An ALEPH Compiler

L. Csirmaz

May, 2020

1 Preface

In the early 1980's computer languages appeared in large numbers. Only a few of them survived and are in use even today. ALEPH, an acronym for "A Language Encouraging Program Hierarcy," almost completely disappeared, and this work is an attempt to resurrect it.

As a programming language, ALEPH has an interesting structure, unusual parameter passing mechanism, and a limited range of constructs. ALEPH compilers have been constructed for a wide range of computer architectures (which had a much larger variety at that time than it has today), and these compilers generated efficient and succinct code (an important requirement those days). Designed by D. Grune, R. Bosch and L.G.L.T. Meertens in the Mathematisch Centrum, Amsterdam in the late seventies, it was "suitable for any problem that suggests top-down analysis (parsers, search algorithms, combinatorial problems, artificial intelligence problems, etc)". It is no wonder, that an early ALEPH compiler was written in ALEPH itself.

2 Language features

ALEPH has some interesting features which cannot be found in today's mainstream languages. This section discusses some of them: its calling mechanism, virtual memory, and datafiles.

2.1 Call-then-store calling mechanism

Probably the most prominent distinguishing feature of ALEPH is its parameter passing mechanism. Arguments – called affixes – of ALEPH routines – called rules – fall into two main categories: *incoming* and *outgoing* ones (some of the arguments can be both). Inside the routine arguments behave like local variables, and it is the caller's responsibility to store the values of the outgoing parameters in their final destination, which is done only if the called routine reports success.

This feature handles routines with several return values naturally without the need of introducing compound structures tailored for each such routine. If not needed, the caller can discard any of the outgoing values. The basic example is the `divrem` function with two in(coming) and two out(going) arguments: the first out argument is the result of integer division, and the second one is the remainder. This function is typically implemented as a single machine instruction, leaving the two results in two different registers.

ALEPH's *call then store* calling mechanism automatically controls the side effects through parameters. Usually, when a parameter is passed by reference, any change to the parameter's value takes effect immediately, and changing the value of the global variable which has been passed as argument changes the parameter's value as well – sometimes creating a bug rather difficult to find. In contrast, in ALEPH the value of a parameter changes only when it is changed explicitly; and this change does not escalate (thus revocable) until the routine returns.

2.2 Tractable data flow

Data flow in ALEPH is quite restricted: a simple case selection – reminiscent of LL(1) parsing –, and a jump which is an abbreviation for tail recursion. There are no repetitive statements in ALEPH; iteration is handled by recursion. It means that data flow inside a routine is tractable: liveliness and reachability properties can be checked statically during compilation. In particular, an ALEPH compiler can check that

- all instructions are reachable;
- the flow can reach a return point;
- when a local variable or parameter is used it has a value assigned to it (it is not "uninitialized");
- out parameters have been assigned values when the routine reaches any of its return points;
- if a local variable (or parameter) has been assigned a value, it is actually used.

There are no uninitialized global values in an ALEPH program. Variables and tables are initialized when they are declared; and a stack can be extended only by supplying values for the extension.

2.3 ALEPH data types

The basic data type in ALEPH is *integer*, which is a synonym for a *word* in the target machine. An integer can be considered a *bit string* (which has a positive, zero, or negative value), or as a *pointer* to some location inside a table or stack which, again, contains an integer (see Section 2.4 on how pointers are handled). Consecutive memory locations in a table or stack can be grouped to become a data block or a string. Such a group is pointed to by its last (topmost or rightmost) element. Block elements are identified relative to the address of the block by a selector. Strings, on the other hand, behave like black boxes, and can be manipulated using built-in routines (externals) only. This way string elements are not restricted to bytes (or wide characters), and can contain elements of an arbitrary character set. Our implementation uses the Unicode character set, strings are stored using the UTF-8 encoding. (It also means that extracting string elements is an expensive operation.) Changing the character set (or the encoding) is completely transparent to the ALEPH program.

A stack or integer array can be extended by adding a block (specifying the value of each block element) or a string (specifying characters in the string in some way). Composite objects are referred to by pointers to their last elements. There are no special data type for such pointers.

As mentioned above, ALEPH has tables and stacks as fixed and extendable array of integer values. For communicating with the outside world it has charfiles and datafiles.

2.4 Virtual memory

ALEPH pointers can only refer to elements in tables and stacks. The complete *virtual memory space* – the allowed range of indices – is distributed among the arrays of the program. Virtual limits are fixed and do not change during runtime, and pointers refer to a list element using its virtual address. The virtual address space of different lists are disjoint, thus a pointer uniquely identifies the list it points into.

A stack does not necessarily occupies its virtual space completely. Existing locations (which correspond to locations in the machine memory) form a presumably much smaller continuous subrange, the *actual memory range*. Stacks can be extended to the right (upwards) until the end of their virtual memory parts, or until there is enough physical memory available. They can shrink from the right when their actual upper limits are lowered; the released virtual memory can be reclaimed again. Stacks can also shrink from the left (behaving like queues), but in this case the released virtual space is lost (for the rest of the program run) and cannot be reclaimed again.

Virtual addresses refer to list elements (i.e., integers), and not directly to the memory. It means that to get a pointer to the next (or previous) list element, the given pointer should be increased (decreased) by 1. Using a 32-bit word size, the virtual address space is limited to 2^{32} words (not bytes). This implementation assumes 32-bit word size. The virtual address space consists of the positive integers between M and $2^{32} - M$ where M is around one million. This has been chosen so that pointers and small values (e.g., characters) would be different; a typical non-pointer value will not be accepted as a pointer. The actual memory an ALEPH program can access at the same time is around 8 Gbytes.

