Aleph Compiler v2.1

L. Csirmaz

Document version 2.1, November 2024

1 Preface

The early 1980's saw a proliferation of computer programming languages. Only a few of them survived and even fewer are in use today. The programming language ALEPH, an acronym for A Language Encouraging Program Hierarchy, almost completely disappeared, and this work is an attempt to resurrect it.

As a programming language ALEPH has many interesting features even by today's standards. Designed by D. Grune, R. Bosch and L. G. L. T. Meertens in the Mathematisch Centrum, Amsterdam [2], its purpose was to offer a language which is "suitable for any problem that suggests top-down analysis (parsers, search algorithms, combinatorial problems, artificial intelligence problems, etc)." ALEPH compilers have been constructed for a wide range of computer architectures (which had a much larger variety at that time), and these compilers generated efficient and succinct code, which was an important requirement those days [3].

ALEPH is a direct descendant of another extinct language, CDL, standing for Compiler Description Language. CDL was designed by C. H. A. Koster [4, 6] as a tool for writing compilers for a wide variety of programming languages and target machines. There had been some more recent work on descendants of CDL [7]. Both ALEPH and CDL belong to the family of the few languages based on affix, or two-level, or van Wijngaarden, grammars [5, 8]. Affix grammars were developed to provide a formal definition of what an ALGOL68 program is [9]. The appealing intuitive meaning of an affix grammar definition combined with the theoretical simplicity and completeness led soon to practical applications. A common feature of those programming languages is that grammatical symbols are interpreted as procedures returning either true or false depending on whether a token sequence derivable from the grammatical symbol has been recognized or not. *Affixes* of the grammatical symbols carry additional contextual information, and behave as (both input and output) parameters of the procedure. Affix values typically come from another, very restricted language.

CDL, and its successor, CDL2 was a popular and widely used compiler writing tool. It is worth noting that the first generation PROLOG compilers were written exclusively in these languages. CDL provides a global logical framework and organizes the data flow among the rules without specifying neither the primitives nor the affix values. While keeping the main design ideas and syntax closely resembling that of the original CDL, ALEPH closed this open endedness by specifying the available data types and fixing the data manipulating primitives.

The unusual *call-then-store* procedure execution mechanism of ALEPH is inherited from CDL. Output parameters (affixes) are local for the called procedure during execution, and are copied back to their destination only after a successful return. Other features unique to ALEPH are modeling the virtual memory as a huge sequence of computer words where stacks and tables occupy consecutive positions whose exact location is outside the control of the programmer; handling character strings as black boxes without direct access to its constituents; and datafiles which allow automatic transfer of stack and table pointers from one program to another. By design, no uninitialized memory location exists in ALEPH, which automatically avoids many hard to discover bugs.

The original version of ALEPH, as defined in the Aleph Manual [2], treats the compiled program as a single stand-alone text—complying to the practice of the time when the language was designed. The present version adds modules by exploiting and expanding the pragmat construct resembling the CDL2 approach. There are several other extensions, changes and restrictions compared to the original specification, hopefully all of them in the spirit of the original design of the language.

In this implementation both the compiler and the linker are written in ALEPH, and the target language is standard C. Both the ALEPH source and the translated C programs are available at

`https://github.com/lcsirmaz/aleph`

Budapest, Hungary
Prague, Czech Republic

Laszlo Csirmaz

Contents

2 A bird's eye overview of Aleph

The ALEPH manual [2] is an excellent introduction to the language and its usage. It is written for novice programmers who have no or little experience. This section contains a concise description focusing on the main differences between ALEPH and modern programming languages.

By design, an ALEPH program can be considered to be a top-down LL(1) parser[1]. ALEPH procedures are called *rules*, and return either *success* or *failure*, implying whether a derived instance of the rule has been recognized (and processed) or not. To enhance the expressive power of context-free parsers, ALEPH rules can be equipped with *affixes*. Affixes carry auxiliary, context sensitive information. The syntax of ALEPH follows the tradition of affix grammars [8] by using the + sign to separate the procedure (rule) name and parameters (affixes).

2.1 Data types

The basic data type of ALEPH is the *word*, which is the storage unit in the target machine. A word can be considered either as a bit sequence interpreted as a signed integer value, or as a pointer which determines a location in the virtual memory. The virtual memory is a sequence of words indexed by words, and is populated partially only by tables and stacks, jointly named *list*s. Tables and stacks occupy disjoint (and far away) segments of the virtual memory. A stack can grow and shrink at its upper end, and shrink at its lower end, while the position and size of tables are fixed and never change. Pointers can only point to a table or stack element (and not to a variable), and a pointer value determines uniquely the list it points into. Elements of a stack can be modified (but not their virtual addresses), while elements of a table are "read only" and do not change.

Consecutive locations in a table or in a stack can be grouped together either to a fixed size *block* or to a *string*. Such a group is pointed to by its last (topmost or rightmost, having the largest address) element. Elements in a block are identified relative to its address by *selectors*. A stack is extended by specifying the elements of a new block using its selectors, and the block is added to the top of the stack, extending its actual range.

Strings are also stored at consecutive locations of a list, and behave like black boxes and can be manipulated by predefined routines only, see Section 2.3. This way ALEPH does not determine how characters in a string are stored, allowing compressed storage of a large character set. Strings can be unpacked to a list of characters, and a list of characters can be packed into a string which is pushed to the top of a stack.

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

Finally, for communicating with the outside world, ALEPH distinguishes character and data files. Files can be opened and closed, character files can send and receive characters. In a data file each entry is marked as either a word or as a pointer to one of the lists associated with the file. While reading, pointer values are automatically adjusted, which allows an automatic transfer of pointers from one ALEPH program to another.

2.2 Rules and calling a rule

In ALEPH parlance procedures are rules, and its parameters are affixes. When writing a call affixes are added to the called rule identifier using + signs as in

```
rule + affix1 + stack + file + -42.
```

Each actual affix is either a word (a constant, a variable, or an indexed list element), a list (stack or table), or a file. No compound affixes (e.g., expressions) are allowed. A rule either returns a logical value or returns nothing, while it can have several "out" affixes for returning computed values. The required affix types are specified in the head of the rule declaration as

```
rule + >affix1> + []stack[]+ ""file + >affix2
```

The first affix `affix1` both receives and returns a word value; the second affix is a stack, the third one is a file, finally `affix2` receives a word value but does not return anything (and the rule body can use `affix2` as a local variable).

The control flow in the rule body is quite restricted: it is a sequence of alternatives probed in the order of their presence. An alternative is a sequence of members guarded by its first member. If this first member succeeds, the alternative is chosen and the remaining members are executed; otherwise the next alternative is probed. Jumps are allowed only as an abbreviation for tail recursion. There are no repetitive statements at all, iteration must be handled by recursion. Next to rule calls a member can be a compound member or an extension. The compound member is an (implicit) rule definition enclosed in parentheses, while an extension adds a block of specified number of values to the top of a stack.

