get char` is called again. We've used nested parenthesizing. This definition of `get next symbol` implies that we have decided that spaces and newlines are allowed in the input in all positions (a decision that was not present in the initial grammar).

1.5 Output

The output is as follows:

```
$ output
'action'print integer+>int:
   out integer+int,put char+STDOUT+newline.
'action'out integer+>int-rem:
   divrem+int+10+int+rem,add+rem+/0/+rem,
   (int=0;out integer+int),put char+STDOUT+rem.
```

The rule `put char` is known to the compiler, as are `STDOUT` and `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 to `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 done by the recursive call of `out integer`. Subsequently, the last digit is printed through a call of `put char`. This is a simple way of printing a number. A more convenient way of printing an integer is provided by the Standard Library rule `print int`.

The predefined character file `STDOUT` sends characters to the console directly.

For the printing of error messages we will 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 into stacks and tables and are operated upon by suitably defined rules (in this case supplied by the system).

The error handler takes the following form:

```
$ Error-message printing
'exit'error+>er:
   put string+STDOUT+strings+er,put char+STDOUT+newline,
     exit+1.
'table'strings[]=
  ("Right parenthesis missing":no paren,
   "Integer missing":no int
  ).
```

The table `strings` contains two strings stored in compressed form; 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 `STDOUT` for printing.

The rule call `exit+1` never returns, it terminates the program while `1` will be passed to the operating system as an indication what went wrong. This is by

no means the normal program termination: normal program termination ensues when all work is done. The rule `error` is defined to be of type `'exit'` indicating that this rule never returns, in particular it neither succeeds nor fails.

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.
```

At this point 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. Does ALEPH contain 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 root 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. At that moment it is done, and so is `read expressions and print results`. 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 Creating a module

Looking back to the context-free grammar in Section 1.1 specifying the problem which was to be solved, we observe that the auxiliary input and output rules developed in Section 1.4 and Section 1.5 can be used more generally. These rules would remain unchanged if the problem, and, consequently, the grammar describing the problem, changes. Thus we decided to create an ALEPH library module which provides these auxiliary rules "out of the box," so that they can be used without copying them verbatim into the main program. The module will contain three of the rules: `is symbol`, `digit`, and `error`. The rule `get next symbol` and the variable `buff` are used by these rules only, thus they can be local to our module. We choose `base` as the module name. According to the best practice, the name of the file containing the source code of the module should resemble the module name. It is particularly important as ALEPH modules are invoked by specifying file names and not module names. Thus the source text of the module will be put into a file named `base.ale`, where the extension `.ale` indicates that it is an ALEPH file.

In general, an ALEPH module has two parts: the *head* which specifies which rules (and other constructs) this module provides, and the *body* which contains the realization of these resources. When the module is used, only the head part is consulted, so this part must provide complete information on the provided items. When the module is compiled, both parts are processed, letting the compiler check that all promised resources are indeed defined. Modules use the ALEPH pragmat facility extensively to control these aspects of compilation.

The overall structure of an ALEPH module typically starts with a pragmat specifying the module name. It is followed by the module head, followed by the body. The latter one is enclosed between the conditional pragmats `if=compile` and `endif=compile`. Accordingly, we start our module with the title and head as

```
'pragmat'module=base. $ module name
  $ module head
  'predicate'is symbol+>n, digit+d>.
  'exit'error+>er.
```

The first line specifies the module name, informing the compiler at the same time that it is reading a module source. The subsequent prototypes (rule heads without rule body) specify the three rules exported by this module. These rule heads convey all information needed when they are used. The affix of `is symbol` will be a character specified by the caller, and `digit` will return the value of the next digit in the affix `d`. The affix of the `error` rule specifies the error message to be printed; it has been chosen to be a pointer to one of the predefined strings in the table `strings`. Our module needs access to this table (when it wants to retrieve the error string), and the main program also needs access to this table (when defining those strings). So the table `strings` will be *declared* in our module; and will be made available to the main program through a table-prototype in the module head, so that the main program can use fillings to populate this table. Thus we also add the line

```
'table'strings.
```

to the head. The lack of the square brackets after the tag indicates that this is a prototype and not a declaration.

The module body is enclosed between conditional pragmats to ensure that it is executed only when the module is compiled. The module body must also contain a root. This root is executed before the main root, so it can perform all necessary initializations. If no initializations are necessary, this root can consist of a sole dummy action, written as `'root'+.` In our case the initialization performed by the `initialize` rule can be transferred here; with this move the main root simplifies to

```
'root'input.
```

Without repeating the rules developed in Section 1.4 and Section 1.5, the overall structure of the body of our module will be similar to the one below.

```
'pragmat'if=compile.   $ start of module body
  'table'strings[].            $ table declaration
  'variable'buff=/ /.          $ only in this module
  'constant'stop=-1.           $ only in this module
  'predicate'is symbol+>n: ... $ rule body
  'predicate'digit+d>:  ...    $ rule body
  'exit'error+>er: ...         $ rule body
  'action'get next symbol: ... $ rule body
  'root'get next symbol.       $ initialization
'pragmat'endif=compile.$ end of module body
'end'
```

The rule `get next symbol`, the variable `buff` and the constant `stop` are declared in the module body, and they are available only in this module. If the main program wants to check whether the input is exhausted (as was done in Section 1.6), a possible solution is to add another rule, without affixes, to the module head, and define it in the body to return success when there are no more input characters, and failure otherwise.

To use items exported by a module, the main program should *require* the module using the "require" pragmat as follows:

```
'pragmat'require="base".
```

This pragmat specifies the source file name (possibly with path information and without extension) enclosed in quote marks, and not the module name. The main program could use rules, variables, constants, tables, etc., exported by the module exactly the same way as if they were declared there. Error strings can be added to the `strings` table exported by our module using fillings, which can spread across the main program, such as

```
'table'strings=("Right parent missing":no paren).
'table'strings=("Integer missing":no int,
                "Extra characters":no end).
```

1.8 The anatomy of a rule

Although the rule `put string` used in error messages 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 quote 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. The right arrow in `0->count` indicates the assignment of the value on the left to the variable on the right, one of the primitive actions in ALEPH. `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 preceded by a colon denotes the re-execution of that rule with the affixes it had upon its initial call (of course this is only allowed inside 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 importantly, 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 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 Section 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, add+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, add+res+r+res;
      +).
```

In a sense this is a more appropriate form than the one given in Section 1.2: now the affix `r` occurs where it belongs, that is, in the position of a local affix of the parenthesized part only. To obtain the version in Section 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,add+res+r+res;+.
```

1.9 Variable number of affixes

An ALEPH rule can be designated to handle an unspecified number of affixes. We present a version of the rule `put string` which can send an unspecified number of strings to an ALEPH file. The new rule could be called as

```
put many strings+file+table1+str1,
put many strings+file+table1+str1+table2+str2+table3+str3,
```

and so on, where `file` is an ALEPH file, while (`table1`, `str1`), (`table2`, `str2`), etc., specify the strings to be printed. Our rule simply calls `put string` with the given `file` and each string in the affixes in turn.

```
'action'put many strings+""file+@+table[]+>string:
   put string+file+table+string,
```

```
(shift affix block+@, put many strings+file+@; +).
```

The *anchor* `+@` in the rule head does not correspond to any actual affix, it only marks the position from which affixes can repeat an unspecified number of times in a call to the rule. Affixes following that anchor form the *repeat affix block*. This block behaves like a window which is aligned, in all the times, with one of the actual affix blocks in the rule call. Formal affixes of the repeat block are matched against the actual affixes appearing in that window. Operations on these formal affixes are performed on the actual affixes seen in the window. As an example, the rule call

```
put string+file+table+string,
```

in the first line of the body of our rule sends that string to `file` whose specification (table and pointer) is actually visible in this window.

When the execution of our rule starts, the window is aligned with the first variable block of affixes in the rule call. Thus the very first call to `put string` will send the first string to `file`. The call to the built-in rule `shift affix block`, as the name suggests, shifts the window to the right by one block, failing if there are no more blocks. Affixes in the shifted out block are lost for this rule as the window cannot be moved backward. If there is no next affix block, then the dummy action `+` finishes our rule. Otherwise it calls itself recursively with the same `file` and the anchor `+@` representing the current (in this case the second, as the first block has already been shifted out) and the still unvisited (if any) affix blocks. As in rule in the previous Section, this trivial right recursion can be written equivalently as the jump `:put many strings`.

Using that an anchor, as the last actual affix, represents the current and all subsequent univisited affix blocks, we could easily write a rule which sends an unspecified number of strings to a particular ALEPH file, say to `STDOUT`, as follows.

```
'action'print strings+@+table[]+>string:
   put many string+STDOUT+@.
```

2 Syntactical description of Aleph

The syntax of the ALEPH language is given in the form of a context-free grammar. This type of grammar follows the well-known scheme: a grammatical notion on the left hand side of the syntax rule is defined by the alternatives on the right hand side. Alternatives are separated by semicolons; successive notions in an alternative are separated by commas. Each grammatical notion is written as one or more words connected with dash characters. To make the grammar more succinct, enclosing one or more notions in square brackets indicates that the group may or may not be present, i.e., it is optional. Similarly, either a notion, or a group ot notions enclosed in parentheses, can be followed by a star * or by a plus +. The star indicates that the notion, or the group of notions, can be repeated zero or more times; while the plus requires at least one, but otherwise arbitrary number of instances. Thus both [notion+] and notion* require zero, one, two, etc., occurrences of notion. To illustrate these conventions, the following part of the ALEPH grammar, which describes the global structure of an ALEPH *unit*

> unit: information*, root, information*, end-symbol.
> information: declaration; prototype; pragmat.

can be verbalized as follows:

> *A* unit *is a sequence of* declaration*s,* prototype*s and* pragmat*s, ending with an* end-symbol*, which contains exactly one* root*, and only the* root *and the closing* end-symbol *are required.*

A syntactically correct ALEPH unit is the character representation of a sequence of terminal notions produced by this grammar; unit*s* are composed from one or more *source files*. These terminal notions fall into the following categories:

a) *symbols*, notions ending with -symbol,
b) *tags* (identifiers), notions ending with -tag, and
c) character, integer, and string *denotations*, ending with -denotation.

Representations of terminals and the description of formatting rules, including where can comments be inserted and how the program text is composed from the source files, are covered in Chapter 9.

There are two kinds of ALEPH unit*s*. A unit is either a *module* or it is a *main program*; module units include library modules. The ALEPH compiler *compiles* a single unit and produces an intermediate `ALICE` code. Complete ALEPH programs are *linked* from the `ALICE` codes of a main program and several modules.

Syntax:

> unit: information*, root, information*, end-symbol.
> information: declaration; prototype; pragmat.
> root: root-symbol, actual-rule, point-symbol.
> declaration: rule-declaration; data-declaration; external-declaration.
> prototype: rule-prototye; data-prototype.

This grammar says that a unit must contain exactly one root, and must end with an end-symbol. The order in which declarations, prototypes and the root appear is immaterial. The position of pragmats, however, can be significant, see Chapter 6. Example of a unit:

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

The first line is a data-declaration, the second line is the root, and the third line contains the end-symbol. For complete examples see Chapter 10.

The unit is a *module* if it contains a `module=...` pragmat; this pragmat defines both the module name and its name-space. The unit is a *main program* if it contains no such a pragmat; the above example is a main program.

A complete ALEPH program is *linked* from a main program and from zero or more modules. Execution of the linked program starts with processing all data-declarations in the main program and in the modules, in such an order that no data item is used before its value has been established. If no such an order exists the linker would have given an error message. After all constants, variables, stacks, tables and files have thus been established, the actual-rules in the roots of the linked modules are executed (Section 3.1.2) in some undefined order, allowing the modules to perform some initialization. If any module root stops prematurely for whatever reason, the program run is terminated. The order in which module roots are executed is undefined, but can be synchronized using the Standard Library rule `wait␣for`, see Section 8.5.

After all module roots finish with normal completion, the actual-rule of the main program is executed. If it reaches its normal completion, the program finishes with a termination state of `0`. If program termination is due to calling the Standard Library rule `exit`, then the termination state is specified by the value of the affix to this rule. In other cases the termination state is not defined.

3 Program logic

Rule definitions and applications constitute the basic mechanism which determines the logical flow in an ALEPH program. This section describes ALEPH primitives and the way how these primitives can be combined to form an actual-rule.

3.1 Rules

A rule-declaration defines what is to be done when the rule is called, whereas applying the rule in an affix-form specifies that the rule is to be called. A rule, when called and returns, will either succeed or fail; while there are rules which never return.

3.1.1 Rule declaration

Each rule in a unit must be defined either in a rule-declaration or in a defining rule-prototype. Syntax:

rule-declaration:
    typer, rule-head, actual-rule, point-symbol.
typer:
    action-symbol; function-symbol; predicate-symbol;
    question-symbol; exit-symbol.
rule-head: rule-tag, formal-affix-sequence.
rule-prototype:
    typer, rule-head, (comma-symbol, rule-head)*, point-symbol.

Example of a rule declaration:

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

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