2.5 Datafiles

ALEPH programs communicate with the outside world through *files*. ALEPH distinguishes between two file types: character and data. Character files accept and read characters from the character set used; in our version, Unicode characters. Datafiles can be used to communicate between different ALEPH programs. Data files are written and read one item a time, where an item is either an integer or a pointer. As the programmer does not have direct control over how virtual addresses are distributed, the datafile should not store the values of the pointers but the *list* the pointer points to and the *r*elative address of the pointed item in that list. From this information the pointer can be restored independently of the virtual address distribution.

In a datafile declaration, all lists must be specified to which the datafile can have pointers. By storing the lists in the datafile first, each additional item requires a single extra bit only, which specifies whether the item is a pointer or not. When opening an ALEPH datafile, stored lists are paired with the specified ones so that the appropriate pointer transformation could be made. How the pairing should be done is not specified by the ALEPH Manual. One possibility is to match them by position; another one is to match them by name. This implementation uses the second possibility, but instead of the list name it uses a 32-bit hash value of the name stored in the file. When opening a datafile, each specified list is matched to the one with the same hash. It is an error if a pointer in the file has no matching list.

3 Syntax

The ALEPH programming language is defined in the *Manual* which was published several times by the Mathematisch Centrum, Amsterdam, from 1974 until 1982. Implementations of any computer language frequently add new features, restrict or leave out others. This happened with ALEPH as well; these changes are also reflected in the different printings of the Manual. The ALEPH language implemented – and used – by the present compiler is no exception. Making modular programming possible – as opposed to using monolith programs prevalent at the time of the creation of ALEPH – required some new features. Other extensions and restrictions came up naturally; some of them date back to the time of the first ALEPH compilers in the early 80's. Changes introduced here respect the original design ideas and philosophy as much as possible.

3.1 Program text representation

ALEPH is an Algol-like language in which key words are distinguished by a different typeface. In practice coding is typically done using a single (monospace) typeface. There are several approaches to distinguishing keywords from the surrounding text, but none of them is perfect. In this implementation, bold face keywords are enclosed between apostrophe characters, e.g.

```
  'variable'x=0.
  'root'print int+STDOUT+x.
  'end'
```

Other approaches are using CAPITAL letters for keywords (as in several PASCAL implementations); restricting keywords (as in C and related languages); using an initial escape character only and whitespace at the end (i.e., leave out the closing apostrophe); etc.

Characters in ALEPH are not restricted to eight bits, and character strings cannot be indexed. In this implementation any UTF-8 character can be a string character. According to the ALEPH Manual `newline` and `newpage` are not characters. This implementation relaxes this restriction, and the newline characters can be part of a string, but not of a string denotation (strings in the program text). All strings should be closed in the same line they start in. Two separate consecutive strings (even if they are on different lines) are concatenated, thus strings can be continued on the next line.

3.2 Hexadecimal constants

Whereever integers are accepted, constants in hexadecimal notation are recognized and accepted as well. An example is `0x1234abcd`. A minus sign `-` can also appear before a hexadecimal constant.

3.3 Relations

The external rule `equal+x+y` tests the equality of `x` and `y`. The same test can be done by using the identity `x=y`. Similar shorthands were added (that were not present in the original language) for the other comparison operators:

`x!=y` or `x-=y`	for `x` and `y` differ;
`x<=y`	for `x` is less than or equal to `y`;
`x<y`	for `x` is smaller than `y`;
`x>y`	for `x` is greater than `y`;
`x>=y`	for `x` is greater than or equal to `y`.

3.4 Lists in scalar context

In an affix position where a single integer is required, a list `L` (a table or stack) can appear with the meaning that the value is that of its topmost (rightmost) element, namely, `L[>>L]`. Similar abbreviation exists for lists with selectors. Thus the rule call

```
add + a*L + b*L + L
```

is equivalent to

```
add + a*L[>>L] + b*L[>>L] + L*L[>>L]
```

This extension means that occasionally a stack looks – and behaves – like a variable, making writing and comprehending stack operations easier.

This extension come naturally, and was really handy in cases when the content of a newly added block on the top of a stack had to be manipulated. In cases when the list was entered by mistake, the extension prevents the compiler from complaining about affix type mismatch. The implicit reference to the topmost list block came, unexpectedly, as an extra complication during macro substitution.

3.5 Actual and virtual limits

Each table and stack has a *virtual address space* assigned by the compiler (with almost no control of the programmer) which is fixed during the run of the program. The *actual address space* of stacks changes when the stack is extended or shrunken, see Section 2.4. For a list `L` the constructs `<<L` and `>>L` return the actual lower and upper bounds of `L`, respectively. To get the *virtual* limits of the same list, one can use `<L` and `>L` (with a single left symbol and right symbol). In constant expressions only virtual limits can be used (as the actual limit can change during program run).

For tables the actual and virtual limits are always equal. In case of stacks actual limits are always within the virtual limits. Fixed stacks (i.e. stacks without size estimate) have equal actual and virtual limits.

3.6 Dummy affix

When the value of an out formal affix is not needed (the value can be thrown away), rather than forcing the programmer to invent some dummy variable, the dummy symbol can be used. This symbol has representation `?` by the Manual, this implementation also accepts the sharp-sign `#`. Visually it looks better, not indicating any uncertainty. Compare the two lines below:

```
divrem+a+5+#+rem,
divrem+x+10+?+q.
```

3.7 String as actual affix

A string denotation can be used as an actual affix. This extension simplified to write some program texts, and, hopefully, made it more transparent by bringing the string definition and usage closer. Without this feature strings are typically put into a table with a pointer constant pointing to the string, and then use the table name and the string pointer:

```
'table'MESSAGE=("unknown identifier":unknown tag).
'action'tag error:error+MESSAGE+unknown tag.
```

With this extension the string can appear as an argument:

```
'action'tag error:error+"unknown identifier".
```

Actually, as an actual affix a string translates to two affixes: first a special table and second a pointer which points to the string in that table. The specified string is automatically added as a filling to that table. The compiler might (or might not) merge identical strings.