Due to its simplicity, the control flow inside a rule is tractable. Liveliness and reachability properties can be checked statically during compilation. In particular, the ALEPH compiler checks statically that in a rule

- all members are reachable;
- the flow can always reach a return point;
- when a local variable or parameter is used it has an assigned value (it is not "uninitialized");
- an out parameter has been assigned a value along all paths ending in a return point;
- if a local variable (or parameter) has been assigned a value, it is actually used.

2.3 Externals

Basic data manipulation, such as addition, comparison of words (as integers), and similar operations are done by standard external rules. As an example, `incr+x` increases the value of its argument by one; `equal+x+y` tests for equality, returning *success* if `x` and `y` are equal, and *failure* otherwise. Some externals have equivalent syntactic variants, such as the assignment (called transport in ALEPH) `make+src+dest`, which can also be written as `src->dest`, or `equal+x+y`, which can also be written as `x=y`.

All file operations are also done by externals. Standard external are used for some stack operations (such as shrinking), and for manipulating strings. Externals can be redefined; this feature allows to mimic overloading the basic operators, for an example see Section 5.6.

3 Prototypes, modules and libraries

The ALEPH Manual [2], the official definition of the language, has several revisions. All of them were published by the Mathematisch Centrum, Amsterdam, between 1974 and 1982. Implementations of a computer language frequently add new features, while restrict or leave out others. This happened with ALEPH as well; changes in the ALEPH manual reflect the evolution as new applications and new compilers appreared. The present implementation, and usage, of the ALEPH language is no exception. Making modular programming possible—as opposed to monolith programs prevalent at the time of the inception of ALEPH—required new features. Other extensions and restrictions came naturally; some of them date back to the time of the first ALEPH compilers. All changes made in this implementation hopefully respect the original design ideas and philosophy as described in [3]. For a more detailed description of the changes and new features see Section 4.

3.1 Prototypes

The original ALEPH specification has no prototypes, but prototypes are indispensable when the program is split into smaller modules which are compiled independently. Each module should have complete information on all program constructs (rules, variables, lists, etc.) it imports, and should provide that information on constructs it exports. Prototypes, the stripped down declaration heads, are just the right constructs for this purpose.

3.2 Modules

An ALEPH module provides certain resources to the main program and to other modules, and is compiled independently. A module (or the main program) can *require* resources provided by other modules, and a module can *provide* resources defined within it. Accordingly, an ALEPH module is split into a *public* and a *private* part. The public part specifies the provided items using prototypes, while the private part contains the realization of those resources. When the module is required, only the public part is scanned. When the module is compiled, both the public and private parts are processed, thus the compiler can check that all items this module promises to export are indeed provided by its private part.

The public part of a module may contain genuine declarations next to the list of export prototypes, and may require additional modules. These public declarations are compiled into the invoking program locally.

If an ALEPH module, say `modA`, requests another module, say `modB`, then the terminology "`modB` is directly visible from `modA`", or "`modB` is immediately above `modA`" is used. The module `modB` is *above* `modA` if there is such a visibility chain from `modA` to `modB`. Resources provided by a module are available to every other module which are below it, except for those resources which are redefined by some intermediate module. The original resource can still be reached using *qualifiers*. In general, the same resource can be provided by several visible modules, in which case the module with the smallest rank (that is, the smallest number of hops in immediate visibility) is chosen as the provider.

Requested modules are processed only once, thus no special measures should be taken when the visibility is circular. This is the case, for example, when module `modA` requires `modB`, and module `modB` requires `modA`. For a more detailed description on ALEPH modules see Section 5.

3.3 User libraries

An ALEPH module can be designated as a *user library*. Resources provided by such a library module are available to all non-library modules, but only as a last resort if no other definition was found.

In general, resources provided by a plain module `modA` are not automatically avaiable for all other modules: `modA` must be required explicitly (possibly through a request chain). In contrast, designating `modA` as a user library, any module can use resources provided by `modA` without requesting it. A user library can request other modules; these requests determine another visibility structure which is independent of, and above, the visibility structure of the plain modules. Resources provided by the collection of user libraries are those provided by the lowest rank modules in the library visibility structure.

There is another library layer above the user libraries starting with a single *standard* ALEPH *library* module. This extra layer provides the implementation of basic ALEPH primitives as required by the Manual [2]. Consequently, a user library module can use (and redefine transparently if it chooses so) these primitives without any further arrangements.

4 Enhancements and changes

This extensive section describes the main changes between the implemented language and the language specified in the ALEPH manual [2]. The description assumes a basic level familiarity with the ALEPH language.

4.1 Program text representation

ALEPH is an Algol-like language [9] in which keywords are distinguished by a different typeface. Practical coding, however, uses a single (monospace) typeface. There are several approaches to distinguish keywords from the surrounding text but none of them is perfect. This implementation requires keywords to be enclosed between apostrophe characters such as in this example:

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

Other possibilities are: using capital letters for keywords (as in several PASCAL implementations); restricting keywords (as in C and related languages); using an initial escape character and whitespace at the end (e.g., leaving out the closing apostrophe); and so on. This choice reflects the influence of ALGOL 68 [9] on ALEPH and on its relatives.

Characters in ALEPH are not necessarily restricted to single byte ones as strings cannot be manipulated directly. This implementation allows any unicode character as a string character. According to the ALEPH Manual `newline` and `newpage` are not characters. This implementation relaxes this restriction. A newline character (with code `10`) can be part of a string, but not of a string denotation (a string appearing in the program text). In the program text all strings should be closed in the same line they start. Two separate consecutive strings (even if they are on different lines) are concatenated, thus strings can be continued on the next line, but cannot contain newline characters. There are no escape characters either, and presenting a quotation mark in the string it should be doubled forming a quote image. Thus

`"a" "b"`	is a concatenation and has two characters: `/a/` and `/b/`;
`"a""b"`	has three characters: `/a/`, `/"/` and `/b/`.

A character written between slash characters represents its unicode value, in particular the code for the slash character is written as `///`.

4.2 Hexadecimal constants

Wherever an integer denotation is accepted, hexadecimal constants are recognized and accepted as well. An example is `0x1234abcd`. The minus sign can also appear before a hexadecimal constant: `-0xffff`.

4.3 Relations

The external rule `equal+x+y` tests for equality of `x` and `y`. The same test can be written equivalently as the relation `x=y`. Similar shorthands are added for other comparison operators:

`x!=y` and `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`.