```
'function'square+>x+xx>,list size+T[]+n>.
```

Both the rule-declaration and the rule-prototype define the rule-tag to be a rule of type designated by typer, to be identified by the rule-tag, and to have the formal affixes given by its formal-affix-sequence. A rule-declaration, in addition, specifies in the actual-rule how to execute the rule when it is called.

There are five rule types: *predicate*, *question*, *action*, *function*, and *exit*, each designated by the corresponding typer symbol. These types arise from three criteria, each expressing a fundamental property of the logical structure of the actual-rule. A rule

a) either always succeeds or is capable of failing;
b) when it succeeds, it may or may not have side effects;
c) can never return.

Accordingly, a rule is of type

predicate, when it can fail *and* has side effects (restrictions on the rule structure typically prevent these side effects from becoming effective when the rule fails);

question, when it can fail *and* has no side effect;

action, when it always succeeds *and* has side effects;

function, when it always succeeds *and* has no side effects;

exit, when it never returns, consequently has side effects.

The rule type is checked against the logical construction of the actual-rule. If an action or function is found to be able to fail, an exit rule to be able to return, or a non-exit rule not able to return, an error message is given. In other cases, if a discrepancy is found, a warning is given.

With each of the following rule-prototype examples a summary of what the rule does is provided. From this explanation it should be clear why the rule was specified with the given type.

`'predicate'digit+d>.`	if the next symbol in the input file is a digit, its value is delivered in `d`, the input file is advanced by one symbol (side effect) and `digit` succeeds; otherwise it fails.
`'question'is hexdigit+>d.`	if `d` is a hexadecimal digit character, the rule succeeds, otherwise it fails.
`'action'skip up to point.`	the input file is advanced until the next symbol is a point.
`'function'add+>x+>y+sum>.`	the sum of `x` and `y` is delivered in `sum`.
`'exit'error+>n.`	give an error message and terminate.

3.1.2 Actual rule

The actual-rule defines the local variables used exclusively by the rule-body, and specifies one or more alternatives which determine the control flow in the rule. Syntax:

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

Example of an actual rule:

```
-d:
  digit+d,mult+res+10+res, add+res+d+res,integer1+res;
  +.
```

Here `-d` is a local-affix, one alternative is

```
digit+d,mult+res+10+res, add+res+d+res, integer1+res
```

and + is another; `add+res+d+res` is a member, and + is a last-member.

When an actual-rule is executed (through a call of the rule of which it is the actual-rule, see Section 3.4), the following takes place.

a) First, space is allocated for every local affix.
b) Second, the rule-body is executed, which means executing its alternatives or its classification.
c) After the result of the actual-rule thus has been assessed, the space reserved for the local affixes is returned.

Execution of alternatives is discussed here, for that of classifications see Section 3.7.

The execution of the alternatives starts with a search to determine which of them applies in the present case. The applicable alternative is the (textually) first one whose *guard* succeeds. The guard of an alternative is its first member, or if it has no member, its terminator. Thus the guard of the first alternative is executed: if it succeeds, the first alternative applies. Otherwise the guard of the second alternative is executed: if it succeeds, the second alternative applies, etc. If none of the guards succeeds, the rule-body fails.

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

either all its members and its last-member have succeeded, in which case the alternative 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 fails.

If the chosen alternative succeeded, the actual-rule succeeds; if it failed, the actual-rule fails. This completes the execution of the actual-rule.

The sequence of alternatives must satisfy the following restrictions.

a) If the guard of an alternative cannot fail, the alternative must be the last one.

This restriction ensures that all alternatives can, in principle, be reached. Violating this restriction causes an error message.

b) If an alternative contains a member that cannot return, then this member must be the last one.

This restriction ensures that in an alternative all members can, in principle, be reached. Violating this restriction causes an error message.

c) If an alternative contains a member that has side-effects, then this member may not, in the same alternative, be followed by a member that can fail.

The third restriction ensures that side-effects of a member cannot materialize if the member fails; this in turn ensures that the tests necessary to determine the applicable alternative do not interfere with each other. Violating this restriction causes a warning. The programmer is urged to either reconsider the formulation of the problem, or make sure that the side effects will not have ill consequences.

3.1.3 Member

Members are units of action. Such an action is either a primitive operation, a rule call, or it is composed from 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, the part

```
declaration+type+idf
```

is an affix-form, `:declaration sequence option` is a last-member, as is `+`.

The notion last-member has been introduced into the syntax to ensure that a terminator (Section 3.5) will occur last in an alternative.

3.2 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.

3.2.1 Formal affix

Formal affixes define the number and the types of the affixes of a rule. Syntax:

formal-affix-sequence: formal-affix*, [anchor, formal-affix+].
formal-affix: plus-symbol, formal.
formal: formal-variable; formal-stack; formal-table; formal-file.
anchor: plus-symbol, anchor-symbol.

formal-variable: in-variable; out-variable; inout-variable.
in-variable: right-symbol, variable-tag.
out-variable: variable-tag, right-symbol.
inout-variable: right-symbol, variable-tag, right-symbol.
formal-table: [field-definition], table-tag, sub-bus.
formal-stack: sub-bus, [field-definition], stack-tag, sub-bus.
sub-bus: sub-symbol, bus-symbol.
formal-file: quote-image-symbol, file-tag.

Example of a formal-affix sequence:

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

Here `file` is the tag of a formal-file, `table` is the tag of a formal-table, finally `string` is the tag of an in-variable.

A formal-variable defines a *variable*. An in-variable or an inout-variable obtains an initial value from the calling rule; it is *initialized*. An out-variable has the attribute *uninitialized* at the beginning of each alternative in the actual-rule. The value of an out-variable or an inout-variable will be passed back to calling rule, so it must have attribute *initialized* attribute at the end of each alternative of the actual-rule where the control can return, namely which does not end in a jump, a failure-symbol, or an exit rule. Changes to an in-variable do not materialize outside the actual-rule. An in-variable behaves as if it were an initialized local-variable.

A formal-stack or a formal-table defines a stack or a table, respectively. If the field-definition (Section 4.2.2) is absent, the formal list is supposed to have one selector: the tag of this selector is the same as the tag of the formal list itself. For example, `[]list[]` has the same meaning as `[](list)list[]`.

A formal-file defines a file, which can be both character and data file.

The anchor, if present, marks the affix position from where the remaining affixes form the *repeat affix block*. The anchor is not matched against any actual-affix. In the example

```
+""file+@+table[]+>ptr
```

the repeat affix block has length two and is formed by the affixes `table` and `ptr`.

All affix tags in the formal-affix-sequence must be different. The cannot have qualifiers, and also must be different from the rule-tag that precedes this sequence.

3.2.2 Actual affix

Actual-affixes occur in rule calls specified in affix-forms, see Section 3.4. Each actual-affix corresponds to some formal-affix of the called rule. Syntax:

actual-affix-sequence: actual-affix∗, [anchor].
actual-affix: plus-symbol, actual.
actual: source; list-tag; file-tag; string-denotation.

Example of an actual-affix-sequence:

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

In this example `-511` is an integral-denotation, `/?/` is a character-denotation, `alpha` may be 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 meaning from the corresponding formal-affixes. The interrelation is discussed in Section 3.4.

3.2.3 Local affix

Local affixes are variables local for the actual-rule or the compound-member in which they are defined. Syntax:

local-affix-sequence: local-affix∗.
local-affix: minus-symbol, local-tag.

Example of a local-affix-sequence:

```
-count
```

None of the tags in the local-affix-sequence can have a qualifier, and all these tags must be different. A local-tag 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, meaning that the variable has been used.

3.3 Operations

Operations are the primitive actions in the ALEPH language. They are the assignments (transport), comparing two integer values (comparison), and adding new elements to a stack (extension). Syntax:

operation: transport; comparison; extension.

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

Example of a transport (assignment):

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

Example of a comparison:

```
ext*list[pnt] != -0x3b7
```

Example of an extension:

```
(* pnt->sel, nil->ect->ors *)list
```

3.3.1 Transport

A transport (assignment) is a function since it has no (inherent) side effects and always succeeds. Syntax:

transport: source, (arrow-symbol, destination)+.

The execution of a transport starts with the evaluation of its source. A source is evaluated as follows.

a) If the source is an integral-denotation, its value is the numerical value of digit sequence considered as a number in decimal notation, negated if preceded by a minus-symbol; or the hexadecimal value of the hex-digit sequence following the hex-symbol, also negated when preceded by a minus-symbol; or the numerical value of the manifest-constant as determined by the compiler.
b) If the source is a character-denotation, the value is the (numerical) code of the visible-character between the absolute-symbols `/`.
c) 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 *initialized* attribute.
d) If the source is a table-element or a stack-element, its value is determined as follows (see also Section 4.2.4 and Section 4.2.5). Let T be the table-tag or stack-tag in the source, and denote by L the (global or formal) list identified by T. First, a selector S is determined. If the of-symbol is present, S is the selector in front of it. If it is absent, then S is the same as T, this is the *standard selector* of the list. If S is not a selector of the list L, then it is an error.
   Next, an index P is determined. If the source between the sub-symbol and bus-symbol is present, it is evaluated, and the value is P. Otherwise P is set to the actual upper bound of L. As an example, `list` is equivalent to `list*list[>>list]`.
   Now consider the block B in L that has an address equal to P. If no such a block exists, then it is an error. The value of the list-element is the value identified by the selector S in the block B.
e) If the source is a limit, its value is described in Section 4.2.3.
f) If the source is a dummy-symbol, there is an error.

The value of the source is called V. Now the assignments of the transport are executed in textual order. A single assignment is executed as follows.

a) If the destination is a variable-tag, the value V is put in the location identified by the variable. If the variable is formal or local, then it has the *initialized* attribute in the rest of the alternative in which this transport appears.
b) If the destination is a stack-element, a selector S and a block B in the list L identified by the stack-tag is determined as above. The value V is put into the location of the block B identified by the selector S.
c) If the destination is a dummy-symbol, the assignment is a dummy action.

Examples:

`0->cnt->res`	now `cnt` and `res` are both zero
`list->list[q]->q`	the value at `list*list[>>list]` is put in the location identified by `list*list[q]` (so `list` must be a stack), 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

3.3.2 Comparison

A comparison is a question, i.e., it has no side effects and may either succeed or fail. Syntax:

comparison: source, relation, source.
relation:
lt-symbol; le-symbol; eq-symbol; ne-symbol; ge-symbol; gt-symbol.

Both sources are evaluated as described above. The two resulting values are considered to be signed integers and compared using the specified relation:

symbol	example	succeeds if
lt-symbol	`x < y`	`x` is less than `y`
le-symbol	`x <= y`	`x` is less than or equal to `y`
eq-symbol	`x = y`	`x` is equal to `y`
ne-symbol	`x != y`	`x` and `y` differ
ge-symbol	`x >= y`	`x` is greater than or equal to `y`
gt-symbol	`x > y`	`x` is greater than `y`

The return value of a comparison is "success" if the relation holds between the two values, and "fail" otherwise. Note that even if both source values represent pointers to lists, the numerical values of these pointers are compared, and not the values they point to.

3.3.3 Extension

An extension is an action, i.e., it has side effects and always succeeds. Syntax:

extension: left-of-symbol, field-transport-list, right-of-symbol, stack-tag.
field-transport-list: field-transport, (comma-symbol, field-transport)*.
field-transport: source, (arrow-symbol, selector-tag)+.

Call the stack indicated by the stack-tag S. All selector-tags appearing in the field-transport-list must be selectors of S.

First, the sources in the field transports are evaluated as described in Section 3.3.1 and their values are remembered. Subsequently a block B is created with as

many empty locations as the calibre of S. Next, all 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 the selector-tags.

No more than one value may be put in a given location in B, and at the end of the extension locations filled in B must form an end-section of B without holes. Violating this condition results in an error message. Finally, the stack S is extended by a (possibly incomplete) block containing the filled locations. In the case when the block is incomplete, a warning is issued.

Example: given a stack `st` declared as `[](sel,ect,or)st[]`, the first of the three extensions

```
(* 3->ect, 5->sel->or *)st,
(* 0->or *)st,
(* 5->sel->ect *)st
```

adds the block `(5,3,5)` to `st` and increases the actual upper bound `>>st` by 3. The second extension adds a single zero element to the top of `st` and increases `>>st` by 1. The last extension is incorrect as the mentioned selectors do not form an end-segment of the block.

3.4 Affix form

Calling (executing) a rule is done through an affix-form. It specifies both the rule to be called and the actual affixes (parameters) the rule is called with. Syntax:

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

Example:

```
string elem+tbl+str+cnt+symb
```

Let R be the rule identified by the rule-tag in the affix-form. Executing the affix-form means calling R so that its formal-affixes are specified by the actual-affixes of the affix-form. To establish the correspondence between the formal and actual affixes the actual-affix-sequence is modified as follows. Each string-denotation in the actual-affix-sequence is replaced by two affixes: a table-tag identifying an internal table, and a constant-tag pointing to an instance of the specified string in that table. Furthermore the trailing anchor in the actual-affix-sequence, if present, is deleted.

After this preparation the correspondence between actual and formal affixes is established in their textual order: the first actual corresponds to the first formal, the second actual to the second formal, and so on. In this process the formal anchor affix, if present, is skipped, and not matched against any actual affix. If the formal affixes are exhausted while the actual affixes are not, the formal affixes are reset to the affix immediately following the formal anchor. If there is no formal anchor, then this is an error. The matching process ends when there are no more actual affixes in the prepared list. At this point *either* the formal affixes must be exhausted as well, *or* both the last affix in the actual-affix-sequence, and the next unprocessed formal-affix of the called rule must be an anchor.

For an illustration of the process, assume that the formal-affix-sequence of the called rule is

```
+A[]+>B>+@+""C+D>+[]E[]
```

which has a *repeat affix block* of length 3, see Section 3.2.1. The affixes in the following actual-affix-sequences are correctly matched up with the formal affixes labelled by the corresponding upper case letters:

```
+a+b+@,
+a+b+c+d+e,
+a+b+c1+d1+e1+c2+d2+e2+c3+d3+e3+@.
```

In the first case the matching parts of both the actual-affix-sequence and the formal-affix-sequence end with an anchor; in the other two cases both sequences are exhausted simultaneously. In the last case the formal-affix-sequence is reset twice. The following affix sequences do not match correctly with the above formal-affix-sequence

```
+a+b, +a+b+c+d, +a+b+c+d+@, +a+b+c1+d1+e1+c2.
```

If the actual-affix-sequence ends with an anchor, then the actual-rule, of which this affix-form is a member, must also have an anchor in its formal-affix-sequence (Section 3.2.1), and also there must be an anchor in the formal-affix-sequence of the rule R. These two *repeat affix blocks* must have the same number of elements, and are also matched against each other in their textual order.

The matched actual and formal affixes must satisfy the following conditions.

a) 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 during the execution of the rule R are executed directly on the actual. If the formal has field-definition, then the block size (calibre) and the standard selector of the formal and that of the actual must be equal, while the selectors other than the standard selector, may differ. If the formal has no field-definition, then 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.

b) The actual corresponding to a formal-stack must be a stack-tag identifying a (global or formal) stack. All actions performed on the formal during the execution of R are executed directly on the actual. Restriction on calibre and selectors is the same as for formal-tables.

c) 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.

d) The actual corresponding to a formal in-variable must be a source; the actual corresponding to a formal inout-variable or a formal out-variable must be a destination. The dummy-symbol, however, is not allowed for formal inout-variables. The destination for an inout-variable must have the *initialized* attribute.

e) If the matching actual and formal affixes are both from the corresponding *repeat affix blocks*, then the actual corresponding to a formal inout-variable must also be a (formal) inout-variable, and the actual corresponding to an out-variable must be either an out-variable or an inout-variable.

After establishing the correspondence between the actual and formal affixes, the following steps are executed.

Step 1.

Prepare the list of affixes to be passed to the called rule as follows.

a) If the last affix in the actual-affix-sequence is not an anchor, then this list consists of the (actual) affixes in their original order as prepared above.

b) If the actual-affix-sequence ends with an anchor, then first take all affixes as prepared above. By the conditions the actual-rule of which this affix-form is a member, has a *repeat affix block*. The actual (visible) affixes in that block, plus all affixes in subsequent (pending) blocks are copied to the end of the list.

Step 2.

The copying part of the affix mechanism is put into operation. For each actual-affix in the list prepared in Step 1, if the corresponding formal-affix is either an in-variable, or an inout-variable, a transport (see Section 3.3.1) is executed with the actual as the source, and the formal as its only destination.

Step 3.

Prepare repeat affix blocks. If the actual-rule rule R (as identified by the rule-tag in the affix-form) has a *repeat affix block* of length S, then actual-affixes corresponding to the formal affixes in that affix block are grouped into blocks of length S starting from the leftmost actual-affix. Elements of the first block are matched against the formals in the formal *repeat affix block*, while the remaining blocks, if any, are the "pending" or "unseen" blocks. If R has no repeat affix block, this step is skipped.

Step 4.

Subsequently the rule R is executed. If the actual-rule R succeeds, the affix-form succeeds, if it fails, the affix-form fails.

If the affix-form succeeds, the restoring part of the affix mechanism is executed: for each formal-affix that is either out-variable or inout-variable a transport is executed with the formal as its source and the actual as its destination, in the order in which these affixes appear. This copying is also executed for affixes in the repeat affix block.

Example 1: Suppose the following rule and variable declarations:

```
'function'sq+>a+>b>+c>: b->c, a->b.
'variable'x=0,y=1.
```

The actual-rule `sq+x+y+x` swaps the values of `x` and `y`. Indeed, the copying part described in Step 2 executes the transports `x->a`, `y->b` *before* calling `sq`, and the transports `b->y`, `c->x` *after* `sq` successfully returns.

Example 2: With the following declarations:

```
'action'P+""file+@+T[]+>ptr: $ do something ...$.
'action'PP+@+T[]+>ptr: P+STDOUT+"first"+@.
```

the actual-rule `PP+"str1"+"str2"` will call `PP` with four affixes; the first and the third are the internal table containing inline strings, the second and fourth affixes are pointers pointing to the (packed forms) of the strings `"str1"` and `"str2"`. The rule `PP`, in turn, calls `P` with a total of seven affixes of which the last four comes from the repeat affix blocks of `PP` (two in the visible block and two pending). The overall effect is the same as that of the actual-rule

```
P+STDOUT+"first"+"str1"+"str2".
```

3.5 Terminators

Syntax:

terminator: jump; success-symbol; failure-symbol.
jump: repeat-symbol, rule-tag.

3.5.1 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` should be executed. If `,c` is removed, the rule becomes correct. Likewise, the rule

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

is incorrect: after an unsuccessful return from `:bad2`, the affix-form `c` should be executed. If the parentheses are removed, the rule becomes 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 Section 3.6) the actual rule until C turns into a rule. Then the above applies.

3.5.2 Success and failure symbols

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

3.6 Compound members

Compound-members serve to turn a composite rule-body into a single member. Syntax:

compound-member: open-symbol, [local-part, colon-symbol],
rule-body, close-symbol.
local-part: rule-tag, local-affix*; local-affix+.

Example:

```
(order-n: 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 no formal affixes, and its local affixes are specified in the local-part. The call then calls that rule. 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) The *type* of the rule-body is determined, and the corresponding typer (see Section 3.1.1) is placed in front of the rule-tag.

Example: For the compound-member

```
(1->n, m=n; incr+m, m=n)
```

the created rule-declaration is

```
'question'zzgrzl: 1->n, m=n; incr+m, m=n.
```

Since the rule created from a compound-member has no formal affixes, the copying mechanism does not prevent the side effects from materializing. If `n` is a local or formal tag, then if the guard of the first alternative in the rule body

```
(1->n,m=n);
n=0, do something;
```

fails (because the value of `m` is not one)), the side effect of changing `n` remains in effect and the guard of the second alternative will never succeed.

Neither the rule-tag nor any of the local-tags in the local-part can have qualifiers.

3.7 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 value (as opposed to the successive execution of guards). Classifications allow a faster selection of alternatives at the cost of a less versatile selection mechanism. Syntax:

classification: classifier-box, class-chain.
classifier-box: box-symbol, source, box-symbol.
class-chain: (class, semicolon symbol)+, last-class.
class: area, comma-symbol, alternative.
last-class: class; alternative.
area: sub-symbol, zone, (semicolon-symbol, zone)*, bus-symbol.
zone: [constant-value], up-to-symbol, [constant-value];
constant-value; list-tag.

A list-tag in a zone must identify a global (and not local or eternal) stack or table; moreover the constant-value cannot be an actual upper or lower limit, nor an external constant-tag (as these values are not known during compilation time.

Example 1:

```
(n:get+char,
   (=char=
    [/0/:/9/],        dgt->type;
    [/a/:/z/;/A/:/Z/],ltr->type;
    [/+/;/-/;/*/;///],op->type;
    [0:31;127],       :n;
    [:],              err->type))
```

Example 2:

```
=tag=
[var   decl], handle variable+tag;
[macro decl], handle macro call+tag;
[proc  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. Let the resulting value be V. The areas in the classification are searched in textual order for the first area which "contains" V. If such an area is found, the alternative following that area applies and is executed (see Section 3.1.2). If there is no such an area, the last-class must be an alternative, which then applies and is executed. Otherwise the program run is aborted with an error message.

A given area contains the value V if one of its zones contains V. Whether a given zone contains V is determined as follows.

a) If the zone is a constant-value, then it contains V if V is numerically equal to the constant-value.
b) If the zone contains an up-to-symbol, it is designated by two boundaries. The left boundary L is the constant-value before the up-to-symbol or, if it is missing, the smallest (negative) representable integer. The right boundary R is the constant-value after the up-to-symbol or, if it is missing, the maximal representable integer. The zone contains V if $L \leq V \leq R$.
c) If the zone is a list-tag, this list-tag must identify a global (and not a formal) list. The zone contains V if V is in the virtual address space (see Section 4.2.1) assigned to that list.

Areas may coincide partially, but it is an error if an area cannot be reached by some value of the source.

A classification can fail only if one of its alternatives fail; and has side-effects if at least one of the alternatives has side-effects.

4 Data

The basic way of representing information in ALEPH is through integers. There are four integer-based data types:

a) integers (constants),
b) variables that contain integers,
c) ordered lists of integers (tables), and
d) ordered lists of locations that contain integers (stacks).

Integers used in data declarations can be given in the form of expressions.

The basic way of routing information into and out of the program is through files. There are two file types:

a) charfiles, files containing only integers that correspond to characters, and
b) datafiles, files containing integers and pointers to prescribed stacks and tables.

There are three primitive actions on integer-based data: transport (assignment), comparison, and extension. Additional integer handling (such as addition and multiplication) are done through external rules. There are no primitive actions on files, all file handling is done through externals. Syntax:

data-declaration:
    constant-declaration; variable-declaration; static-variable-declaration;
    table-declaration; stack-declaration; static-stack-declaration;
    table-filling; stack-filling;
    file-declaration.

data-prototype:
    constant-prototype; variable-prototype; static-variable-prototype;
    table-prototype; stack-prototype; static-stack-prototype;
    file-prototype.

4.1 Constants and variables

Constant and variable tags are initialized through expressions. Expressions are evaluated during compilation.

4.1.1 Expressions

Syntax:

expression: [unary-operator], base, (binary-operator, base)*.
base: constant-value; open-symbol, expression, close-symbol.
unary-operator: minus-symbol; complement-symbol.
binary-opeator: times-symbol; by-symbol; plus-symbol; minus-symbol; or-symbol; and-symbol; xor-symbol.

Examples:

```
-3 + 0x7f & byte size
((/e/+1)*char size + /n/+1)*char size+/d/+1
/////// $ the code of / divided by itself
```

The value of an expression is the integral value that results from evaluating the expression according to the standard rules of arithmetic. Operators applied in the order of their priorities; operators of equal priority are applied from left to right. Unary operators have the highest priority. They are followed by multiplication and division, then addition and subtraction. The Boolean (bitwise) operators have the lowest priority.

An expression through the constant-value (Section 3.3) cannot contain actual upper or lower limits, and cannot contain external constant-tags. There is no restriction, however, on the usage of constant-tags exported by other modules.

4.1.2 Constants

A constant-declaration specifies the integral value which is represented by a constant-tag. This relationship is fixed and does not change. Syntax:

constant-declaration: constant-symbol, constant-description, (comma-symbol, constant-description)*, point-symbol.
constant-description: constant-tag, equals-symbol, expression.
constant-prototype: constant-symbol, constant-tag, (comma-symbol, constant-tag)*, point-symbol.

Example for a constant-declaration:

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

The value of the expression cannot depend on the constant-tag being declared. Consequently, the declaration

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

results in an error message. The constant-tag in a constant-prototype should originate either from a constant-declaration or from a pointer-initialization, see Section 4.2.6.

4.1.3 Variables

An ALEPH variable consists of a variable-tag and a location; the location may or may not contain a value. If it contains a value, then the variable *has* that value. The content of the location may change. Once a location has obtained a value it can't become empty again.

A global variable is declared in a variable-declaration; a formal variable originates from a formal-affix-sequence, and a local variable originates from a local-affix-sequence.

variable-declaration: variable-symbol, variable-description,
    (comma-symbol, variable-description)*, point-symbol.
variable-description: variable-tag, equals-symbol, expression.
static-variable-declaration: static-symbol, variable-declaration.

variable-prototype: variable-symbol, variable-tag,
    (comma-symbol, variable-tag)*, point-symbol.
static-variable-prototype: static symbol, variable-prototype.

Examples:

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

For each variable-description a location is created with the variable-tag as its label, and filled with the value of the expression. Variables can be used in sources and destinations. They cannot be used in expressions.

A static variable-tag behaves as a variable in the module it is declared, but in other modules it behaves like a constant-tag. The content of a variable-tag can only be manipulated by rules in its defining module; in other modules such a variable is "read only".

4.2 Tables and stacks

4.2.1 The address space

ALEPH stacks and tables together are called *lists*. Items in lists are identified by unique addresses; these addresses come from a range of integer values called *virtual memory space*. The range starts at some small positive number and ends at a large positive number. Lists occupy separate consecutive locations in this virtual memory. List elements are identified (indexed) by their virtual memory addresses. Variables in ALEPH have no virtual addresses as they are not stored in the virtual memory space.

When an ALEPH program is compiled, the complete available virtual memory space is distributed among the lists in the program with almost no control of the programmer. The chunk of memory addresses assigned to a list is its *virtual address space*. Frequently only a part of this virtual space is in use, which is called the *actual address space* of the list.

The virtual memory is distributed among all lists as follows.

a) For each table and for each stack with empty size-estimate in its declaration, the size of the actual address space is determined by its fillings. The allocated virtual space is the amount necessary to store elements of those fillings.
b) For a stack with an absolute-size the allocated virtual space is the maximum of the given value and the total size of the fillings.
c) The remainder of the virtual address space is distributed over the rest of the stacks proportionally to the specified relative-size.
d) External tables and stacks set up their own virtual limits from a separate virtual memory range.

For each list the largest address in its virtual address space is its *virtual upper limit*, while the lowest address plus the calibre (block size) minus one is its *virtual lower limit*. These values are determined at compile time and do not change.

The initial values of the *actual limits* of the list are calculated as follows. The *actual lower limit* is set to the *virtual lower limit*. The initial value of the *actual upper limit* is calculated from the fillings of the list. These fillings are positioned next to each other at the beginning of the virtual address space. The *actual upper limit* is set to the largest (rightmost) occupied virtual address. If the list has no filling, then its *actual upper limit* is set to its virtual lower limit minus the calibre. This value is one less than the lowest virtual address assigned to the list, thus does not point into the list's address space.

The actual limits of a table are equal to its virtual limits and do not change. Actual limits of a stack can change, but they always remain within its virtual limits.

Suppose, for example, that the following list declarations (see Section 4.2.4 and Section 4.2.5) occur in the program:

```
'table' pw[]=(1,10,100,1000).
'stack' [= 5 =] dg []=(0),
        [ 30  ] st [],
        [ 50  ] (num,denom) ax[] =
                ((365,113):pi, (191,71):e).
```

Assuming that the virtual memory is indexed from 10000 to 99999, the virtual address space could have the following layout:

address	contents	belongs to	selector	pointer
10000	1	pw	pw	<pw, <<pw
10001	10	pw	pw	
10002	100	pw	pw	
10003	1000	pw	pw	>pw, >>pw
10004-10009	--	unassigned		
10010	0	dg	dg	<dg, <<dg, >>dg
10011-10013	--	dg		
10014	--	dg		>dg
10015-10018	--	unassigned		
10019	--	unassigned		>>st
10020	--	st		<st, <<st
10020-32327	--	st		
32328	--	st		>st
32329	--	unassigned		
32330	355	ax	num	
32331	113	ax	denom	<ax, <<ax, pi
32332	191	ax	num	
32333	71	ax	denom	>>ax, e
32334-99998	--	ax		
99999	--	ax		>ax

A stack can be extended to the right while raising its actual upper limit through an extension (Section 3.3.3). Items can be removed from the right of a stack through a call of a run in the Standard Library (see Section 8.2) such as `unstack` or `unstack to`, after which the discarded address space can be reclaimed again, but not the values in it.

4.2.2 Selectors

Elements of an ALEPH table or stack are subdivided into consecutive blocks. The address of the rightmost item in a block is the block address, and values in the block are referenced through the block address and a selector-tag. The selector-tag identical to the list-tag is called the *standard selector*. The available selectors, which might miss the standard selector, are specified in a field-definition.

Field-definitions can be used in a formal-affix-sequence to specify the block structure of a formal table or stack; in list declarations and prototypes; and in fillings. If no field-definition is specified in a declaration, then the blocks in the list have a single element, and the selector of that element is the standard selector. Syntax:

field-definition: field-list-pack+.
field-list-pack:
    open-symbol, field, (comma-symbol, field)*, close-symbol.
field: dummy-symbol; selector-tag, (equals-symbol, selector-tag)*.

Examples for field-definitions:

```
(s1,s2=t1,s3=t2)
(s1,s2,s3)(#,t1,t2)
```

A field-definition contains one or more field-list-packs. Each of them defines the calibre of the list as the number of the fields in it, this number must be the same for each field-list-pack in the field-definition.

Selectors of the block elements are specified from left to right by the selector-tags in the field-list-pack. The same block element can be identified by multiple selectors; the additional selectors are specified after the initial selector using an equals-symbol, see the first example. The second example specifies the same block structure with the same selectors, emphasizing which selectors are used together. The dummy-symbol # is used as a placeholder.

If a list is defined with a field-definition, then a matching prototype must also contain a field-definition, while a list-filling may omit it. All field-definitions of the same list must define the same calibre (block size), and the same standard selector, but could define different selector names which accumulate: any of the specified selectors can be used to identify elements in the list. Nevertheless, all field-definitions must be consistent on the standard selector.

4.2.3 Limits

Limits provide the block size, the virtual and actual limits of a list as constant values. The actual limits of a stack can change by an extension (Section 3.3.3), or by stack-manipulating externals (Section 8.2). Syntax:

limit: limit-token, list-tag.
limit-token: static-limit; dynamic-limit.
static-limit: vlwb-symbol vupb-symbol; calibre-symbol.
dynamic-limit: alwb-symbol; aupb-symbol.

Examples:

```
>table, <>blocksize, <<stack.
```

The calibre is the block size of the list. Static limits are determined during compilation; actual limits of a stack may change during the program run as a consequence of actions that change the stack size. Actual limits of a table do not change.

If the list-tag in the limit is a formal affix, the returned value is the limit of the corresponding global list. In particular, the calibre of a formal-affix is the calibre of the corresponding global list, and not the calibre of the local affix.

4.2.4 Tables

Tables originate from table declarations. They contain values that are determined at compile time and cannot change. Syntax:

table-declaration: table-symbol, table-description,
    (comma-symbol, table-description)∗, point-symbol.
table-description: table-head, sub-bus, [equals-symbol, filling].
table-head: [field-definition], table-tag.

table-prototype: table-symbol, table-head,
(comma-symbol, table-head)*, point-symbol.

Examples:

```
'table'(s1,s2,s3)(#,t2,t3) T[]. $ declaration
'table'(up,down=S) S.  $ prototype
```

The calibre of the table `T` is three. The first element of a block of `T` can be accessed by the selector `s1` only; the last element by both selectors `s3` and `t3`. The table `T` has no standard selector, consequently using the list-element `T[ptr]` results in an error. The table `S` has calibre two, and the standard selector identifies the last element of its block.

A table-declaration declares a table, specifies the block size (calibre), selectors, and whether it has a standard selector or not. The content of the table is specified by fillings, one of which can be optionally added at the declaration. The final order in which the different fillings are stored in the table is unspecified.

4.2.5 Stacks

Stacks originate from stacks-declarations. Syntax:

stack-declaration: stack-symbol, stack-description,
(comma symbol, stack-description)*, point-symbol.

static-stack-declaration:
static-symbol, stack-declaration.

stack-description:
sub-symbol, size-estimate, bus-symbol, stack-head, sub-bus,
[equals-symbol, filling].

stack-head: [field-definition], stack-tag.

size-estimate:
empty; absolute-size; relative-size.

absolute-size: box-symbol, constant-value, box-symbol.

relative-size: constant-value.

stack-prototype: [static-symbol], stack-symbol, stack-head,
(comma-symbol, stack-head)*, point-symbol.

Examples:

```
'stack'[= line width =](char)print line[]. $ declaration
'stack'BUFFER, (flag,data) RULE.  $ prototype
```

The stack `print line` has a fixed number of elements specified by the constant-tag `line width`, it has one-element blocks, and no standard selector. This stack cannot grow beyond its specified limit, however it can still shrink. In the prototype two stacks are specified. For `BUFFER` no field-definition is given, the same prototype can be written equivalently as `(BUFFER)BUFFER`.

A stack-declaration declares a stack, specifying the required virtual memory. If the size-estimate is empty, the allocated virtual memory will be determined

by the total size of fillings. The absolute-size specifies the number of requested virtual locations (not blocks); this number must be positive. The relative-size must be positive and at most 100. The constant-value in the size-estimate cannot depend on values which are determined after the distribution of the virtual memory, such as virtual limits or pointer constants.

A static stack-tag behaves as a stack in the module it is declared, but in other modules it behaves like a table-tag: neither the list itself, nor its content can be modified.

4.2.6 Filling

The initial content of lists is defined in fillings. Fillings of the same list can spread across the program, even across different modules. Each filling is stored in the designated list at consecutive locations, but the order in which the different fillings appear is unspecified. A filling may contain pointer-initializations which define a constant-tag with the value of the virtual address of the block it follows.

```
table-filling:
    table-symbol, table-head, equals-symbol, filling.
stack-filling:
    stack-symbol, stack-head, equals-symbol, filling.
filling: open-symbol, filling-unit,
    (comma-symbol, filling-unit)*, close-symbol.
filling-unit:
    constant-value, [multiplier], pointer-initialization*;
    open-symbol, compound-block, close-symbol,
        [multiplier], pointer-initialization*;
    string-denotation, pointer-initialization*.
compound-block: value-block; selector-block.
multiplier: times-symbol, constant-value.
pointer-initialization: colon-symbol, constant-tag.
value-block:
    (constant-value, comma-symbol)*, constant-value,
        [times-symbol], (comma-symbol, constant-value)*.
selector-block:
    fill-transport, (comma-symbol, fill-transport)*.
fill-transport: constant-value, (arrow-symbol, selector-tag)+.
```

Examples:

```
'stack'(ch,p)optor=
  ( (/+/->ch,3->p),(3->p,/-/->ch),(5->p,/^/->ch) ).
'table'messages=("tag undefined":bad tag).
```

Values in the filling specify the content of consecutive locations of the designated list. Each filling-unit defines one or more blocks of the list which are added to

the list in the order they appear. Each filling-unit can be followed by pointer-initializations. If present, all constant-tags in it are defined to have the virtual address of the block.

If the filling-unit is a constant-value, then it defines a single element block with the specified content. If there is a multiplier, then the block is repeated by the specified value, which must be at least one.

If the filling-unit is compound-block, then it specifies a complete block, the number of elements must be equal to the calibre of the list. In a value-block the block elements are specified in a left to right order. One of the elements can be followed by the times-symbol * indicating that this element should be repeated, if necessary, to get the required number of block elements. Example: the filling

```
'stack'(a,b,c,d,e,f,g,h)big block=( (1,0*,1)*100 ).
```

adds 100 blocks to the stack `big block`, each consisting of a `1`, six `0`, and another `1`. The block can also be written as `(1,0*6,1)`.

If the compound-block is a selector-block, then the constant-values are stored at the location(s) specified by the selector-tag(s). One of the selectors can be replaced by the times-symbol * to mean that the value should be copied to all selectors not mentioned explicitly in the selector-block.

If a value-block specifies less elements or more elements than the calibre, a warning is given. If a selector-block specifies less elements than the calibre, then those elements must form an end-segment of the block, and a warning is given.

If the filling-unit is a string-denotation, the string is stored, in compressed form, at several consecutive locations, forming a *string block*. As other blocks, the string block is also identified by the address of its last (rightmost) element. String blocks can be manipulated by Standard Library rules, see Section 8.3.

4.3 Files

ALEPH files originate from file declarations. They are associated with file-system elements through an explicit or implicit call to the Standard Library rule `open file`, and detached by the rule `close file` (see Section 8.4). Files can be opened for *reading*, when the content of the associated file is made available for the ALEPH program, or for *writing*, when the data written to the file is stored in the associated computer file. A file cannot be opened for reading and writing simultaneously. Syntax:

file-declaration: file-typer, file-description,
    (comma-symbol, file-description)*, point symbol.
file-typer: charfile-symbol; datafile-symbol.
file-description: file-tag, [file-area], equals-symbol,
    [right-symbol], string-denotation, [right-symbol].
file-area:
    sub-symbol, list-tag, (semicolon-symbol, list-tag)*, bus-symbol.
file-prototype:
    file-typer, file-tag, (comma symbol, file-tag)*, point-symbol.

Examples:

```
'charfile'OUTPUT=>"result.txt", INPUT="data/survey.txt">.
'datafile'TEMP[tag;link]= >"/tmp/xbcrq.bin">.
```

A file-description declares a file of the type indicated by the file-typer. The file-area can be used in datafile declarations only.

If an ALEPH file is not opened explicitly, then the first file operation using the file-tag tries to associate it with the file-system element specified in the string-denotation. The optional right-symbols before and after the string-denotation restrict this implicit opening: the file is opened for *writing* only if there is a right-symbol before the string (indicating the data is allowed to flow into the specified string), and opened for *reading* only if there is a right-symbol after the string (indicating that data can flow from the string). When the file is opened explicitly by a call of `open file`, this restriction does not apply.

ALEPH contains no explicit file handling statements: all file handling is done through rules in the Standard Library, see Section 8.4. Files are read and written sequentially. When a file is opened for reading, it can be positioned to a previously visited location; in particular it can be reset and reread from the beginning. Files opened for writing cannot be positioned. Writing to a file ends when the file is closed, or when the program terminates.

4.3.1 Character files

An ALEPH character file is a sequence of unicode characters stored using UTF-8 encoding. A charfile returns and accepts unicode characters, converting these characters to and from UTF-8 encoding on the fly. Standard Library rules allow two ways of processing a character file.

Characterwise: when a character file `file` is opened for reading, each call of the external predicate `get char+""file+ch>` yields the next unicode character in `ch`, failing when the end of file is reached. When opened for writing, the action `put char+""file+>ch` adds the unicode character `ch` to the file written, ignoring the request if `ch` is illegal or zero.

Linewise: the predicate `get line+""file+[]stack[]+ch>` extends the `stack` with the character codes of a complete line in the following way. If there are no more characters in the file, then the rule fails, and the `stack` is unchanged. If the next character in the `file` is a `newline`, then it is skipped, and `ch` is set to the code of the newline character; if the next character is not a `newline`, then `ch` is set to the value `rest line`. Subsequently unicode characters are read from `file` until either the end of file, or until the next character would be a `newline`. Codes of the characters read are put on the top of `stack` in the order they are read, and then `get line` returns with success.

The action `put line+""file+table[]+>ch` assumes that all elements in `table` are codes of unicode characters, appends all of these characters to `file`, and also a character with code `ch`, except when `ch` is `rest line`, which is ignored.

Character files `STDIN` and `STDOUT` defined in the Standard Library read from the standard input and write to the standard output, respectively; typically it means reading from and writing to the console.

4.3.2 Data files

An ALEPH datafile consists of a list of *data items*. A data item is an integer value and an indication about its meaning. This indication is either `numerical`, in which case the integer value stands for itself, or it is a serial number identifying an ALEPH list, in which case the integer value is an offset from the left end of that list.

A data item is written to a datafile by a call of the action `put data+""file+>item+>type`. The data item is constructed from the `item` and `type` affixes and from the file-area in the file-description of the `file` in the following way.

If the affix `type` equals `numerical`, then the data item consists of the value of the `item` affix and indication `numerical`.

If the affix `type` equals `pointer`, then the value of `item` must either be zero (indicating the null pointer), or be an address in the virtual address space of one of the lists whose list-tag is enlisted in the file-area. The data item then consists of the offset from the left end of that list and the serial number of the list in the file-area.

Data items are read from a datafile by calls of the predicate `get data+""file+item>+type>`. If there are no more data items on the datafile, the predicate fails. If there is still a data item on `file`, it is read and the `item` and `type` are reconstructed, using the lists in their order of appearance in the corresponding file-area.

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

5 Externals

External items can be used in the same way as internally declared ones. An external rule differs from an internal rule in that its body is not given by the ALEPH construct, but is instead obtained from some external source. In the same way values of other external items are obtained from external sources. The necessary information for correct compilation is supplied by the external-declarations. Since the correct usage requires special knowledge about the low level details of the compiled code, external-declarations are restricted to *library mode* only. Many items in the Standard Library are realized by externals. Syntax:

external-declaration:
    external-rule-declaration; external-data-declaration.
external-rule-declaration:
    external-symbol, typer, external-rule-description,
        (comma-symbol, external-rule-description)*, point-symbol.
external-rule-description:
    rule-head, equals-symbol, string-denotation.

Example:

```
'external''function'add+>x+>y+z>   = "#3=#1+#2",
                    subtr+>x+>y+z> = "#3=#1-#2".
'external''exit'exit+>code = "_exit(#1)".
```

An external-rule-description defines a rule of the type specified by typer and known to the ALEPH program by the name rule-tag. The string-denotation describes, in an implementation dependent way, how calls of the rule-tag are handled. It is the responsibility of the supplier of the string-denotation to see that it translates to actions which are in accordance with the type of the rule and that no side effects will occur when a call of the rule fails.

An ALEPH program can access and manipulate system data using external-data-declarations. For example, the external table `STDARG` is pre-filled with the command line arguments specified when the ALEPH program is launched. Syntax:

external-data-declaration:
    external-symbol, data-typer, external-data-description,
        (comma-symbol, external-data-description)*, point-symbol.
data-typer: constant-symbol; variable-symbol;
    table-symbol; stack-symbol; file-symbol.
external-data-description:
    data-head, equals-symbol, string-denotation.
data-head:
    [field-definition], list-tag;
    constant-tag;
    variable-tag;
    file-tag.

Examples:

```
'external''table'(head,tail)graph="initGraph".
'external''constant'sys time="return time()".
```

The string-denotation in the external-data-description specifies, in an implementation dependent way, how the declared item is accessed. In the external table declaration above, `initGraph` is the name of an external procedure which is responsible to prefill the table with content and set its virtual and actual limits. Using the external constant `sys time` as a constant-source either in a transport or as an actual-affix would supply the actual system time.

6 Pragmats

Pragmats control many aspects of the compilation. The exact position of a pragmat in the program may be significant. Syntax:

pragmat: pragmat-symbol, pragmat-item,
    (comma-symbol, pragmat-item)∗, point-symbol.
pragmat item:
    pragmat-tag, equals-symbol, pragmat-value;
    pragmat-tag, equals-symbol, pragmat-value-list.
pragmat-value-list: open-symbol, pragmat-value,
    (comma-symbol, pragmat-value)∗, close-symbol.
pragmat-value:
    tag;
    integral-denotation;
    string-denotation.

Example:

```
'pragmat'module=basic,
    require=("mod1","mod2),
    macro=(show integers, is value, set all bits).
```

The tag in a pragmat must be an identifier, it cannot have a qualifier. Similarly, the string-denotation must be a proper-string and not a manifest-string,

Before a pragmat is evaluated, it is preprocessed: the pragmat-value-list is removed in the following way.

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

The pragmat-item `require=("mod1","mod2")` thus has the same meaning as `require="mod1", require="mod2"`.

6.1 Compiler output

Pragmats in this group control the output of the ALEPH compiler.

`'pragmat'tab width=`integer.
: The right hand side of the `tab width` pragmat must be a positive integer less than 100. The value will be used as the tab size for program text printing. The default value is `8`.

`'pragmat'right margin=`integer.
: Sets the right margin for program text and dictionary printing. The default value is `120`.

`'pragmat'list=on/off.`
: The tag on the right hand side of the `list` pragmat can be `on` or `off`. The program text source is printed when this value is `on`. The default is `off`.

'pragmat'dictionary=on/off.
: The tag in the pragmat can be on or off. When it is on, each tag with the corresponding source line number is stored in a dictionary. The dictionary, if not empty, is printed when the compilation finishes. The default is off.

'pragmat'warning level=integer.
: Set the warning level between 0 and 9, 0 being the most verbose. The default value is 4.

'pragmat'error=proper-string.
: Issues a compilation error with the given message.

'pragmat'warning=proper-string.
: Issues a compilation warning at level 9 with the given message.

6.2 Compilation pragmats

Pragmats in this group control aspects of code generation.

'pragmat'bounds=on/off.
: Turn index checking on or off. If it is on, list indices are checked to point into the actual address space of the list. When it is off, no checking is performed. The default value is off. Index checking cannot be turned on or off inside an actual-rule.

'pragmat'count=on/off.
: Turn profiling on or off. If it is on when compiling an actual-rule, a counter is added which counts the number of calls to this rule. These call numbers are printed out after the run of the compiled program terminates. The default value is off.

'pragmat'trace=on/off.
: Turn tracing on or off. If it is on when compiling an actual-rule, code is added which prints out the rule name and the value of its in- and inout-affixes whenever the rule is called. The default value is off.

'pragmat'macro=rule-tag.
: In the compilation unit of this pragmat, calls of the actual-rule corresponding to the specified rule-tag are implemented by textual replacement. The replacement, however, can result in a syntactically incorrect program text or in a different semantics, see ection 7.5. The rule-tag cannot have a qualifier.

'pragmat'library mode=on/off.
: Turn library mode on or off. The value determines whether library extensions are allowed or not. The default of off.
: If library mode is on, the @ character is considered to be a letter, thus private tags can be created. Directory listing ignores tags starting with @. More importantly, external-declarations and the front matter and back matter pragmats are allowed in library mode only.

'pragmat'front matter=proper-string.
: The string, without quote marks, is copied to the front of the generated code. In the string the character pair %n is replaced by the newline character. This

pragmat is accepted only when `library mode` is `on`. The order in which strings from different `front matter` pragmats appear is unspecified.

`'pragmat'back matter=`proper-string.
This pragmat is handled similarly to `front matter`, only the supplied string is copied to the end of the generated code.

6.3 Conditional pragmats

Conditional pragmats can be used to instruct the compiler to ignore certain parts of the source file. They have the syntax

```
    'pragmat'if=tag.       'pragmat'else=tag.     'pragmat'endif=tag.
or
    'pragmat'ifnot=tag.    'pragmat'else=tag.     'pragmat'endif=tag.
or
    'pragmat'ifdef=tag.    'pragmat'else=tag.     'pragmat'endif=tag.
or
    'pragmat'ifndef=tag.   'pragmat'else=tag.     'pragmat'endif=tag.
```

The tag in `if` and `ifnot` pragmats can be one of `compile`, `module`, `library mode`, `list`, and `dictionary`. The program text between the `if` and `else` pragmats is processed if tag is (or is not) in effect, otherwise it is skipped; and the opposite is true for the text between `else` and `endif`. The `else` part may be missing. The tag in `ifdef` and `ifndef` pragmats can be any identifier, and the compiler checks if this tag has (has not) been defined at this point by a declaration, an import prototype, or by a `define` pragmat.

`'pragmat'define=`tag.
Mark the specified tag as "defined" for use in an `ifdef` pragmat.

Example for a conditional pragmat:

```
  'pragmat'if=module,include="private",else=module,
                    include="public",endif=module.
```

It adds the source file `private` among those to be processed if a `module` pragmat has been processed previously, otherwise it adds the `public` source file.

The `if...endif` pragmats must be nested properly, and the skipped text must be syntactically correct (as it is scanned to find the closing pragmat). The `'end'` symbol marking the end of the source file is never ignored: conditional pragmats do not extend over the end of the current file.

6.4 Module pragmats

`'pragmat'compile=on/off.`
The value of this pragmat is set automatically to facilitate module handling, see Chapter 7. It cannot be manipulated directly in the program text.

`'pragmat'title=`proper-string.

This pragmat is ignored if the pragmat `compile` is `off`, otherwise it sets the title of the unit under compilation. The default title, if none is specified, is `"aleph"`. The title string is not used explicitly, but it can be included in the program text by the manifest-string `_title_`, and it also appears in the intermediate `ALICE` code, as well as in the linked final code.

`'pragmat'std library=on/off.`
Enable or disable Standard Library source files. The default value is `on`. This pragmat is accepted only when `compile` is `on`.

`'pragmat'module=`tag.
If `compile` is `on`, the current compilation unit is defined to be a module and the value of the `prototype` pragmat is set to `public`. The tag in the pragmat defines both the module name and its namespace. After the pragmat has been processed, the condition `if=module` holds in subsequent conditional pragmats. If `compile` is `off`, the `module` pragmat specifies the name and namespace of the invoked module, and also sets the `prototype` pragmat to `import`.

`'pragmat'require=`proper-string.
The string specifies a filename, and the ALEPH source in that file must be a *module*. The module is added to the *module hierarchy* of the unit under compilation, and the filename is added, if it is not there, to the pool of the *required* source files to be processed.

`'pragmat'library=`proper-string.
Add the filename specified by the string, if it is not there, to the pool of ALEPH source files to be processed as a *user library*. This pragmat is accepted only if `compile` is `on`.

`'pragmat'include=`proper-string.
Add the filename specified by the string to the pool of ALEPH source files to be processed as an addendum to the current source file.

`'pragmat'prototype=import/public/none/reverse.`
This pragmat has three possible values: `import`, `public`, and `none`; the default value is `none`. The pragmat determines how the compiler handles prototypes. The case `prototype=import` indicates that the tag in a prototype has a declaration outside this unit (and then it cannot occur in a declaration in this unit). In the case `prototype=public` a tag appearing in a prototype automatically gets the *public* flag, and must be declared in this unit (in particular, it cannot be imported, and this unit must be a module). When `prototype=none`, prototypes are used for type checking only, and do not imply any specific behavior. Finally, `prototype=reverse` swaps the current prototype value between `import` and `public`, while keeping `none` unchanged.

7 Units

7.1 Source files

Translating an ALEPH unit may require processing several files. The compiler maintains a pool of source files to be processed. Files in this pool are handled one at a time: a source is read, processed and closed before opening the next one. Once opened, a source file is read until either the end of file or an end-symbol is encountered. Everything after the end-symbol is ignored. Initially the pool contains the source file(s) submitted for compilation. Additional source files are added to the pool by the `require`, `library`, and `include` pragmats, see Section 6.4. Processing of library modules starts only after all non-library sources have been completed. Files provided by `require` and `include` pragmats inside a library module are treated as library files.

Before opening a source file, all pragmats are reset to their default values (see Chapter 6), except for the `compile` and `prototype` pragmats, which are handled as follows.

a) If the source file was submitted directly for compilation, these pragmats are set to `compile=on`, and `prototype=none`.
b) If the source file was added by a `request` or by a `library` pragmat, their values are `compile=off` and `prototype=import`.
c) If the source file was added by an `include` pragmat, both `compile` and `prototype` are reset to the values they took when the `include` pragmat was executed.

7.2 Modules

According to the best practice the name of the source file containing the source text of the module should resemble the module name. It is particularly relevant in ALEPH as modules are invoked by specifying file names, and not module names. While not recommended, different module units (that is, modules in different source files) can share the same module name, and, consequently, the same namespace.

The first non-comment line in a typical ALEPH module is a `module` pragmat, specifying both the module name and the namespace. This pragmat is followed by the public part of the module, or the *module head*. The head contains the prototypes of the (public) tags to be exported by, and defined in, this module. The module source is concluded by the private part, or *module body*, which defines the exported items together with the optional auxiliary, unexported items. The body is enclosed between `ifdef=compile` and `endif=compile` conditional pragmats.

The module is compiled by submitting its source file to the ALEPH compiler. The source file is opened with `compile=on` and `prototype=none`. The leading `module` pragmat informs the compiler that it is handling a *module*. Executing this pragmat sets the module name (and namespace) as well as the pragmat `prototype` to `public`. The prototypes in the module head specify items to be

exported. The conditional pragmat `if=compile` enclosing the body holds, thus the body is processed as well. The compiler checks that items with the *public* flag (that is, which have prototypes in the head) are indeed defined correctly.

When the module is *required* by a `require` pragmat, its source is opened with `compile=off`. The leading `module` pragmat specifies the name and namespace of the invoked module, and sets the `prototype` pragmat to `import`. Due to this setting the subsequent prototypes are considered to have declarations outside the unit under compilation, as is the case now. Reaching the `if=compile` conditional pragmat the condition fails, thus the module body is skipped.

Next to prototypes, the module head may contain additional ALEPH constructs. A `require` pragmat in the head automatically re-exports the imported items (using their original namespace), while declarations in the head are compiled into the invoking program locally, using the module's namespace. It may happen, without any program, that the head of module `A` requires module `B`, while the head of module `B` requires module `A`.

7.3 The module hierarchy

An ALEPH unit (a *main program* or *module*) may require several modules. Any module may also require other modules in its head, which modules may also require additional modules, and so on. The "`X` required module `Y`" relation defines a hierarchy among the involved units. In this hierarchy `A` is *below* `B` if there is a "require" chain from `A` to `B`. Resources provided by module `B` are automatically available for every unit below `B` in this hierarchy.

When `A` needs a resource, that resource might be provided by several modules. In the basic case, among the potential offers that module is chosen which is above `A` and has the *smallest rank*, that is, which requires the smallest number of "require" hops to reach it from `A`. By default, `A` has rank zero above itself.

The same module can be required by different modules, in which case this module appears in the hierarchy at several places. Nevertheless, it is still processed only once. Modules added as *user library* using the `library` pragmat, and modules required recursively by these library modules, form a second hierarchy. Elements of this hierarchy are *above* all elements of the first one by a very high hop number. Using this arrangement, resources defined by a library module are available to every plain module, but only as a last resort: if no other definition can be found, then consult the offers in library modules. On the top of the user library hierarchy there is still another hierarchy: the Standard Library providing the realization of many basic operations.

7.4 Tag definitions

Global tags can have a *qualifier* specifying the *namespace* this tag belongs to, such as `q::x`, where `q` is the qualifier. Without providing an explicit qualifier, tags in definitions (declarations and import prototypes) inherit the actual namespace. This namespace is empty in the main program; otherwise it is the same as the module name as defined by the `module` pragmat. The explicit qualifier, if

present, cannot be empty. A tag with a qualifier identifies only those definitions where the same implicit or explicit qualifier is used.

The process of finding the defining occurrence of a tag in the module hierarchy goes as follows. Suppose the tag with identifier x occurs in A, where A is either a module or the main program, and the tag has qualifier q which is either explicit or implicit. (In case A is the main program, the implicit qualifier is empty.) First, check modules which are *strictly below* A. If some of them defines q::x (where the qualifier in the definition can be either explicit or implicit), then the one with the *smallest absolute rank* (having the minimal number of "require" hops from the main program, see Section 7.3) is chosen.

If this step does not give result, then consider A and the modules *above* A. If x has no explicit qualifier, then it matches any definition of x in those modules; if x has an explicit qualifier q::x, then it matches only those definitions where the qualifier is (explicitly or implicitly set to) q. Among the candidate modules that one is chosen which has the smallest rank above A. If this search does not yield a unique definition, then it is an error.

This procedure is illustrated in the table below. Both modules in source files "f1" and "f2" have the first line

```
'pragmat'module=t.
```

which sets the module name to t for both; "f2" is required by the module in "f1", and "f1" is required by the main program. Tags (with the indicated explicit qualifier) in the "definition" column are defined in the module head; the numbers below them identify the definition. Numbers below the tags in the "usage" column show their defining occurrences.

file	require	module	definition	usage
"f2"		t	b v (1) (2)	a v x (3) (4) (6)
"f1"	"f2"	t	a v (3) (4)	a b x (3) (1) (6)
main	"f1"		a t::x x (5) (6) (7)	a t::a b v x (5) (3) (1) (4) (7)

The tag a used in the module from file "f1" (and also in the module from file "f2") has the implicit qualifier t, therefore it does not match definition (5) in the main program. The tag x in these modules matches definition (6) of t::x in the main program, but not definition (7) of x (which is without qualifier). Observe that definition (2) is not used at all; each occurence of v in "f2" uses definition (4) from "f1".

7.5 Macro rules

To improve efficiency, calls of rules declared in the same compilation unit can be implemented by textual substitution. When the rule name appears in a macro pragmat (Section 6.2), calls to this rule in the current compilation unit are replaced by the rule-body, and replacing the formal-affixes by the actual ones.

The substitution, however, can result in a syntactically incorrect program text, or in a different semantics. The following examples illustrate these cases and explain the additional restrictions a macro rule must satisfy.

a) In a macro, a formal in-variable affix cannot be assigned to. Indeed, suppose the rule `macro` is defined as

```
'function'macro+>x+y>: 1->y->x,x->y.
```

After textual substitution the replacement can be syntactically wrong as in

```
macro+1+z      ⇒     (1->z->1,1->z).
```

b) There is a problem with the dummy-symbol `#` as actual-affix. Using the same `macro` rule as above, the following substitution results in incorrect syntax:

```
macro+u+#      ⇒     (u->#->u, u->#).
```

c) While a macro rule can have a variable number of affixes, neither `shift affix block` nor `get affix blockno` can be used in a macro text as illustrated by the following example. The rule `is zero` below checks whether one of its arguments has value zero; while rule `math` computes the product of its arguments if none of them is zero, otherwise it computes their sum:

```
'question'is zero+@+>x: x=0; shift affix block+@,:is zero.

'function'math+y>+@+>x:
  is zero+@,0->y,(nxt:add+x+y+y,shift affix block+@,:nxt;+);
  1->y,(nxt:mult+x+y+y,shift affix block+@,:nxt;+).
```

If `is zero` were substituted verbatim in rule `math`, it would shift out all affixes and then the computation in `math` would not be carried over. Similarly, suppose the rule `macro` is defined as

```
'function'macro+a>+@+>q: q->b, get affix blocno+a+@.
```

where `b` is some global variable. After verbatim substitution the repeat block could vanish completely causing syntax error:

```
macro+b+2+T    ⇒     (2->b,get affix blockno+b+2+T)
```

d) Standard selectors are not carried over. Example:

```
'function'macro+t[]+x>: t[ptr]->x.
```

where `ptr` is some global variable. After textual substitution

```
macro+S+z      ⇒     (S[ptr]->z)
```

while `S` might not have a standard selector.

e) Out affixes get their values only after returning from a call. The rule call `swap+x+y+x` swaps the value of `x` and `y` if it is defined as