3.8 Root rule

The root of an ALEPH program (the only directly executable instruction) can have local affixes and a rule body. As it is executed only once, thus there is no need to designate a separate rule for this purpose. An example is:

```
'root'(rep:put line+STDOUT+V+newline,
         (next perm+<<V+V,:rep;+)).
```

3.9 Matching formal and actual list affixes

Every table and stack has an associated *block structure* which determines the block size of that list together with the set of selectors. By default, the block size is 1, and the selector is the same as the list name. Otherwise the list definition must have a selector list which contains the selectors in a left to right order. For example, the formal stack definition

```
[](tag,left,right)tags[]
```

specifies that the block structure of the formal stack tags has three elements with selectors `tag`, `left`, and `right`, where `right` is the rightmost element of the block. Also, the list `tags` has no *standard selector*, namely a selector with the name of the list.

Neither in a formal definition nor in a declaration the list's selector list pack can be the empty `()` (the ALEPH Manual allows this in formal list definition). The selector list pack, however, can be missing completely. The blocks of the formal and of the actual lists are compared as follows.

- the formal list has no selector list pack:
  there is no restriction on the actual list. Observe, however, that in this case the standard selector of the formal and the actual might be different. Suppose we have the rule declaration

  ```
  'action'set zero+[]st[]: 0->st.
  ```

  which sets the topmost element of the stack `st` to zero. (It is an action as it makes a global change.) With the declaration `'stack'(L,b)L` the assignment `0->L` clears the second to last element of `L`, while `set zero+L` clears the last element of the list.
- the formal list has a non-empty selector list pack:
  the actual list must have the same block size and the same standard selector (selector names, however, can be different). If this restriction is violated, a warning is issued; if the called rule is a macro, then this is an error.

3.10 Extension syntax

An extension adds a new block to the top of a (formal or global) stack. It is specified as a sequence of assignments where the destinations are the selectors of the block; it is enclosed between `*` symbols, followed by the list tag. If the stack `st` has three selectors `sel`, `ect`, and `ors`, then the extension

```
* pnt->sel, 0->ect->ors * st
```

extends `st` with a block of three elements. To make such extensions more appealing visually, parentheses can be inserted *before* and *after* the `*` as follows:

```
(* pnt->sel, 0->ect->ors *) st
```

This change destroys the $LL(1)$ property of the ALEPH syntax, as a `(*` sequence can be either the start of an extension, or that of a compound block which has an extension as its first member. Nevertheless the improved readability of ALEPH programs justifies the additional parser work.

3.11 Expressions

In ALEPH an expression must have a value which can be determined during compilation. Originally expressions can be used at several places; this implementation restricts them to constant and variable declarations only. It is not an essential restriction as new constant tags can be declared with the desired values.

An expression must evaluate to a constant value. An expression can refer to a constant tag declared later (or in another module), but cannot depend on itself. Thus

```
'constant'a=b+2.
'constant'b=/a/.
```

is accepted (here `/a/` is the value of the character `a` in the used coding), while

```
'constant'p=q+1,q=1-p.
```

is not, and gives an error message.

Next to the usual arithmetic operators `+`, `-`, `*` and `/`, the following Boolean operators can be used:

- `~x` for the (binary) complement of `x`;
- `x&y` and `x|y` for the bitwise and and bitwise or;
- `x^y` for the bitwise xor (modulo 2 addition) operator.

They have lower priorities than the arithmetic operators.

In expressions integer denotations (both decimal and hexadecimal), constant tags, pointer constants (defined in fillings), virtual bounds and list block size (caliber) can be used. List size estimates and repeat numbers (see Section 3.13) are evaluated before the virtual bounds are determined, thus these values cannot depend on these bounds.

3.12 Stack size estimate

A stack declaration specifies the virtual address space size this stack requires. This size estimate can either be *fixed* or *relative*. In both cases it must be an integer denotation or a constant tag; no expression is allowed.

The *fixed size* is specified between = symbols; its value cannot be larger than 1,000,000 (and, of course, must be positive). The compiler reserves at least as much virtual space for the stack. (The final virtual space can be larger if the stack has fillings which totals to a larger amount.)

The *relative size* estimate should give a positive integer not bigger than 100. After reserving virtual addresses for tables and fixed size stacks, the remaining space is distributed proportional to the requested relative amount.

3.13 Filling

Next to selectors and block size, a table and stack declarations may specify the initial content of the list. This filling is a sequence of integer denotations, constant tags (including pointers), strings, and blocks. Each one can be followed by the repeat symbol `*`, followed by either an integer or by a constant tag specifying how many times this item should be repeated. Then the optional pointer initialization follows: a colon `:` and a constant tag which is defined to have the the value of virtual address of the last defined list item. Example:

```
'constant'tsize=10.
'stack'T=(0*tsize:tzero,1*tsize,"string":tstring).
```

adds ten zeroes, ten ones, followed by the internal representation of the string `string`. It also declares `tzero` to be the (virtual) address of the lastly added `0`, and `tstring` to be the (virtual) address of the last element of `T` (which, if no further filling is added, is the same as `>>T`).

In the filling a compound block defines the content of a list block. The compound block must have exactly as many elements as the list block has (violating this requirement will result in a warning). A block element must be either an integer or a constant tag (possibly a pointer constant). In the block the constant value is followed by an arrow symbol `->` and the selector where it must be stored. Example:

```
'stack'(ch,prio)optor=(
 (/+/->ch,3->p), (3->p,/-/->ch),(5->p,/^/->ch)).
```

which adds three blocks of size two to the stack `optor`.

The original block syntax is also accepted: the compound block of the filling contains, in the left to right order, the values which should be added to the list. One of the integer values can be followed by the repeat symbol `*` with the meaning that this element will be repeated as many times as necessary to fill the whole list block. Example:

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

adds 100 blocks to the stack `big block`, each consisting of eight zeroes.