4.4 Lists in scalar context

In an actual affix position where a single word is required, a list `L` (a table or stack) can also appear with the meaning that the value it represents is that of its topmost (rightmost) element, namely, `L[>>L]`. The same abbreviation can be used with selectors. Thus the rule call

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

is equivalent to

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

Using this extension a stack looks—and behaves—like a variable, which makes writing and comprehending stack operations easier. This extension comes handy when the content of a newly added block on the top of a stack is to be manipulated. The drawback is that it prevents the compiler reporting a parameter type mismatch when the list is used by mistake. This extension causes, quite unexpectedly, extra complications during macro substitution, see Section 4.26 for details.

4.5 Dummy affix

When the value of an out formal affix is not needed (the value is thrown away), rather than forcing the programmer to invent some dummy variable, the dummy symbol can be used with the meaning that the returned value will not be used. This implementation encourages the character `#` for this purpose, while the official representation `?` is also accepted.

4.6 String as actual affix

A string denotation can be used as an actual affix. This extension simplifies writing program texts as the actual string can appear where it is used. Without this feature strings are to be put into a table with a pointer constant pointing to them, and then the string is identified by the table name and the string pointer together:

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

With this extension the string can appear directly as an actual affix:

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

The string denotation translates into two affixes: first, a special internal table, and second, a pointer which points to the string in that table.

4.7 Manifest constants

Manifest constants start and end with an underscore, and are replaced either by an integer or by a string while scanning the source text. In particular, they are replaced before macro substitution, thus `_line_` and `_rule_` in a macro text reflects the source line of the macro definition and the macro name, and not those of the invoking rule. Similarly, `_title_` and `_module_` expands to the empty string if they appear before the corresponding `title` or `module` pragmat.

`_line_`	integer, the source line number this constant appears in
`_file_`	string, 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 if used inside a rule definition.

4.8 Extension syntax

An extension adds a new block to the top of a (formal or global) stack. An extension is specified as a sequence of assignments where the destinations are selectors of the block to be added; this list is enclosed between `*` symbols and 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 the extensions visually more appealing, parentheses can be inserted as follows:

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

Accepting both extension forms destroys the LL(1) property of the ALEPH syntax. It is so 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. This compiler solves this problem by doing a reasonable amount of look-ahead.

4.9 Variable number of affixes

Allowing and handling a variable number of rule affixes is one of the main achievements of this implementation. Variable number of affixes raises several problems, and poses a potential obstacle for control flow tractability. The used approach meets the following two main requirements: it provides a flexible and usable variable argument mechanism, while keeping all tractability properties of an ALEPH program. This is achieved by restricting how the "invisible" affixes in a variable affix block can be accessed.

A formal affix sequence, which defines the arguments of a rule, may contain the anchor symbol `@` indicating the position from which point affixes to the right end of the list can be repeated indefinitely in a call to this rule. The declaration of `sum` in the snippet

```
'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 forming the `repeat 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` in this order. The rule body starts by setting `a` to zero. The `add` rules after the label `nxt` add `b` and `c` to `a`. The built-in routine `shift affix block` shifts out the visible affix block at `b` and `c` and moves the next block into this place, assuming that there is still a pending, unseen block. If there is none, the rule `shift affix block` fails, and `sum` ends. Otherwise a jump is made to the label `nxt`, where the next group of two input affixes are added to `a`. In summary, `sum` adds up all of its input arguments and returns their sum in `a`.

The main point which guarantees the tractability of the data flow is that shifted out blocks are lost to this rule, and there is no way to reach the affixes there.

In a rule call the anchor symbol `@` can appear only as the last actual affix with the meaning that the present (visible) and all subsequent (pending) affix blocks of the calling rule are passed as arguments. The rule `negate` defined as

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

adds up the first two of its input affixes `b` and `c`; if there are no more affixes, then this sum is the result. Otherwise, it calls `sum` to add up the value of the rest affixes, 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 built-in routine `get affix blockno+n>+@` returns the number of pending affix blocks. This number is always positive and it is `1` if and only if `shift affix block+@` would fail.

The main application of variable number of arguments is formatted printing. In ALEPH the compiler this feature has been used mainly to format error messages, but it also turned out to be useful in code generation. Rudimentary formatted printing can start with the format string passed by two affixes: the table and the string pointer. Format characters starting with `%` require 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`, as is done in the program snippet

```
'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 could be pushing and popping an unspecified number of elements to and from a stack. We remark that the rule pop below works correctly only when the stack has calibre (block size) one, as unstack+st discards a complete block and not a single element from the stack, see Section 4.17. The assignment st->x stores the topmost element of the stack st in x as discussed in Section 4.4.

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

Passing all affixes in the variable block can be used, e.g., to suppress low-level warning messages with a code similar to the one below:

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

4.10 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;  $ either zero or one
 [-10:10],action 2;  $ range from -10 to 10
 [L;1000],action 3;  $ either in L or equal to 1000
 [:],      action 4)  $ everything else
```

The area in the square brackets determines whether the alternative following it is chosen or not. An area may contain integer denotations (decimal or hexadecimal), constant tags (including constant pointers) and global (not formal) lists; no expressions are allowed. A global list stands for its complete virtual address range. All values within the area are determined during compilation. It is an error if some of the alternatives cannot be reached (this would happen if the first two areas in the above example are swapped); and the compiler gives a warning if it could happen that none of the alternatives are chosen. When running the program and none of the alternatives succeeds the program aborts with an error message. An "otherwise" case must be specified explicitly as in the example above.

4.11 Expressions

In ALEPH all expressions are evaluated during compilation. Originally expressions could 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 value whenever necessary.

An expression evaluates to a constant value. It may contain constant tags declared later (or even in another module), but cannot depend on itself. Thus

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

is accepted where /a/ is the value of character 'a' in the used coding, while

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

gives an error message as the value of p depends on itself. In addition to the usual arithmetic operators +, -, *, and / the following Boolean operators can also 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.

The Boolean operators have lower priorities than the arithmetic ones.

In an expression integer denotations (both decimal and hexadecimal), character denotations (a character between / symbols), constant tags, pointer constants (defined in fillings), virtual bounds and block size (calibre) can be used. List size estimates and repeat numbers (see Section 4.18) are evaluated before the virtual bounds are determined, thus these values cannot depend on virtual bounds.

4.12 Root rule

The only executable command of an ALEPH program is its root. It can have local affixes and a rule body. As the root is executed only once there is no need to designate a separate rule for this purpose. Example:

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

Roots of modules are executed before the root of the main program (see Section 5), they can perform all necessary initialization for the module. Modules which do not require initialization should use an empty root:

```
'root'+.
```

To control the order of module initializations, a module can call