```
'function'swap+>a+>b>+c>: b->c,a->b.
```

but as a macro it does `(y->x,x->y)`, with a completely different result.

Items a) and c) are checked during compilation, and error messages are issued if the conditions are violated. For b), if the actual affix is the dummy affix `#`, the formal out affix in the macro is replaced by a newly created anonymous local variable (which may be removed during optimization). For d) the macro

substitution mechanism remembers the last substituted formal affix and uses that value for the standard selector what the rule call would do. For e) and other side effects, no warning is, or can be, given, but substitution clearly changes the semantics. So use macros with care.

8 Standard Library

Items in the Standard Library can be used in all programs without further notice. Any of these items, except for the variable affix handling rules, can be redeclared.

8.1 Integers

8.1.1 Constants

`'constant' max int, min int, int size.`
: `max int` is the largest representable integer, `min int` is the smallest (most negative) representable integer. `int size` is the number of decimal digits necessary to represent `max int`.

8.1.2 Arithmetical operations

`'function'add+>x+>y+z>, subtr+>x+>y+z>, mult+>x+>y+z>, div+>x+>y+z>.`
: These functions compute the standard arithmetic functions yielding the result in the last affix `x+y->z`; `x-y->z`; `x*y->z`; integer part of `x/y->z`.

`'function'addmult+>x+>y+>z+u>, divrem+>x+>y+q>+r>.`
: `addmult` yields `x*y+z` in `u`; `divrem` returns the quotient in `q` and the remainder in `r`: `x/y->q`, `x-y*q->r`.

`'function'incr+>x>, decr+>x>, get abs+>x+y>, min+>x+>y>, max+>x+>y>.`
: `incr` increases, `decr` decreaseseases its affix by one; `get abs` yields the absolute value of `x` in `y`; `min` and `max` yield `min(x,y)` and `max(x,y)`, respectively, in `y`.

8.1.3 Comparison

`'question'less+>p+>q, lseq+>p+>q, equal+>p+>q, not equal+>p+>q,`
`mreq+>p+>q, more+>p+>q.`
: These rules succeed if the corresponding relation holds between the integer values `p` and `q`.

`'question'is+>x, is true+>x.`
: succeeds if `x` is *not* zero, equivalent to the comparison `x!=0`.

`'question'is false+>x.`
: succeeds if `x` is zero, equivalent to the comparison `x=0`.

8.1.4 Bit operations

`'function'bool invert+>x+y>.`
: The bitwise complement of `x` is stored in `y`.

`'function'bool and+>x+>y+>z>, bool or+>x+>y+z>, bool xor+>x+>y+z>.`
: Bitwise Boolean operations.

`'function'left clear+>x>+>n.`
Shift `x` (including the sign bit) left by `n` positions; zeros enters from right. `n` must be positive and at most 32.

`'function'right clear+>x>+>n.`
Shift `x` (including the sign bit) right by `n` positions; zeros enter from left. `n` must be positive and at most 32.

8.2 Tables and stacks

`'question'was+T[]+>p.`
Succeeds if `p` is between `<<T` and `>>T`, inclusive. This rule can be used to test if `p` points into the list `T`.

`'function'next+T[]+>p>, previous+T[]+>p>.`
The calibre of `T` is added to, or subtracted from, `p`, moving the pointer to the next or previous block, respectively.

`'function'list length+T[]+len>.`
The number of actual elements (not blocks) in `T` is yielded in `len`. This value equals `-1+<>T+>>T-<<T`.

`'action'unstack+[]st[].`
The stack `st` must contain at least one block. The right-most block of `st` is removed; this operation decreases `>>st` by `<>st`. The location of the removed block can be reclaimed by an extension, its contents are lost.

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

`'action'scratch+[]st[].`
All blocks in `st` are removed. The locations can be reclaimed through extensions, but the contents are lost. After this operation `>>st` will be equal to `-1+<<st-<>st`.

`'action'release+[]st[].`
All blocks in `st` are removed, as in a call of `scratch`. Additionally, the memory allocated for `st` is also returned to the operating system.

`'predicate'request space+[]st[]+>n`
After a successful return the allocated actual memory of the stack `st` is guaranteed to hold `n` additional locations. The rule fails if the requested memory would go beyond the virtual limits of `st`, or the operating system cannot provide the requested memory.

8.3 Strings

Strings are stored in lists as special *string blocks*. In a list a string block is identified by the address of its last (rightmost) element, and contains the characters of the string in a packed form. All string handling rules assume that the affixes

determine a string block; in case they discover some discrepancy, the program run is aborted.

`'function'string length+T[]+>p+n>.`
: The pointer `p` must point to a string block in `T`. The number of unicode characters in the string is yielded in `n`.

`'function'string width+T[]+>p+n>.`
: The number of locations in the string block (the block size) is yielded in `n`.

`'action'previous string+T[]+>p>`
: The pointer `p` must point to a string block; `p` is moved to the (possibly non existing) location just before the string block. It is achieved by subtracting `string width` from `p`.

`'function'compare string+T1[]+>p1+T2[]+>p2+cmp>.`
: The two strings are compared lexicographically. If the first string is smaller than the second one, then `cmp` will be a negative number; if they are equal then `cmp` will be zero; if the first string is greater than the second one then `cmp` will be a positive number.

`'function'compare string n+T1[]+>p1+T2[]+>p2+>n+cmp>.`
: The two strings are compared lexicographically, but before comparison they are truncated to contain at most `n` characters. The value of `n` must be non-negative.

`'question'string elem+T[]+>p+>n+c>.`
: The pointer `p` must point to a string block in `T`. If the string has an `n`-th character counting from 0, the code of that character is yielded in `c` and `string elem` succeeds; otherwise it fails.

`'action'unstack string+[]st[].`
: The last block in the stack `st` (pointed by `>>st`) must be a string block. This block is removed from `st`. After this operation `>>st` decreased by the width of the string block.

`'action'pack string+from[]+>n+[]to[].`
: The rightmost `n` elements of `from` must be values that correspond to nonzero unicode codes; `n` must be non-negative. Characters with these codes are packed into a string block, and the stack `to` is extended with this block. After `pack string` returns, the new string is pointed by the actual upper limit `>>to`.

`'action'unpack string+T[]+>p+[]to[].`
: The pointer `p` must point to a string block in `T`. The string is unpacked yielding a sequence of `m` unicode characters, where `m` is the string length. The stack `to` is extended by the `m` character codes in left-to-right order.

`'action'copy string+T[]+>p+[]to[].`
: The stack `to` is extended by a string block identical to the one pointed by `p` in the list `T`.

8.4 Files

An ALEPH charfile or datafile is associated with a file-system element by an explicit or implicit call to the `open file` rule, and detached by a call to `close file`, or when the program terminates. Each ALEPH file has an associated error variable, and each file operation sets it. The error code is zero when the operation succeeds, otherwise it is different from zero and the value indicates the type of the error.

8.4.1 General

`'predicate'open file+""f+>mode+T[]+>p.`

Associate the file `f` with the file-system element specified by the string pointed to by `p` in the list `T`. The `mode` specifies how the file should be opened. It is `/r/` for reading, `/w/` for writing, and `/a/` for appending. Only character files can be opened for appending. The rule `open file` succeeds when the file was successfully associated with `f`. If the rule fails, the error code in `f` indicates the reason.

The special filenames `"<<stdin>>"`, `"<<stdout>>"` and `"<<stderr>>"` associate the ALEPH charfile `f` with the three standard streams, typically reading from, and printing to the console. The first can be opened for reading only, the other two for writing only.

`'predicate'open temp file+""f+[]T[]+>p.`

Associate `f` with a temporary file and open it for writing. The supplied string must end with six uppercase `X` letters, and it must be on a stack since the rule replaces these characters with other ones to make the filename unique.

`'action'close file+""f.`

Detaches the file `f`, possibly after writing out some buffered data. When the rule returns `f` is detached, but there might be file-system errors indicated by a non-zero error code.

`'function'get file error+""f+err>.`

Return the error code of the last operation on `f`. The code is zero if the operation was successful, otherwise it indicates the error condition.

`'function'get file pos+""f+pos>.`

Retrieve the actual file position in `pos`. The yielded value is correct only when `f` is opened for reading. The position is maintained internally, so there is no overhead in determining it. The position at the beginning of the file is zero.

`'action'set file pos+""f+>pos.`

If the file `f` is opened for reading, set the position to `pos`. The next item to be retrieved from `f` is the same as the next item when position was retrieved by `get file pos`. Setting file position to zero resets `f` before the first item.

`'action'unlink file+T[]+>p.`

Delete the file specified by the string in `T` pointed by `p`. Check the error code for success.

8.4.2 Character input and output

An ALEPH character file is a sequence of unicode characters stored using UTF-8 encoding. A charfile returns and accepts unicode characters, converting these characters to and from UTF-8 encoding on the fly. The unicode character with zero code is not allowed in an ALEPH character file. Character handling rules require a character file at their formal-file affix.

`'constant'new line, rest line.`

The ALEPH character with code `new line` marks the end of a line; it is identified with the *newline* unicode character with code 10. The constant-tag `rest line` is reserved for administrative purposes; its value is not a valid character code.

`'predicate'get char+""f+char>, fgetc+""f+char>.`

These rules are equivalent. If the file associated with `f` is not exhausted, the code of the next unicode character is delivered in `char`, and the file is advanced by one character. The rule fails if there are no more characters in the file.

`'predicate'ahead char+""f+char>.`

If the file associated with `f` is not exhausted, the code of the next unicode character is delivered in `char`, but the file is not advanced; a subsequent call to `ahead char` will deliver the same code. The rule fails if there are no more characters in the file.

`'predicate'get line+""f+[]st[]+cint>.`

If the file associated with `f` is exhausted, `get line` fails. Otherwise the next character is read. If it is a newline character, its code is assigned to `cint` and the character is skipped, otherwise `cint` is set to `rest line`. Then zero or more characters are read until either the end of file, or until the next character would be a newline. The stack `st` is extended with these characters in a left-to-right order.

`'predicate'get int+""f+int>.`

A call of `get int` will read and skip any number of spaces and newline characters until it finds a digit, plus-sign or minus-sign. If no such character was found, `get int` fails. Otherwise the rule will read and collect one or more digits until a non-digit is found: this non-digit is not read. The optional sign and the digits are considered as a signed decimal number; the value is delivered in `int`.

`'action'put char+""f+>char, fputc+""f+>char.`

These rules are equivalent. The value of `char` must be a positive valid unicode character. This character is written to the file associated with `f`.

`'action'put line+""f+T[]+>cint.`

The list `T` must only contain positive unicode values. These characters are written, in a left-to-right order, to the file associated with `f`. If `cint` is not `rest line`, this character is appended at the end.

`'action'put int+""f+>int.`

Exactly `intsize+1` characters are written to `f`. These characters are: zero or more spaces, the minus sign if `int` is negative, followed by the characters of the decimal representation of the absolute value of `int`.

`'action'put string+""f+T[]+>p, fprint string+""f+T[]+>p.`

These rules are equivalent. The pointer `p` must point into a string block in the list `T`. Characters of this string are written to the file associated with `f`.

`'action'fprintf+""f+t[]+@+>p.`

This rule provides a rudimentary formatted printing. The first two affixes after the file `f` specify a *format string*. Characters in this string are written to `f`, except for the following format character pairs, which specify how the additional in-affixes of the rule will be written.

format	the next affix is written as
`%c`	character,
`%d`	decimal integer without leading spaces,
`%x`	hexadecimal,
`%n`	newline (no affix is consumed).

8.4.3 Standard input and output

The following rules communicate directly with standard input and output.

`'charfile'STDIN, STDOUT.`

These character files are associated with the character streams *standard input* and *standard output*. Characters sent to `STDOUT` typically appear on the console; characters read from `STDIN` are the ones entered at the console. `STDIN` is line-buffered.

`'predicate'getc+char>.`

Get the next character from `STDIN`.

`'action'putc+>char, print char+>char.`

Both rules send the character `char` to `STDOUT`.

`'action'print int+>n, print string+T[]+>p, printf+t[]+@+>p.`

Corresponding to `put int`, `put string`, and `fprintf`. These rules write to `STDOUT` an integer, a string, and formatted output, respectively.

8.4.4 Data input and output

`'constant'numerical, pointer.`

These constants are predefined values that can be used as type indications in datafiles. For their usage see Section 4.3.2.

`'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`. When the rule fails the error variable associated with `file` gives information about the reason.

`'action'put data+""file+>data+>type.`

The file `file` must be a datafile. A data item is written to the file, consisting of the value `data` and the type `type`. For a more detailed description see Section 4.3.2. When the request cannot be executed (possible reasons are: the file-system is full; `file` was not opened and cannot be opened automatically; a pointer data is not covered in the file-area), the program run is terminated.

`'predicate'put datap+""file+>data+>type.`

Performs the same action as `put data`, but in case of an error fails and sets the error variable associated with `file`.

8.5 Miscellaneous

`'question'shift affix block+@.`

The actually visible repeat affix block is shifted out, and the next block is moved into its place, assuming that there is still a pending, unseen block. If there is no more blocks, the rule fails. This rule cannot be redefined.

`'function'get affix blockno+n>+@.`

The number of pending, unseen affix blocks is yielded in `n`. This number is always positive, and it is 1 if and only if `shift affix block` would fail. This rule cannot be redefined.

`'exit'exit+>code.`

This is the only Standard Library rule which is guaranteed not to return. The call of this rule closes all open files and terminates the program run with the specified `code`.

`'table'STDARG.`

The table `STDARG` contains, in reverse order, the command-line arguments as string blocks. Consequently, the upper limit `>>STDARG` of the table points to the first argument. The following compound-member walks through the arguments in their original order, and calls `cmdarg` with each of them:

```
(-ptr: >>STDARG->ptr, (nxt:ptr< <<STDARG;
          cmdarg+STDARG+ptr,previous string+STDARG+ptr,:nxt))
```

`'action'msleep+>ms.`

This rule sleeps for the specified amount of milliseconds before returning. As this rule clearly makes no "global changes", it would be tempting to declare it as a `'function'`. In that case, however, the compiler would complain that a function without out or inout formal affixes is a no-op and can be discarded.

`'action'backtrack.`

If the ALEPH program is linked with the `-g` flag, additional code is added which maintains the *call stack*, the list of embedded rule calls starting from the root of the main program and ending at the actual-rule which is currently executed. In this case a call of the rule `backtrack` provides a printout of the rule names in this stack. If the program was not linked with the `-g` flag, this rule does nothing and returns immediately.

`'action'wait for+T[]+>p.`

The pointer `p` must point to a string in the table `T`. The rule checks if there is a linked module with the specified name (as defined by the `module=...` pragmat) whose root has not been executed yet. If yes, it calls those roots and returns after all is finished; otherwise it returns immediately. The rule `wait for` aborts the program run if two module roots would wait for each other producing a deadlock.

9 Program representation

The program, produced by the notion program, consists of a series of terminals. Terminal notions fall into the following categories:

a) symbols,
b) tags (identifiers),
c) denotations.

An ALEPH program is a textual representation of the sequence of terminals sprinkled with *comments* and *formatting*. ALEPH has quite liberal formatting rules. Any number of white spaces (tabs, space and newline characters) are allowed before and after any terminal notion; spaces and tabular characters (but no newlines) are allowed (and ignored) inside identifiers, decimal, and hexadecimal constants. Thus

`new␣line`, `newline`, and `ne␣wli␣ne`

represent the same tag. Similarly,

`1␣000␣000`

is a legal representation of the number one million. There can be no space, however, in the head `0x` of a hexadecimal constant (since it is the representation of the hex-symbol which is required to appear exactly), while there can be spaces after it and between the hexadecimal digits, such as `0x␣ff␣e8`.

Any number of *long comments* can be added before and after a terminal notion. A long comment starts with a dollar-sign `$`, and ends at the next dollar sign or at the end of the line, whichever comes first. A short comment consists of a sharp-sign `#` followed by spaces, letters and digits so that it contains at least one non-space character. Short comments can be inserted before another comment or before a terminal notion, assuming either that the short comment ends with a newline, or the representation of the subsequent terminal does not start with a letter or digit.

9.1 Symbols

The character representation of terminals ending with -symbol is either a single character, such as the obvious representations of point-symbol or colon-symbol; two consecutive characters, such as the representation `->` of the arrow-symbol; or a "bold word" written between apostrophes such as the representation `'end'` of the end-symbol. When more than one representation is listed, any of them can be used with the same meaning. If the symbol has multi-character representation, no additional characters are allowed between the listed characters. For example, `0x` is the representation of the hex-symbol which indicates a hexadecimal number, while `0␣x` is a digit zero followed by the letter `x`.

symbol	representation
absolute-symbol	`/`
action-symbol	`'action'` or `'act'` or `'a'`
actual-affix-symbol	`+`

alwb-symbol	`<<`
anchor-symbol	`@` (at sign)
and-symbol	`&` (ampersand)
arrow-symbol	`->`
aupb-symbol	`>>`
box-symbol	`=`
bus-symbol	`]`
by-symbol	`/`
calibre-symbol	`<>`
charfile-symbol	`'charfile'`
close-symbol	`)`
colon-symbol	`:`
comma-symbol	`,`
complement-symbol	`~` (tilde)
constant-symbol	`'constant'` or `'cons'`
datafile-symbol	`'datafile'`
dummy-symbol	`#` or `?`
end-symbol	`'end'`
eq-symbol	`=`
equals-symbol	`=`
exit-symbol	`'exit'` or `'e'`
external-symbol	`'external'` or `'x'`
failure-symbol	`-`
formal-affix-symbol	`+`
function-symbol	`'function'` or `'fct'` or `'f'`
ge-symbol	`>=`
gt-symbol	`>`
hex-symbol	`0x`
le-symbol	`<=`
left-symbol	`<`
left-of-symbol	`(*`
local-affix-symbol	`-`
lt-symbol	`<`
minus-symbol	`-`
ne-symbol	`!=`
of-symbol	`*`
open-symbol	`(`
or-symbol	`|` (pipe)
plus-symbol	`+`
point-symbol	`.`
pragmat-symbol	`'pragmat'`
predicate-symbol	`'predicate'` or `'pred'` or `'p'`
qualifier-symbol	`::`

question-symbol	`'question'` or `'qu'` or `'q'`
quote-symbol	`"`
quote-image-symbol	`""`
repeat-symbol	`:`
right-symbol	`>`
right-of-symbol	`*)`
root-symbol	`'root'`
semicolon-symbol	`;`
stack-symbol	`'stack'`
static-symbol	`'static'`
sub-symbol	`[`
success-symbol	`+`
table-symbol	`'table'`
times-symbol	`*`
up-to-symbol	`:`
variable-symbol	`'variable'` or `'var'`
vlwb-symbol	`<`
vupb-symbol	`>`
xor-symbol	`^` (caret)

9.2 Tags

Identifiers in ALEPH are non-empty sequences of digits and latin letters starting with a lower or upper case letter. Inside identifiers spaces and tab characters (but not newlines) are allowed and ignored. A *qualifier* is an identifier followed by the qualifier-symbol; a *qualified identifier* is an identifier preceded by a qualifier. The representation of a local tag (that is, formal and local affixes and labels) is an identifier; the representation of a global tag is either an identifier, or a qualified identifier. Global tags without explicit qualifiers have an implicit one. The implicit qualifier is the name of the module where the tag is defined, or it is empty (with no corresponding explicit qualifier) when the tag is defined in the main program.

9.3 Denotations

Denotations specify a value which is determined during compilation. A denotation is either explicit, such as a sequence of decimal digits determining an integral constant, or implicit when the value is supplied by the compiler depending on the exact location of the denotation. Syntax:

integral-denotation:
    [minus-symbol], digit+;
    [minus-symbol], hex-symbol, hex-digit+;
    manifest-constant.
character-denotation: absolute-symbol, visible-character, absolute-symbol.

string-denotation:
    proper-string;
    manifest-string.
proper-string:
    quote-symbol, string-elem∗, quote-symbol.
string-elem:
    non-quote-character; quote-image-symbol.

The notion proper-string has been introduced to explicitly forbid manifest-strings at certain locations. Example of integral-, character- and string-denotations:

```
-0xffef, /t/, """quoted string""", _file_,
```

here the last one is a manifest-string. The implemented manifest constants and strings are the following.

`_line_`	constant, the source line number this constant appears,
`_file_`	string, the name of the source file,
`_module_`	string, module name as set by the `module=...` pragmat,
`_title_`	string, the title as set by the `title="..."` pragmat,
`_rule_`	string, the rule name (without the qualifier) if used inside an actual-rule.

Only the listed characters are allowed between the two underscore characters. The string returned by `_module_` and `_title_` is the empty string before encountering the `module` or the `title` pragmats. The manifest string `_rule_` similarly returns the empty string if used outside an actual-rule.

Integral denotations use decimal and hexadecimal digits. Decimal digits are the characters from `0` to `9`; hex-digits contain, additionally, both the lower case letters from `a` to `f` and the upper case letters from `A` to `F`. The value of an integral-denotation is the decimal or hexadecimal value of the sequence, negated if the minus-symbol precedes it. Spaces and tabular characters are allowed (and ignored) between the digits and also after the minus-symbol, but no spaces are allowed between `0` and `x` in `0x` marking a hexadecimal number, as it is the representation of the hex-symbol.

A visible-character is any unicode character which occupies a single character position. It includes the space character, but excludes the newline, tabular, and other control characters. The character-denotation provides the code of the character between the two absolute-symbols `/`.

A non-quote-character is a visible-character different from the quote mark `"`. A proper-string is a (possibly empty) sequence of non-quote-characters and quote-image-symbols (the latter is two consecutive quote marks) enclosed in quote marks. There is no escape character in ALEPH, therefore the newline character (and other control characters) cannot occur in a string-denotation.

ALEPH allows long strings to be split into shorter ones. Consecutive strings separated by white spaces (even if they are in different lines) are concatenated. Strings to be concatenated must be properly separated since inside a string two consecutive quote marks denote a single quote mark. Thus

`"a" "b"` is concatenated resulting in the two-character string `"ab"`, while

`"a""b"` is a single string with three characters: `a "` and `b`.

10 Examples

10.1 Towers of Hanoi

```
'pragmat'title="Towers of Hanoi".
'charfile'print=>"out.txt".
'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'
```

10.2 Printing Towers of Hanoi

```
'pragmat'title="Towers of Hanoi, full printing".
'stack'[=size=]a[],[=size=]b[],[=size=]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,
     print char+new line, decr+ln,:lines).
'action'print disc+st[]+>line-index:
   subtr+line+1+index,add+index+<<st+index,
   (was+st+index,print actual disc+st[index];
    print blank disc).
'action'print actual disc+>nmb-spc:
   subtr+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+>ch:
   cnt=0; print char+ch,decr+cnt,:repeat.
'root'-n: size->n,(fill a: n=0; decr+n, (* n->a *)a, :fill a),
   print towers,move tower+size+a+b+c.
'end'
```

10.3 Symbolic differentiation

```
'pragmat'title="Symbolic derivation".
'stack'[10](op,left,right)expr[].
'table'operator[]=("+":plus op, "-":min op, "*":tim op,
    "/":div op, "^":pow op,
    "log ": log op $ log(f) is represented as 0 "log" f $).
'stack'[1]const[]=("0": c zero, "1": c one, "2": c two).
'stack'[1]var[]=("x": x var).
'action'derivate+>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,
   derivate+f+df, derivate+g+dg,
   (=op*expr[e]=
    [plus op],M+df+plus op+dg+de;
    [min op], M+df+min op+dg+de;
    [tim op], M+df+tim op+g+n1,M+f+tim op+dg+n2,
                M+n1+plus op+n2+de;
    [div op], M+df+tim op+g+n1,M+f+tim op+dg+n2,
                M+n1+min op+n2+n1,M+g+pow op+c two+n2,
                M+n1+div op+n2+de;
    [log op], M+dg+div op+g+de;
    [pow op], M+g+min op+c one+n1,M+f+pow op+n1+n1,
              M+df+tim op+g+n2,M+n2+tim op+n1+n1,
              M+c zero+log op+f+n2,M+n2+tim op+dg+n2,
              M+f+pow op+g+n3,M+n2+tim op+n3+n2,
              M+n1+plus op+n2+de
    ).
'action'M+>left+>op+>right+res>:
  (*left->left, op->op,right->right*) expr, >>expr->res.
'action'print expr+>e:
   print expr1+e,print char+newline.
'action'print expr1+>e:
   was+const+e,print string+const+e;
   was+var+e,print string+var+e;
   print char+/(/,(= op*expr[e] =
     [plus op; min op; tim op; div op; pow op],
       print expr1+left*expr[e];+),
     print string+operator+op*expr[e],
     print expr1+right*expr[e],print char+/)/.
'stack'expr=((xvar->left->right,pow op->op): x to x,
             (log x->left, xvar->right,div op->op): log x by x,
             (c zero->left,xvar->right,log op->op): log x).
'action'test+>f-df-ddf:
   derivate+f+df,derivate+df+ddf,
   print expr+f,print expr+df,print expr+ddf.
'root'test+x to x, test+log x by x, test+log x.
'end'
```

10.4 Quicksort

Module:

```
'pragmat'module=quicksort,title="Quick sort module".
$ Sorting elements from a[from] until a[to]
'action'quicksort+>from+>to+[]a[].
'pragmat'if=compile. $ module body starts here
'action'quicksort+>from+>to+[]a[]
     -left-middle-right-amiddle:
   from >= to; $done
   from->left,to->right,add+from+to+middle,div+middle+2+middle,
   a[middle]->amiddle,
  (split: $ values smaller than amiddle go to the beginning
     (push right: left>to;
                  amiddle<a[left];
                  incr+left,:push right),
     (push left:  from>right;
                  a[right]<amiddle;
                  decr+right,:push left),
     (left<right,
        (-elem:
         a[left]->elem,a[right]->a[left],elem->a[right]),
        incr+left,decr+right,:split;
      middle<right,
         a[right]->a[middle],amiddle->a[right],decr+right;
      left<middle,
         a[left]->a[middle],amiddle->a[left],incr+left;
     +)
  ),quicksort+from+right+a,quicksort+left+to+a.
'root'+.
'pragmat'endif=compile.
'end'
```

Main program:

```
'pragmat'title="Applying quicksort".
'pragmat'require="quicksort".
'stack'[]A[]=(/3/,/1/,/4/,/6/,/5/,/3/,/2/,/3/,/1/).
'root'quicksort+<<A+>>A+A,put line+STDOUT+A+newline.
'end'
```

10.5 Permutations

Module:

```
'pragmat'module=perm,title="next permutation".
$ next permutation in lexicographic order; fail if last
'pragmat'macro=next perm.  $ compile to the invoking program
'predicate'next perm+[]st[]: perm from+<<st+st.
'predicate'perm from+>i+[]st[]. $ exported rule
```

```
'pragmat'if=compile. $ module body starts here
'action'invert+>from+>to+[]S[]-v:
   from>=to;
   S[from]->v,S[to]->S[from],v->S[to],incr+from,decr+to,:invert.
'predicate'perm from+>i+[]st[]-p-elem:
   i >= >>st,-;
   add+i+1+p,
     (perm from+p+st;
      st[i] >= st[p],-;
      invert+p+>>st+st,st[i]->elem,
        (rep:elem>st[p],incr+p,:rep;+),st[p]->st[i],elem->st[p]
     ).
'root'printf+"Permutation module initialized ...%n".
'pragmat'endif=compile.
'end'
```

Main program:

```
'pragmat'title="printing permutations".
'pragmat'require="perm".
'stack'[]T[]=(/a/,/b/,/c/,/d/).
'root'(nxt: put line+STDOUT+T+newline,(next perm+T,:nxt;+)).
'end'
```

11 References in the manual

[1] L. Csirmaz, *Aleph Compiler, v2.4*, available at `https://lcsirmaz.github.io/aleph/alephcomp.html`

[2] D. Grune, R. Bosch, L. G. L. T Meertens. *Aleph Manual* in: D. Grune, R. Bosh: *On the design of ALEPH*, *Mathematical Centre*, Amsterdam, 1986

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

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