3.14 Multiple list filling

Fillings for a list can spread across the program (actually, can spread across several modules). A list description (without size estimate), followed by `=` and a filling can appear multiple times across the program. Fillings specified this way are accumulated. Their order is unspecified, but within a single filling the order of the added elements is kept intact.

If a stack has no declaration, namely a list head with a size estimate, then its virtual size will be equal to the total size of the fillings. The stack can still shrink, but cannot expand beyond its virtual upper limit.

3.15 Exit rule type

Executing the terminator `'exit'16` causes the program to terminate with exit value 16. The `'exit'` statement is replaced internally by a call of the external rule `exit`, in this case as `exit+16`. Consequently `'exit'` must be followed by an actual affix, and not by an expression.

In general, next to the four rule types *predicate*, *question*, *action*, and *function*, a fifth is added: *exit*. A rule is of type *exit* if it never returns. Thus the (external) rule `exit` is of type *exit*, as well as the rule

```
'exit'was error+>x:
     x>=0,exit+0;
     put string+STDERR+"Error:",put int+STDERR+x,exit+-1.
```

This rule prints some additional message before terminating the program run. A rule of type *exit*

· should not have out or inout affixes (as there is no way to use the returned values); and
· cannot return.

When an exit rule is defined, these conditions are checked. When such a rule is used, it is treated as a terminator which can neither succeed nor fail.

3.16 Classification

A classification chooses exactly one of the possible alternatives based on the value of a source included in the classifier box. An example is

```
 (= last*L[n] =
  [0;1],   action 1;
  [-10:10],action 2;
  [L;1000],action 3;
  [:],     action 4)
```

The area in the square brackets determines whether the alternative following it will be chosen. No expression is allowed in an area, only integer denotation (decimal or hexadecimal), constant tag (including constant pointer) and global (not formal) list. This latter stands for its virtual address range. All values in an area are determined during compilation, and the compiler gives an error message if some of the alternatives cannot be reached (this happens, for example, if the first two areas in the above example are swapped); and gives a warning if it is possible that none of the alternatives are chosen. When running the program and none of the alternatives succeeds the program run is aborted with an error message.

3.17 File area, file string

In a file declaration an area can be given which restricts what values are allowed to send to or receive from that file. In this implementation no area is allowed for character files, and the area of a datafile should contain only those lists (tables and stacks) to which pointers will be sent to or received from. How datafiles work is discussed in Section 2.5.

The string denotation and the direction (the > symbol before and after the string) is used as follows. Files can be opened by the external rule

```
    'a'open file+""file + >mode + t[]+>ptr.
```

where `mode` is the character `/r/` for reading, `/w/` for writing, and `/a/` for appending (appending is implemented for character files only); the last two arguments specify the file name string (with possible path information).

Without explicitly opening the file, the first file operation tries to open it. The string denotation gives the file name, and the direction restricts the access: the file opens automatically for reading only if there is a > *before* the string, and for writing if there is a > *after* the string.

3.18 Static stack and static variable

Variables and stacks can be declared to be static by adding the **'static'** keyword before their declaration, such in

```
'static''variable'resources=0.
'static''stack'[=20=]values.
```

Static variables and stacks behave identically to variables and stacks in the module they are declared. In other modules, however, they are "read only", which means that other modules cannot change the value of a static variable, and cannot modify, extend, shrink, or manipulate otherwise a static stack.

3.19 Prototype

A prototype informs the compiler about a tag. A table or stack prototype has no size estimate and filling; a constant, variable, file prototype has no data (or initial value); a rule prototype has no actual rule. They look like an external declaration without the **'external'** keyword and the string denotation. Example:

```
'charfile'PRINTER.
'action'print tag+>tag.
'constant'max tag pointer.
```

Prototypes are used to inform the compiler about tags which are defined in other modules, see Section 3.22 more on modules.

3.20 Pragmats

Pragmats are used to control certain aspects of the compilation; their semantics changed significantly compared to the ALEPH Manual. The following pragmats are recognized by this implementation:

`tab size=8`	sets tab size for program text printing
`library=on/off`	switch to library mode
`stdlib=off`	don't read the standard library
`list=on/off`	switch program text printing
`warning level=8`	set warning level between 0 and 9
`dictionary=on/off`	collect where each tag occurred in the program
`compile=on/off`	pragmat flag, see below
`title="title"`	specify program title
`module="title"`	specify module name, switch to module mode
`bounds=on/off`	compile with index checking
`count=on/off`	count how many times rules are called
`trace=on/off`	trace rule calls and affixes
`include="file"`	add `file` to the sources to be read
`require="file"`	require module headers from `file`
`prototype=none`	specify how prototypes are handled (Section 3.19)
`macro=rule`	`rule` should be treated as a macro.

Command-line arguments starting with double dash are pragmats:

```
--XXXXX
```

is parsed as

```
 'pragmat' XXXXX .
```

except that no conditional pragmats are accepted (see below). There are other command-line shorthand starting with a single dash:

`-L`	`library=on`
`-l`	`list=on`
`-M`	`prototype=public`
`-d`	`dictionary=on`
`-W`	`warning level=3`
`-I XXXX`	`path="XXXX"` (only in command-line)
`-o XXXX`	`title="XXXX"`

There is no `path` pragmat, it can only be used as a command-line argument. The path specifies the directories, separated by `:`, which are searched for source files (included and required) and the standard library.

Standard values for some pragmats are

```
tab size=8,
list=off,
dictionary=off,
library=off,
compile=on,
prototype=none.
```

The pragmat `include=("file1","file2")` adds these files to the end of the source list to be processed. Similarly, `require=("file1","file2")` also adds these files. The difference is the following. When the compiler opens a source file, pragmat values are reset to their default values as indicated above. For `include` files the `compile` and `prototype` pragmats are set to values which were in effect when the source file was added to the list. For `require` files

```
  'pragmat'compile=off,prototype=import.
```

is added. It is an error to add the same file both as `include` and as `require` at the same time.