```
wait for+"xxx"
```

to force the root of the indicated module xxx to terminate before the call returns. The wait for rule aborts with an error message if two modules would wait for each other producing a deadlock. The wait for rule requires the *name* of the module (see Section 5) specified as a string, and not the name of the file containing the module text. If no module was linked with the given name, the rule returns immediately. If there are more modules with the same name, all those module roots are called.

4.13 Actual and virtual limits

The complete *virtual memory space*—the allowed range of indices—is distributed among the tables and stacks with almost no control of the programmer. These virtual bounds are fixed and hard-coded into the compiled program. 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 typically does not occupy its virtual space completely. Existing locations (which correspond to locations in the machine memory) form a presumably much smaller continuous *actual memory space*. 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.

Thus the *actual memory space* of a stack changes when the stack is extended or shrunken For a list `L` the constructs `<<L` and `>>L` return the actual lower and actual upper bound of `L`, respectively. To obtain the fixed *virtual* limits of the same list, use `<L` and `>L` (with a single left symbol and right symbol). In expressions only the fixed virtual limits can be used as only these are available during compilation.

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 with an exact size estimate or no size estimate) have equal actual and virtual limits.

4.14 Size estimate

The size estimate in a stack declaration specifies how much virtual address space this stack requires. The estimate is given between square brackets, and can be *fixed*, *relative*, or *empty*. In the first two cases the limit must be either an integer denotation or a constant tag; no expression is allowed.

- Fixed size is written between = symbols and the value cannot be larger than 1,000,000 (and, of course, must be positive). The compiler reserves at least that much virtual space for the stack. (The final virtual space can be larger if the stack has fillings which total to a larger amount.)
- A relative estimate should yield an integer between 1 and 100. After reserving virtual addresses for tables and fixed size stacks, the remaining virtual space is distributed proportionally to the requested relative amount.
- If size estimate is left empty, the stack size (both virtual and actual) is determined by the amount of its fillings (see Section 4.18). Such a stack can still shrink, but cannot expand beyond its virtual upper limit.

4.15 Table declarers

To distinguish table declarations from prototypes and fillings (see Sections 4.22 and 4.18, respectively), a table declaration must contain an empty size estimate:

```
'table'[](length,width)TBL,[]pi=(3,1,4,1,5,9,2,6).
```

4.16 List selectors

Every table and stack has an associated *block structure* which determines the block size, called calibre, of that list, together with the set of its selectors. When no selectors are specified, the block size is 1 and the selector of that element is the name of the list – the *standard selector*. In general, this standard selector is used implicitly when the list is indexed without specifying a selector. If the list definition has a selector list, then this list contains the block selectors in a left to right order. The formal stack definition

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

specifies that the block structure of the formal affix `tags` has three elements with selectors `tag`, `left`, and `right` in this order. Also, the list `tags` has no standard selector, namely a selector with the name of the list. The selector list, if present, cannot be empty, meaning it must contain at least one selector.

The same block element can be identified by different selectors. These additional selector names are specified after the initial selector separated by an equal sign as in the stack declaration:

```
'stack'[1](s1,s2=t1,s3=t2)stack.
```

To emphasize which selectors are used together, multiple selector packs are accepted. Each pack must have the same number of selectors, the dummy symbol `#` can be used as a placeholder. Thus the previous declaration can also be written as

```
'stack'[1](s1,s2,s3)(#,t1,t2)stack.
```

If a list was defined with selectors, then a matching prototype must also contain a selector list pack, while a filling may omit it, see Section 4.18. All selector packs must define the same block size and the same standard selector, but could define different selector names which accumulate: any of the specified selectors can be used.

4.17 Matching formal and actual lists

The block sizes of the formal and the actual lists are compared as follows.

- The formal list has no selectors.
  There is no restriction on the block size of the actual list. Observe, however, that in this case the standard selector of the formal and the actual list 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. With the declaration `'stack'[1](L,b)L` the assignment `0->L` clears the second to last element of `L`, while `set zero+L` clears its last (topmost) element.
- The formal list has a selector pack.
  The actual list must have the same block size and the same standard selector, while selector names might be different. If this restriction is violated, a warning is issued; if the called rule is a macro, then this is an error.

4.18 Filling

In addition to specifying the size estimate and the optional selectors, a table or stack declaration may also define the initial content of the list. This content can also be specified separately using *fillings*. Fillings 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 final order is unspecified, but within a single filling the order of the added elements is kept intact. The list description in the filling may contain a selector pack only if the corresponding list is defined with selectors. If there is a selector pack, both the block size and the standard selector must be the same as in the definition; the selector names, however, can be different.

The filling itself is enclosed in parentheses, and is a sequence of integer denotations, constant tags (including constant pointers), strings, and blocks; no expression is allowed. Each item, except for strings, can be followed by the repeat symbol `*` and either an integer or a constant tag specifying how many times this item should be repeated. Then the optional pointer initialization sequence follows: a colon and a tag which is defined to have the value of the virtual address of the lastly defined list item. The pointer initialization can be repeated. In the example

```
'constant'tsize=10.
'stack'T=(0*tsize:tzero:tz1,1*tsize,"string":tstring).
```

the filling adds ten zeroes, ten ones, followed by the internal representation of the string `"string"` to `T`. It also declares both `tzero` and `tz1` to be the (virtual) address of the lastly added `0`, and `tstring` to be the (virtual) address of the last element of the representation of `"string"` (which, if no further filling is added to `T`, 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 block size (calibre) of the list; violating this requirement results in a warning. A block element must be either an integer denotation or a constant tag (possibly a pointer constant), but not a string. In the block the constant value is followed by an arrow symbol `->` and the selector where it will be stored. The filling in the example

```
'stack'[1](ch,p)optor=
     ( (/+/->ch,3->p),(3->p,/-/->ch),(5->p,/^/->ch) ).
```

adds three blocks of size two each to the stack `optor`. One of the selectors can be replaced by the repeat symbol `*` to mean that the value is copied to all selectors not mentioned explicitly in this block.

The original block syntax is also accepted: the compound block of the filling contains, in left to right order, the values (an integer denotation or a constant tag) which should be added to the list. One of the 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=( (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)`.

4.19 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 it becomes `exit+16`. Consequently `'exit'` must be followed by an actual affix, and not by a constant expression.

In general, next to the four rule types *predicate*, *question*, *action*, and *function* specified by the ALEPH Manual [2], a fifth one is added: *exit*. A rule is of type *exit* if it never returns. The external rule `exit` is of type *exit*, as well as the rule `error` defined below which prints some additional message before terminating the program:

```
'exit'error+>x:
   x>=0,exit+0;
   put string+STDERR+"Exit level ",put int+STDERR+x,exit+1.
```

An exit rule cannot have out or inout affixes as there is no way to use the returned value. When an exit rule is defined, this condition is checked. When such a rule is used, it is treated as a terminator which can neither succeed nor fail. An exit rule has an implicit side effect (aborts the program), thus it cannot be used in functions and questions. Violating this restriction gives a warning message.

4.20 File area, file string

ALEPH distinguishes two file types: character and data. Character files accept and write characters; in this version the used character set consists of Unicode characters. During character transput there is an automatic conversion from and to UTF-8 encoding. The ALEPH program receives and sends Unicode characters.

Data files communicate between different ALEPH programs. Data files are written and read one item a time; an item is either an integer (word) or a pointer. The data file does not store pointer values directly, rather a pair consisting of an indication of the list the pointer points to and the relative address of the pointed item in that list. From this information the pointer can be restored independently of the virtual address distribution. A datafile declaration specifies all lists whose pointers can be transmitted. By storing the virtual limits of these lists in the datafile first, each additional item requires a single extra bit only specifying whether the item is a pointer or not. When opening an ALEPH data file for reading, stored limits are paired with the lists in the file area so that the appropriate pointer transformation can be made.

According to the ALEPH manual, a file declaration can have an area which restricts what values are allowed to send to or receive from that file. This implementation does not allow areas for character files, and the area of a datafile may contain only lists to which pointers are sent to or received from. The order of the lists is significant: when reading from a data file the first list in the area is matched to the first list when the file was written, and so on.

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

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

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

Without explicitly opening the file, the first file operation tries to open it. The string denotation in the file declaration gives the file name (with possible path information), 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 path string. A file cannot be opened for reading and writing simultaneously.

For a character file the following special filenames identify the standard streams: <<stdin>>, <<stdout>>, and <<stderr>>. Also, the character files STDIN and STDOUT are declared in the standard ALEPH library. They can be used, without opening, to read from the standard input, and write to the standard output, respectively.

4.21 Static stack and static variable

Variables and stacks can be declared to be static by adding the **'static'** keyword before their declaration. Examples:

```
'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.

4.22 Prototype

A prototype informs the compiler about a type of an identifier. 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. Prototypes are similar to external declarations without the **'external'** keyword and the string denotation. Examples:

```
'charfile'STDIN.
'action'print tag+>tag,read tag+tag>.
'constant'max tag pointer.
'stack'(#,#)STACK.
```

Prototypes are also used to inform the compiler about tags which are defined in other modules, and about tags which should be exported from this module. See Section 5 for more information.

4.23 Pragmats

Pragmats control different aspects of the compilation. Their semantics changed significantly compared to the ALEPH Manual. This implementation recognizes the following pragmats:

`tab width=8`	sets tab size for program text printing
`list=on/off`	switch program text printing
`right margin=120`	right margin for program text printing
`dictionary=on/off`	collect tag occurrences
`warning level=4`	set warning level between 0 and 9
`error="message"`	issue an error with the given message
`warning="message"`	issue a warning at level 9
`bounds=on/off`	compile with or without index checking
`count=on/off`	profiling: count how many times a rule is called
`trace=on/off`	trace rule calls
`macro=rule`	`rule` should be treated as a macro.
`std library=off`	don't process the standard ALEPH library
`define=tag`	mark `tag` as defined for an `ifdef` pragmat
`library mode=on/off`	switch library mode
`prototype=none`	specify how prototypes are handled (see Section 5)
`title="title"`	specify program title
`module=tag`	specify module name and namespace
`include="file"`	add `file` to the sources to be read
`require="file"`	require module definitions from `file`
`library="file"`	add `file` as a user library module
`front matter="code"`	insert `code` to the front of the generated code
`back matter="code"`	insert `code` to the end of the generated code

There are additional pragmats which cannot be manipulated in the program text. The most notable one is `compile`, which can be either `on` or `off`. Some pragmat values can be interrogated by conditional pragmats, see Section 4.24. Command-line arguments starting with a double dash, such as `--XX=YYYY` are parsed as

```
'pragmat'XX=YYYY.
```

except that no conditional pragmats (Section 4.24) are accepted. There are other command-line pragmat shorthands starting with a single dash:

`-l`	`list=on`
`-d`	`dictionary=on`
`-W`	`warning level=3`
`-Wall`	`warning level=0`
`-D TAG`	`define=TAG`
`-m XXXX`	`require="XXXX"`
`-y XXXX`	`library="XXXX"`
`-o XXXX`	specify the output file
`-I XXXX`	search directories
`-L XXXX`	standard library directory

The `-o` option specifies the name of the generated `.ice` file. If missing, the `.ice` file name is derived from the first source file, and generated in the current directory. The `-m` option marks the following file to be processed as if it were required; the `-y` option marks the file to be processed as a user library. The `-I` option specifies the list of search directories for included files, requested modules and user library modules. Finally the `-L` option specifies where the compiler should look for the standard library files.

The default value of some of the pragmats is the following:

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

Pragmats `front matter="code"` and `back matter="code"` are accepted in library mode only; the specified string is copied verbatim to the front (to the back, respectively) of the generated code with the exception that the sequence `%n` is replaced by a newline. The pragmat `std library=off` inhibits processing of the standard library, while `library="file"` designates the module `file` as a library module (which can request other modules). The latter two pragmats are not accepted in required or library files.

The `prototype` pragmat has four possible values: `import`, `public`, `none`, and `reverse`. In the first case a `prototype` indicates that the tag has a declaration outside this source (and then it cannot be defined, but can have other prototypes). In the second case a tag appearing in a `prototype` automatically gets the *public* flag, and must be defined in this source (in particular, it cannot be imported, and this source 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. For a sample usage of `reverse` see Section 5.5.

4.24 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` in `if` and `ifnot` pragmats is one of `compile`, `list`, `dictionary`, `module`, `library mode`, etc. 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 (tag), and the compiler checks if this identifier has (has not) been defined until this point by a declaration, an import prototype, or by a `define` pragmat. As an example,

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

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 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.

4.25 Library mode

Pragmats `library mode=on` and `library mode=off` turn the library mode on and off, respectively. This mode determines whether library extensions are allowed or not.

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. Pragmats `front matter` and `back matter` can only be issued in library mode.

4.26 Macro substitution

To improve efficiency rule calls can be implemented by textual substitution. When the rule name appears in a `macro` pragmat (Section 4.23), calls to this rule in the current compilation unit are replaced by the rule body (and replacing formal affixes by the actual ones). Such a 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.

1) In a macro, formal in affixes 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)`

2) There is a problem with the dummy affix `#`. Using the same `macro` rule as above, the following substitution has incorrect syntax:

   `macro+u+#` ⇒ `(u->#->u, u->#)`

3) 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. Rule `is zero` below checks whether one of its arguments has value zero; 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 can vanish completely causing a syntax error:

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

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            ⇒ (S[ptr]->z)
```

while `S` might not have a standard selector.

5) 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 1) and 3) are checked during compilation, and error messages are issued if the conditions are violated. For 2), 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 4) the macro substitution mechanism remembers the last substituted formal affix and uses the same value for the standard selector as the rule call would do. For 5) and other side effects, no warning is, or can be, given, but substitution clearly changes the semantics. So use macros with care.

4.27 Debugging tools

Debugging tools are added at the linking stage of the compilation when macro rules have already been substituted. Consequently these tools have no effect on macro rules.

Index checking. It is controlled by the `bounds=in/off` pragmat. If the pragmat is `on` when a rule is declared, the rule is compiled with index checking. Before touching the indexed element, the value of the index is checked to be within the actual limits of the indexed `stack` or `table`. If this is not the case, the name of the list and the rule in which the error occurs are printed to `stderr`, and the program is aborted.

Profiling. The `count=on` pragmat adds instructions which count how many times the rule is called. After a normal program termination the name of profiled rules and the number of calls to them are printed to `stderr`. The list starts with the largest calling numbers, and ends with rules which were not called at all.

Tracing. If the pragmat `trace=on` is in effect when a rule is declared, the rule is compiled with instructions that print out to `stderr` the rule name, followed by the values of its in and inout affixes. As this would produce a huge volume output, tracing information is printed only when the compiled binary is executed with the `-T` switch as the first command-line argument. Without this switch only the lastly executed 30 rulenames (without arguments) are printed when the program terminates (normally or with an error).

Call stack. Tracing provides the lastly executed 30 rule calls. Frequently the *call stack*, the hierarchy of the pending rule calls gives more information. When adding the `-g` switch to the linker it produces an ALEPH binary which keeps track of the call stack, and prints it to `stderr` when the program terminates either normally or with an error. This feature adds some overhead to all rule calls, and increases the size of the compiled program. The library routine

```
'action'backtrack.
```

prints out the complete, actual call stack starting with the rule containing this rule. In the case the program is *not* linked with the `-g` switch, `backtrack` behaves as a no-op and does nothing.

5 Modular Aleph

A module is an incomplete program part, which can be compiled independently, and which provides resources to be used by the other parts of the program. Modules are frequently written so that it can be reused by other programs. The first non-comment line in an ALEPH module is usually a pragmat specifying both the *name* and the *namespace* of the module:

```
'pragmat'module=XXX.
```

The module name `XXX` must be a valid ALEPH identifier; it must be written without quotation marks. According to the best practice, the name of the file containing the module and the module name should resemble each other. It is particularly important in ALEPH as ALEPH modules are invoked by specifying file names, and not module names. While not recommended, different modules (that is, modules in different files) can share the same module name, and, consequently, the namespace.

In the module the initial `module` pragmat is followed by the public part, or *head* of the module. It contains the prototypes of the (public) tags exported by, and defined in, this module. Next to the public part is the private part, called *body*, which defines the exported items together with the optional auxiliary, unexported items. The body is enclosed between `ifdef=compile` and `endif=compile` pragmats.

An ALEPH module is invoked, or required, by a `require="FFF"` pragmat specifying the file name (possibly with path information, enclosed within quotation marks) of the module. When required only the public part—the module head—is processed from the module text. Prototypes in the head are considered to be external definitions which define items to be imported from the defining module. In contrast, when the module is compiled, both the head and the body are processed. During module compilation the prototypes in the head specify the items to be exported, and the compiler checks that those items are indeed defined correctly in the module body.

The following example illustrates this mechanism for a sample module which exports two items: the rule `do something` and the stack `LEXT`. The module name is `sample` and it is stored in the file `"fsample"`.

```
'pragmat'module=sample.        $ module name
'action'do something+>in+out>. $ prototype
'stack'(adm,left,right)LEXT.   $ prototype
'pragmat'if=compile.           $ module body starts here
 '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.        $ end of module body
'end'
```

When the module is compiled, the file is read with the initial implicit instruction `'pragmat'compile=on`. The `module=sample` pragmat in the first line defines both the name and the namespace of the module to be `sample`, and it also sets the `prototype` pragmat to `public`. Next the module head is parsed where the compiler marks all prototyped tags to be exported (as instructed so by the current setting of `prototype`, see Section 4.22), which also means that the prototyped tags must have a definition somewhere in the module body. The condition in the `if=compile` pragmat holds, thus the material in the module body is processed: the `stack` and `rule` declarations are parsed and executed. The empty root rule (Section 4.12) indicates that the module does not require any initialization. At the end of the module items to be exported are checked against the prototypes and are prepared for export using the module namespace.

When the main program or another module wants to use any of the resources provided by this module, it *requires* it by issuing

```
'pragmat'require="fsample".
```

specifying the file name (between quotation marks) of the module text. Before reading the `"fsample"` file, an initial implicit `compile=off` pragmat is executed. The first line in the module text sets again the name and the namespace (for the duration of `fsample`), but sets the `prototype` differently, to `import`. Subsequent prototypes are scanned and marked as "to be imported". The compiler will use this information to check that these items are used properly in the rest of the invoking program. Reaching `if=compile` the condition fails, therefore the remaining part of the module, the body, is skipped.

A module body can require public items from other modules. It may happen, without any problem, that the body of module `A` requires module `B`, while the body of module `B` requires module `A`.

Next to prototypes the public part of a module may also 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 using the module's namespace.

5.1 The module hierarchy

An ALEPH compilation unit (the main program or a 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 `B` is *above* `A` if there is a "required" chain from `A` to `B`, see also the discussion in Section 3.2. Resources provided by module `B` are automatically available for every unit `A` below `B` in this hierarchy, that is, those units `A` for which `B` is above `A`.

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 equal to or above `A`, and has the *smallest rank*, that is, which requires the smallest number of "required" hops to reach from `A`. By default, `A` has rank zero above itself.

The same module can be required by different modules many times, when this module appears in the hierarchy at several places. Nevertheless, it is still processed only once. The modules added as *user library* (Section 3.3) using either the command-line argument `-y` or the `library` pragmat, and the modules required recursively by these library modules, form a second hierarchy. Elements of this hierarchy are *above* the 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 ALEPH library providing the realization of ALEPH primitives.

5.2 Finding tag definitions

Each `tag` denoting an ALEPH construct 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 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` `x` occurs in `A`, where `A` is either a module or the main program, and `x` 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* (which has the minimal number of "required" hops from the main program, see Section 5.1) 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`. Naturally, this search must yield a unique definition.

This procedure is illustrated in the picture below. Modules in files `"f1"` and `"f2"` set their name to `t`, `"f2"` is required by the module in `"f1"`, and `"f1"` is required by the main program. Tags (with the indicated qualifier) in the "define" column are defined in the module head. Tags in the "use" column are connected to their identified definitions.

file	require	module	define	use
`"f2"`		`t`	`b v`	`a v x`
`"f1"`	`"f2"`	`t`	`a v`	`a x`
main	`"f1"`		`a t::x x`	`a b t::b v x`

The `tag` `a` both in the module from file `"f1"` and from file `"f2"` has the implicit qualifier `t`, therefore it does not match the definition of `a` in the main program. The `tag` `x` in these modules matches the definition of `t::x` in the main program, but not the definition of `x` (without qualifier) there. Both `b` and `t::b` in the main program identifies the single definition of `b` in `"f2"`. Observe that the definition of `v` in `"f2"` is not used at all.

5.3 Requiring and including source files

Source files are handled one file at a time. They are read, processed and closed before opening the next source. Source files can be specified on the command line, required by a `require` or `library` pragmat (or by the equivalent command-line options, see Section 4.23), finally can be included by an `include` pragmat.

The pragmat `require="file"` places the source `file` into the module hierarchy (Section 5.1), and then queues the file to the end of files to be processed as a *module*. This second step is ignored if the same filename is already in the queue, causing each source file to be processed once. Library modules (including the standard library) are handled similarly, but their processing starts only after all non-library sources have been completed. A source file added by a `require` pragmat in a library module is treated as a library.

Using the pragmat `include="file"`, this `file` is *always* appended to the end of the source list, together with the `prototype` and `compile` pragmat values and the module status (is it a module, and if yes, which one) of the invoking source. In contrast to modules and libraries, included sources are processed as many times as they are specified.

As discussed at the beginning of Section 5, when a source is processed as a required module, an implicit `compile=off` pragmat is executed before reading its first character. When the source is to be compiled (specified the file name in the command line), an implicit `compile=on` pragmat is executed before processing. The effect of a `module=XXX` pragmat depends on whether `compile` is on or off. If `compile=on`, then the `module` pragmat switches to module compilation and sets `prototype=public`. If `compile=off`, then it reads a module head, and sets `prototype=import`.

5.4 Using the namespace

A module can have "call-back" rules whose definition must be provided by the invoking program. An example could be a module which provides a sorting algorithm without defining the rule which compares the elements. The skeleton of this module can be