When `compile=off` is in effect, the `title` and `module` pragmats are ignored; the `compile` pragmat has no other effects for the working of the compiler.

The `prototype` pragmat has three possible values: `import`, `public`, and `none`. In the first case a prototype indicates that the tag has a declaration outside this module (and then it cannot be defined, but can have other prototypes). In the second case every tag appearing in a prototype gets automatically the *public* flag, and must be defined in this module (consequently cannot be external, or imported). When `prototype=none`, prototypes are only for checking purposes and do not imply any specific behavior.

The `module` pragmat, if not skipped (i.e., if `compile=on` is in effect), then defines the module name, and also sets `prototype=public`.

3.21 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.
```

where `TAG` is one of `compile`, `list`, `dictionary`, `module`, `library`, etc. The program text between the `if` and `else` pragmats is processed if `TAG` is in effect (it is not in effect), otherwise it is skipped; and just the opposite for the text between `else` and `endif`. The `else` part may be missing. The `TAG` in `ifdef` (`ifndef`) pragmats can be any identifier (tag), and it is checked if this identifier has (has not) been defined until this points. As an example, the pragmat

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

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

The `if` ... `endif` pragmats must be nested properly, and the ignored 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.

3.22 ALEPH modules

An ALEPH module starts with a `module` pragmat which defines the module's name (title). After this the module header follows which contains the prototype of all exported (public) tags. The module's body, which defines the exported tags together with private items, is enclosed between `ifdef=compile` and `endif=compile` pragmats. A small module exporting the rule `do something` and the stack `LEXT` can be defined as follows.

```
 'pragmat'module="sample".  $ module name $
 'action'do something+>in+out>.
 'stack'(adm,left,right)LEXT.
 'pragmat'if=compile.       $ module body $
  'stack'[12](adm,left,right)LEXT=((3,4,5):first item).
  'action'do something+>x+y>: add+first item+x+y.
  'root'+.
 'pragmat'endif=compile.    $ until this  $
 'end'
```

When the module is compiled, the source is read with the pragmat `compile=on`. Thus the `module` pragmat is executed, which sets the `prototype` pragmat to `public`. The module head is parsed, and the compiler marks all prototyped tags to be exported, thus they must have a definition later on. The condition in the `if=compile` pragmat holds, and the material in the module body is read and compiled.

When the module is required by giving **'pragmat'**`require="sample"`, then the module text is parsed with `compile=off` and `prototype=import`. The `module` pragmat at the top is ignored (as now `compile=off`), then the prototypes are scanned. The compiler marks all prototyped tags as "to be defined in another module" and complains if the same tag is defined or not used properly. Reaching the conditional pragmat `if=compile` it fails, consequently the remaining text of the module is skipped.

Modules can require public items from other modules; it may happen that module `a` requires some public item from module `b`, and module `b` requires some public item from module `a`.

As in the sample module above, all modules must have a **'root'**. All module roots are executed before the main root is called (which must be in the only non-module source file). The module root can make the necessary initialization. If no such initialization is necessary, the module can use an empty root as it happened in the example. To control the order of the initialization, a module can call the rule

```
wait for+"another module"
```

This rule checks if there is a module with the supplied title. If not, it fails and returns immediately. Otherwise, before returning success, it makes sure that the referred module has finished initialization. The `wait for` rule aborts the program run with an error message if two modules would wait for each other, thus use this rule with care.

3.23 Library mode

The `library` pragmat determines whether library extensions are allowed or not. **'pragmat'**`library=on` turns the library mode on, while **'pragmat'**`library=off` turns it off. The standard library, if not disabled by **'pragmat'**`stdlib=off`, will be read as the last source.

In library mode the `@` character is considered to be a letter. This way private tags can be created which are not available outside the library. Dictionary listing ignores tags starting with `@`.

External declarations are allowed in library mode only.

The inquire symbol `?`, followed by a tag, can be used to check if the tag has been used but not defined. If this is the case, the inquire symbol and the tag is discarded, and the source is processed. If it is not the case, (that is, either the tag was not used, or it has had a previous definition), then the program text is skipped until the next point symbol. The library snippet

```
?nlcr    'action'nlcr:put char+STDOUT+newline.
?newline 'constant'newline=10.
```

checks first if `nlcr` has been used so far without definition. If yes, the rule definition following `?nlcr` is read, otherwise it is skipped. The next line defines the `newline` constant to correspond to the newline character with code 10. The order of the inquire statements are important. For example a second inquiry starting with `?nlcr` would always be skipped.

3.24 Predefined constants

There are five built-in constants. They start and end with an underscore, and can be used anywhere where an integer denotation or a string is accepted.

`_line_`	integer, the source line number this constant appears
`_file_`	string, name of the the source file
`_source_`	same as `_file_`
`_title_`	string, the title as set by `title=".."` or `module=".."`
`_rule_`	string, the rule name if used inside a rule definition.

4 Variable number of affixes

Allowing – and handling – variable number of affixes in rules is the main novelty of this ALEPH compiler. Variable number of affixes raise several problems, and pose a main obstacle keeping the tractability of the rule properties. The approach used here meets two main requirements: on one hand it provides a flexible and usable variable argument mechanism, and on the other it keeps all tractability properties of ALEPH programs. This is achieved by restricting how the "invisible" affixes in the variable block can be accessed.

A formal affix sequence, which defines the arguments of a rule, can contain a repeat affix symbol `*` indicating that the affixes from that point to the right can be repeated indefinitely. The rule `sum` in the declaration

```
'function'sum+a>+*+>b+>c:
  0->a,(nxt:add+a+b+a,add+a+c+a,
            shift affix block+*,:nxt;+).
```

has the single out affix `a`, and two input affixes `b` and `c`, the latter two forms the repeat affix block. Invoking this rule requires three, five, seven, and so on actual affixes. The first three of the actual affixes are matched against the formal affixes `a`, `b`, and `c`. The rule execution starts by setting `a` to zero, then adding `b` and `c` to `a`. The library routine `shift affix block` shifts out the affix block and moves the next block into its place, assuming there is still a pending, not seen block. If there is none, `shift affix block` fails; and then the rule `sum` ends. Otherwise the execution continues, and a jump is made to the label `nxt`, where the next two input affixes are added to `a`. Finally, `sum` adds up all of its input arguments and return their sum in `a`.

The main point which guarantees tractability data flow is that the shifted out blocks are lost to this rule, there is no way to reach or modify their elements.

The repeat affix symbol `*` can appear as the last actual affix in a call of `sum` with the meaning that the present and all subsequent affix blocks of this rule are passed as arguments. The rule `negate` declared as

```
'function'negate+a>+*+>b+>c-z:
   add+b+c+a,(shift affix block+*,sum+z+*,subtr+a+z+a;+).
```

adds up its first two in affixes `b` and `c`; if there are no more affixes then this sum is the result. Otherwise it calls `sum` to add up the rest, and then subtract it from the sum of the first two. Thus the calls

```
negate+x+3+4, negate+y+3+4+0+1, negate+z+3+4+0+1+x+y
```

put 7 to `x`, 6 to `y`, and −7 to `z`.

The library routine `get affix blockno+n>+*` returns the number of pending affix blocks. This number is always positive, and it is exactly `1` if and only if `shift affix block+*` would fail.

The main application of variable number of arguments is formatted printing, which, in the ALEPH compiler, has mainly been used to report error messages, but it also turned out to be useful in code generation. A rudimentary formatted printing can start with the format string – so it is two affixes: the table and the string pointer. Each format character starting with `%` should have a corresponding affix. When encountering a format char, the affix list is shifted and the next argument, together with the format char, is passed to the rule `handle format char`.

```
'action'format print+T[]+*+>arg-fmt-n-ch:
  arg->fmt,0->n, $ fmt is the string pointer $
  (nxt:string elem+T+fmt+n+ch,incr+n,
         ((ch=/%/,string elem+T+fmt+n+ch,incr+n,
                                        shift affix block+*),
            handle format char+ch+arg,:nxt;
          put char+STDOUT+ch,:nxt);
       put char+STDOUT+newline).
```

Another application can be pushing and popping unspecified number of elements to a stack:

```
'action'push+*>x:
   (* x->BUFFER *) BUFFER,(shift affix block+*,:push;+).
'action'pop+*+x>:
   BUFFER->x,unstack+BUFFER,(shift affix block+*,:pop;+).
```

Passing all remaining affixes in the variable block was used to suppress low level warning messages with a code similar to the one below:

```
'action'warning+>level+T[]+*+>msg:
   level<min level;
   format print+T+*.
```

5 Macro substitution

Calls of a macro rule are implemented by textual substitution. Such a replacement can result in a syntactically wrong text, or in a completely different semantics. The following examples illustrate these cases and explain the additional restrictions a macro rule must satisfy.

1) In a macro formal in affixes cannot be assigned to. Suppose the rule `macro` is defined as

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

then after substitution the replacement is syntactically wrong:

```
macro+1+z            becomes (1->z->1,1->z)
```

2) There is a problem with the dummy affix. Using the same `macro` rule as above, the result has wrong syntax:

```
macro+u+#            becomes (u->#->u, u->#)
```

3) While a macro can have variable number of affixes, neither `shift affix block`, nor `get affix blockno` is allowed. Rule `is zero` below checks if it has an argument with zero value; rule `math` computes the product of its arguments if none 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, it would shifts out all affixes and the computation in `math` cannot be carried over. If the rule `macro` is defined as

```
'function'macro+>a+*+>q: q->b, call+a+b+*.
```

where `b` is some global variable, then after substitution the repeat block can completely vanish so calls of `shift affix block` and `get affix blockno` would give syntax error:

```
macro+1+2+T+rep2     becomes (2->b,call+1+b+2+T+rep2)
```

If a rule head is `XXXX+*+>u` and the same `macro` is called inside this rule, then the formal affix `u` might appear in the substitution text while it has not been passed to the macro explicitly:

```
macro+1+*            becomes (u->b,call+1+b+*)
```

4) Standard selectors are not carried over.

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

where `ptr` is some global variable. After substitution

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

while `S` might not have a standard selector.

5) Out affixes get their values only after returning from the 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`.

Items 1) and 3) are checked during compilation, and error messages are issued if the conditions are violated. For 2), if the actual is a dummy affix, the formal out affix in the macro is turned into a local variable (which, later on, could be optimized out). For 4) the macro substitution mechanism remembers the last substituted formal affix, which gives the correct standard selector. For 5) and other side effect, no warning is, or can be, given, but it changes the semantics. So use macros at your own risk.

6 Redefining assignments and relations

Assignments (transports) and relations (of which identity is an example, see Section 3.3) are handled as a syntactically different way to write a rule call. The assignment

```
 a->b[c]->c
```

is transformed internally to the rule call

```
 @make+a+b[c]+c
```

(recall that the character `@` is a letter in library mode, see Section 3.23), and `@make` is declared in the standard library as a

```
 'external''function'@make+>from+*+to>.
```

Similarly, relations are transformed to calls of rules `@equal`, `@noteq`, `@more`, `@less`, `@mreq`, and `@lseq`, respectively; all of them is a *question* with two input affixes, also defined in the standard library. Any of these rules can be redefined (in library mode) to do something different. As an example, suppose the list `STR` contains strings, and two pointers to `STR` should be equal if the strings they point to are the same, not only if they, as pointers, are equal. So