```
'pragmat'module=sort.          $ module name
'action'qsort+[]st[].
'pragmat'prototype=reverse.    $ reverse the meaning
'question'qless+>x+>y.
'pragmat'prototype=reverse.    $ redo
'pragmat'if=compile.           $ module body starts here
'action'qsort+[]st[]:  ...     $ use qless+x+y for comparison
'root'+.
'pragmat'endif=compile.
```

The rule call `qsort+st` will sort the elements of the stack `st` so that for comparing two stack elements it uses the rule `qless+x+y`, with the assumption that this comparison rule returns true just in case `x` is "smaller than" `y`, whatever "smaller" means. In the module skeleton the first

```
'pragmat'prototype=reverse.
```

pragmat ensures that the `qless` prototype is handled correctly. When the module is compiled then `qless` is marked to be imported (since now the `prototype` pragmat is reversed to `import`). Similarly, when the module is required, then `qless` is to be exported (instead of the default `import`). The second `prototype` pragmat restores its original value; it can be omitted if there are no additional prototypes in the module.

When requiring the `qsort` module the rule `qless` must be specified by the invoking program. Since declarations there use a different namespace, `qless` must be declared with a qualifier:

```
'pragmat'require="sort".     $ require the sorting module
'stack'[]A=(3,4,7,1,0).
'question'sort::qless+>x+>y: $ use the module namespace
     x>y.                    $ sort in reverse order
'root'qsort+A.               $ and use it
```

When compiling the `sort` module the rule `qless` is left unspecified, and it is the linker's responsibility to replace it by the rule defined in the main program. Since there is a single compiled instance of `qsort` (in which calls to `qless` become hard-coded after linking) in the whole program, only a single instance of `qsort` can exist. This means that one cannot use such a sorting procedure with more than one definition of the comparison rule.

To overcome this problem, the invoking module could have a *local copy* of the sorting rule, which then can call the locally defined comparison rule. This would result in different sorting routines in different modules. To achieve this simply move the definition of the rule `qsort` from the module body into the head, and delete both prototypes. After this the module body becomes empty and can be omitted completely (consequently the module need not be compiled). The result is the `isort` module

```
'pragmat'module=isort.       $ inline sorting module
'action'qsort+[]st[]: ...    $ the complete sorting routine
                             $ using qless+x+y for comparison
'end'                        $ end of source
```

When the module `isort` is required, its head is compiled into the invoking unit locally, but keeping the module status and namespace. Thus the comparison rule `qless` still must be defined with a qualifier:

```
'pragmat'require="isort".    $ request the module
'question'isort::qless+>x+>y: x>y.
'stack'[]A=(3,4,7,1,0).
      ... qsort+A ...        $ this is a reverse sort A
```

Since everything is local to the invoking unit, different modules can require this inline sorting module and tailor the comparison rule `qless` according to their different needs.

As a final twist, this arrangement allows the `isort` module to have a *default* comparison rule, which should be redefined only when another sorting order is required. Simply add the default rule to the module head:

```
'pragmat'module=isort.       $ inline sorting module
'action'qsort+[]st[]: ...    $ the complete sorting routine
                             $ using qless+x+y for comparison
'question'qless+>x+>y:x<y.   $ default comparison rule
'end'
```

This comparison rule will be used when the invoking unit does not supply its own `qless` rule. It is so as during the compilation of the rule `qsort`, the defining instance of `qless` inside `qsort` is determined so that

- if there is a definition of `isort::qless` in the invoking unit, then that definition is used;
- if there is no such a definition, then the one in the module head is used,

see the discussion and the Figure in Section 5.2. If the invoking unit supplies its own comparison rule, then the default one is not used at all, and is not included in the final binary.

5.5 Redefining a module resource

The mechanism identifying definitions can be used to redefine transparently resources exported by a module. For an example, suppose that the module `MOD` in file `"MOD"` exports the rule `proc+>x+y>`. We would like to add tracing information before and after the call to this rule, but keep other resources supplied by `MOD` intact. This can be achieved by creating another module, say `MODtr`, which requests `MOD` and redefines `proc` in its head, and provides the alternate rule definition in its body.