```
  (ptr1=ptr2,print+"strings are equal";
             print+"strings are not equal")
```

would print `strings are equal` if the two string pointed by `ptr1` and `ptr2`, as strings, are equal. This can be achieved by defining `@equal` to handle this case:

```
 'pragmat'library=on.
 'external''question'real eq+>x+>y="__equal". $ from stdlib $
 'question'@equal+>x+>y-eq:
   (was+STR+x,was+STR+y),compare string+STR+x+STR+y+eq,eq=0;
   real eq+x+y.
 'pragmat'library=off.
```

Actually, the test `eq=0` should be `real eq+eq+0`, as this way the rule `@equal` calls itself. In the module where this definition appears all equality tests use this rule. You might consider to declare `@equal` to be a `macro`; it also should be done in library mode.

7 Implementation details

The target code of this ALEPH compiler is C. It is assumed that the basic ALEPH data type, which translates to `int` in C, is 32 bit long, and that the compiler and the target code runs on the same architecture[1]. There is no assumption on the pointer's size.

[1]The compiler must know how strings are packed in the target machine to generate list fillings. Also, in a datafile declarations the hash of the list identifiers are computed at compile time, and are used when the ALEPH program runs.

7.1 Tables, stacks

The ALEPH value of a table, stack, datafile and charfile is an index to the global integer array `a_DATABLOCK`, where specific structures are stored. Given the value `idx`, the C macros `to_LIST(idx)`, `to_CHFILE(idx)` and `to_DFILE(idx)` create a pointer to the corresponding list, character, or datafile structure.

Table and stack structures have the following fields.

`int *offset;`	the zero virtual element of the list
`int *p;`	pointer to the beginning of the allocated memory block
`int length;`	length of the allocated block
`int alwb,aupb;`	actual lower and upper bounds
`int vlwb,vupb;`	virtual lower and upper bounds
`int calibre;`	calibre of the list

The ALEPH list element affix `L[i]` is translated to `to_LIST(L)->offset[i]` in C. List limits (actual and virtual limits and calibre) are retrieved from this structure. There are no direct pointers to list elements in the program, thus the whole list can be freely moved in the memory, only the pointers `offset` and `p` should be adjusted.

When a stack is to be extended and no more allocated space is available, additional memory is requested (which may move the whole list to somewhere else in the memory). The elements of the new list block are initialized and the list's actual upper bound is increased. The `unstack` and `unstack to` externals adjust the actual upper bound only. The `release` external actually releases the allocated memory, while `scratch` only sets the actual upper bound to its lowest possible value but keeps the allocated memory.

7.2 Datafile implementation

As discussed in Section 2.5, ALEPH datafiles store integers (computer words), and pointers to lists. An external datafile is a sequence of `1024*sizeof(int)` blocks (read and written as a single block), and each block is arranged as follows.

`B[0]`	magic number, identifying the ALEPH datafile
`B[1..31]`	bitmap for the rest of this block
`B[32..1023]`	the actual data

In the bitmap part the indicator bit for the word at position $32 \le$ `i` < 1024 is at word location `B[int(i/32)]`, and at bit position `(i&31)` where zero is the most significant bit, and 31 is the least significant bit. The `nil` pointer is a pointer with value zero; the `eof` (end of file) indicator is a pointer with value -1; all other pointer values must be positive and belong to one of the datafile zones.

The first few values in the datafile contain the zone list. Each zone occupies three words: the virtual lower and upper bounds and the hash value of the list name. The maximal number of zones is `MAXIMAL_AREA=32`, and this list must fit into the first data block. Data for the first zone is in `B[32]`, `B[33]`, `B[34]`, followed by data for the other zones. The lower and upper bounds (inclusive) must be strictly positive and disjoint, and all pointer values, with the exception of `nil` and `eof` must lie within one of the zones. The hash value is a hash of the list name which is used to match the list name when reading the datafile. The hash is generated from the list name by the library function

```
'function'simple hash+T[]+>p+hash>.
```

Input ALEPH datafiles can be positioned. The last 10 bits in the file position identify the index, which must be between 32 and 1023. Other bits identify the block in which the value is. The actual position is stored internally, thus there is no overhead in determining it. File position can be retrieved for both input and output datafiles, but can only set the position for an opened input datafile. No check is made to make sure that the position is valid (can be set after the `eof` indicator).

When opening a datafile for output, the first block with the number of zones and the corresponding values are created; the file pointer is set just before the very first empty space.

Appending to and existing ALEPH datafile is not done as it raises problems. Read the first block, check if it has the same metainformation as we have right now, read the last block, find the `eof` mark, then set the file position just at the `eof` mark.

Opening a datafile for input requires the following operations: read the first block, compare the zones in the input file to the ones supplied by the file declaration. Comparison is made by the hash values: two lists are the same if they have the same hash value. Lists not found in the declaration are skipped; others are copied into a local structure with the difference between the stored and requested pointer values. When the next input is requested, it is checked whether it is a pointer or not. If numerical, pass it. If it's a pointer, check which list it is in, add the difference and pass it as a pointer. Handle `nil` and `eof` separately. If the zone is not found (the corresponding list was not supplied when opening the datafile), then fail and skip this input.

The datafile structure stored in `a_DATABLOCK` has the following fields:

`unsigned fflag;`	different flag bits
`int      fileError;`	last file error
`int      st1,st2;`	string pointers
`int      fhandle;`	handle, zero if not opened
`int      fpos;`	file position
`unsigned iflag;`	pointer/numerical flag
`int inarea,outarea;`	number of areas
`a_AREA  in[MAXIMAL_AREA];`	input list areas
`a_AREA out[MAXIMAL_AREA];`	output list areas
`int      buffer[1024];`	the buffer

7.3 Character files

While datafiles use direct file input and output, character files use streams, namely the `fgetc()` and `fputc()` library procedures without the backchar facility. Input is assumed to be proper UTF-8 encoded, incorrect codes are silently ignored. A single `get char` ALEPH rule may consume up to four bytes from the input stream. There is no `newpage` character, and writing `newline` sends the newline character (code 10) to the stream.

Input character files can be positioned, they use the `ftell` to retrieve the current file position, and `fseek` to set the file position.

The charfile structure in `a_DATABLOCK` has the following fields:

`unsigned`	`fflag;`	different flag bits
`int`	`fileError;`	last file error
`int`	`st1,st2;`	string pointers
`FILE`	`*f;`	stream handle, `NULL` if not opened
`int`	`aheadchar;`	look ahead character

7.4 Strings

ALEPH uses Unicode characters, and they are stored as UTF-8 encoded C strings whose last byte is `\0`. If the string is in list `L` is pointed by the index `si`, then the content of the list block is

`L[si]`	`width` (calibre) of this block
`L[si-1]`	number of UTF-8 encoded characters in the string
`L[s+1-width]`	start of the C string

Each (integer) location contains four bytes. Thus a string with 6 Unicode characters, encoded length 8 is stored in 4 words as follows: (from larger indices to lower ones):

`5`	`width` of this block
`6`	actual string length
`\0 - - -`	third word with the closing zero byte
`c5 c6 c7 c8`	second word, characters `c5`–`c8`
`c1 c2 c3 c4`	first word, characters `c1`–`c4`

The empty string is stored as a block of three zeros.

7.5 Rules in C

Each rule declaration is translated to a C procedure declaration. If the rule is of type *function*, *action*, or *exit*, then the procedure is `void`, if it is *question* or *predicate*, then it is `int`. The compiled C routines return `0` for failure and `1` for success, but when checking the returned value, any non-zero return value is taken for success.

In ALEPH it is the caller's responsibility to store the output value to its destination, and do it only if the called routine reports success. According to this requirement, formal affixes are transformed to c parameters as follows. First assume that the called rule has no variable affix block. Affixes, which are neither out, not inout ones (that is, file, stack, table, or in affix), are passed as integers in their original order. A local integer array is declared for the out and inout affixes, and this array. containing these affixes in their original order, is passed as the last parameter. Before returning, the called routine should supply the output values in this array, which values will then be stored by the caller.

Rules with a variable affix block have two additional parameters: an integer containing the number of blocks (with value at least one), and an integer array containing all affixes in the variable block regardless of their type. The `shift affix block` rule is implemented by decreasing the block counter by one, and adding the block length to this parameter. The following list shows some formal affix sequence and the corresponding C parameter declaration:

```
    +t[]+>i      (int t,int i)
    +""f+o>      (int f,int A[1])
    +>i>+o>      (int A[2])
    +>io>+*+>i   (int A[1],int Cnt,int *V)
```

The called routine must set all out affixes in the output parameter `A[]`, and otherwise it is free to change (and use) these values if the routine fails. Values corresponding to not out or inout affixes in the variable block `V[]` should never change, and the value of an inout affix should be changed only if the routine returns with success.

7.6 Externals

External declarations are allowed in library mode only (Section 3.23). How the string denotation in the external declaration is interpreted depends on the type of the defined tag.

7.6.1 External constant and variable

Both external constants and variables can be used. External constants cannot be used in expressions or any other place where a constant tag is required. In the C code every occurrence of these tags is replaced by the string.

7.6.2 External table and stack

A list structure is reserved in the global integer array `a_DATABLOCK` as explained in Section 7.1. The string in the external declaration is used as a C procedure which is responsible to initialize this structure. It will be called with two arguments: the index of the associated structure and the calibre of the list. The routine must fill the actual and virtual limits. There is a (small) virtual address space put apart for external lists. The first free address is in `a_extlist_virtual`, and can use addresses up to `max int`. The routine should update this value to reflect its reservation. The routine is also responsible to allocate memory and initialize the content of external tables.

7.6.3 External files

The corresponding charfile or datafile structure is reserved in the global array `a_DATABLOCK`. For the description see Sections 7.2 and 7.3. The string in the external declaration is used as a C procedure which is responsible to initialize the structure. The procedure is called with a single argument: the index of the structure.

7.6.4 External rules

How an external rule is handled depends on the string denotation. If the it starts with a letter (either lower or upper case), then the external rule is assumed to be a C procedure with exactly the same parameter passing mechanism as the compiled rules. The compiler generates a prototype with the correct parameter setting; the C routine implementing the external rule can be compiled and linked independently. Several standard library rules are implemented this way.

If the first character in the external rule's string denotation is not a letter, then a simpler calling mechanism is used: all affixes, independently of their types, are given as parameters. These external rules are typically C macros; an example is the `incr+>x>` external rule whose string denotation is `__incr`, thus the ALEPH rule call `incr+ptr` translates to `__incr(ptr)`. The standard header file contains the C macro definition

```
#define __incr(x)   x++
```

which makes the final translation.

The dummy affix translates to nothing, thus leaves an empty parameter location. Using some C preprocessor tricks these empty arguments can be transformed to different C procedure calls. To ease this work, the ALEPH compiler makes some additional work.

If the external rule is a *function* and all of its out arguments are discarded, then this external rule is not called at all.

If the string denotation of the external rule starts with `@`, then this character is replaced by an underscore. For each out argument either a `0` or a `1` character is appended to the rule string depending on whether the out argument is the dummy symbol or not. Finally, from the argument list all dummy symbols are discarded. In the standard library the `divrem` external rule has two out affixes, and `@divrem` as its string denotation. Accordingly, four C calls are generated: `_divrem11` with four parameters when both the quotient and remainder is used, the three parameter `_divrem01` and `_divrem10` when the quotient (remainder) is discarded; and the two parameter `_divrem00` when no result are requested at all.