```
'pragmat'module=MODtr.
'pragmat'require="MOD". $ read and export module MOD
'action'proc+>x+y>.     $ prototype, redefine this rule
'pragmat'if=compile.    $ module body
'a'proc+>x+y>: before+x, MOD::proc+x+y, after+y.
'a'before+>x: ....      $ print some tracing information
'a'after+>y:  ....      $ print some tracing information
'root'+.
'pragmat'endif=compile.
```

In the module body `MOD::proc` calls the original rule as defined in (exported by) the module `MOD`. Omitting the qualifier here would cause this rule to call itself making an infinite recursion.

Replacing `"MOD"` by `"MODtr"` in the pragmat requiring the module does all the tricks. All other resources provided by `MOD` are still available (without qualifiers), while calls to `proc` are now handled by the new definition in `MODtr`. Observe that internal calls to `proc` in the original `MOD` module are not affected.

5.6 Redefining library items

Assignments (transports in ALEPH parlance) and relations (of which identity is an example, see Section 4.3) are handled as a syntactically different way of writing a rule call. Internally, the assignment `a->b[c]->c` is transformed into the rule call `@make+a+b[c]+c` (recall that the character `@` is a letter in library mode, Section 4.25). The rule `@make` is exported by the standard library and has the prototype

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

Similarly, relations are transformed to calls of rules `@equal`, `@noteq`, `@more`, `@less`, `@mreq`, and `@lseq`, respectively; all of them are *question*s with two input affixes. They are also exported by the standard library. Any of these rules can be redefined (after switching to library mode) to do something different. As an example, suppose the list `STR` contains strings, and two pointers to `STR` should be considered 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 strings pointed by `ptr1` and `ptr2` are, as strings, equal. This can be achieved by redefining `@equal` to handle this case as follows:

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

When `x` and `y` are not string pointers the rule calls the original `@equal` from the standard library. Actually, the test `eq=0` should rather be `stdlib::@equal+eq+0`, as now this `@equal` calls itself. (Fortunately `eq` is not an `STR` pointer thus it won't fall into an infinite recursion.) In the module where this definition appears all equality tests will use this rule. To improve efficiency one might consider declaring this `@equal` to be a macro.

6 Standard library

Elements of the standard ALEPH library can be used without further notice, and can be redeclared in a user library (Section 3.3) or in the main text; see and example in Section 5.6. The ALEPH Manual [2] specifies many standard items, called *standard externals*, which should be available for the programmer. In this implementation most of them, but not all, are defined in the standard library module, called `stdlib`. The main omission is double precision arithmetic, while there are many additions. The complete list can be found in the ALEPH standard library source of the implementation with extensive comments. A few of the standard library items are discussed below.

6.1 Stacks

Calling `unstack+st` discards the rightmost block of the stack `st` by decreasing the actual upper bound of the stack by its *calibre*. The calibre of the declared stack is used even if `st` is a formal affix. The rule `scratch+st` discards all elements of the stack `st`, but keeps the allocated memory. It is implemented by moving the upper bound of `st` to its minimal value. The rule `release+st` additionally releases the memory allocated to the stack `st`, which will be reallocated again when `st` is expanded. This rule replaces `delete+st` from the Manual using the established terminology. Additional memory can be requested in bulk by calling `request space+st+n`. After a successful return the actual bounds do not change, but there are at least `n` additional words of allocated memory above the top of `st`. This rule fails if the requested additional memory is not available.

6.2 Strings

To copy a string to the top of the stack `st` use `copy string+t[]+>p+[]st[];` it is more efficient than unpacking and packing the string. The rule `compare string` compares two strings, returning a positive, zero or negative value depending on the relation of the strings. The rule

```
'q'compare string n+t1[]+>p1+t2[]+>p2+>n+res>.
```

compares the strings up to `n` UTF-8 characters. It is the counterpart of the C function `strncmp`.

6.3 File operations

Each ALEPH file has an associated error code, and ecah file operation sets this code. If the operation succeeds, the code is zero, otherwise it indicates the reason of the error.

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

recovers the error code from file `f`. It is zero if the last file operation preformed by `f` succeeded. Otherwise it is either the `errno` value set by the underlying C or system file operation, or a value above 10000 when it was set by the ALEPH file interface.

'p'open file+""f+>mode+t[]+>p
opens a file and associates it with the ALEPH file f. The string given as the last two affixes specify the file name with possible path information. The mode specifies how the file is opened; it is /r/, /w/ or /a/ for reading, writing, and appending. Only character files can be opened for appending, for details see Section 4.20. Character files can use the strings "<<stdin>>", "<<stdout>>" and "<<stderr>>" as filenames for the corresponding standard streams.

'p'open temp file+""f+[]st[]+>p
creates a temporary file (either character or data) and opens it for writing. The supplied string must end with six X characters, and must be on a stack as the rule replaces these characters by other ones which make the file name unique.

'a'put data+""f+>x+>type
writes x either as a word or as a pointer (depending on type) to the data file f, aborting the program in case of an error (for example when a non-zero pointer data doesn't point into one of the lists specified in the declaration of f). To handle errors use the predicate put datap+f+x+type which fails in case of an error. The error code can be retrieved by get file error,

'a'close file+""f
closes the file f. In case of an error the program is aborted.

'a'unlink file+t[]+>p
deletes the file specified by the string; this rule silently ignores errors.

'a'put string+""f+t[]+>p
writes the specified string to the character file f, while put as string+f+t+p writes the same string between quotation marks and doubles the quotation marks inside the string, producing a proper string denotation.

The ALEPH character files STDIN and STDOUT are associated with the standard input and standard output streams. These ALEPH files can be used, without opening, to read from and write to the terminal, see Section 4.20.

6.4 Miscellaneous

Command-line arguments can be retrieved from the external table STDARG. This table contains these arguments as strings in *reverse order* ending with the first argument at >>STDARG. The following rule prints all command-line arguments to the console in their original order:

```
'a'put arguments-ptr:
  >>STDARG->ptr,(nxt:
     ptr< <<STDARG;     $ no more argument
     put string+STDOUT+STDARG+ptr, put char+STDOUT+newline,
        previous string+STDARG+ptr,:nxt).
```

The rule exit+n terminates the program with exit code n, see Section 4.19. It can be written equivalently as the terminator 'exit'n.

The wait for rule can be used to synchronize module initializations. The root of each module is executed before the root of the main program is called.

The `wait for+"xxx"` call checks whether all modules with name `xxx` has finished executing their roots. If yes, it returns immediately. If not, calls those roots and waits until they return, see Section 4.12.

The rule `backtrack` prints out the complete call stack starting with the rule containing this call and ending with the program root if the ALEPH program was linked with the `-g` switch. Consult Section 4.27 from more details.

A lightweight, single word hash of a block of list elements is computed by the library rules

```
'f'string hash+t[]+>p+hash>.
'f'block hash+t[]+>p+hash>.
```

The first rule returns the hash of the string specified; the second rule computes the hash of a block of words starting at the element `p` and ending at `>>t` (the last element of the list `t`). This hash serves only as a tool to speed up checking whether two strings or two blocks are the same when many such comparisons should be made: they are definitely not equal if their hash is different.

7 Intermediate code: Alice

The ALEPH compiler produces an intermediate code for each module while typically reading several source files: the module source, the headers of required modules, headers of user libraries and that of the standard library. These intermediate codes are read, merged and linked by the linker which generates the final C code.

The intermediate code is called ALICE in reminiscence of the machine independent code designed for the first ALEPH compilers [3]. An ALICE file is a plain text file. Lines starting with the dollar sign `$` are comments and are skipped. The file has four sections: a header, a list defining all items used in this file, a data section, and finally a rule section. A tentative description of the details is given below by a grammar-like description with many comments. Grammar elements starting with D refer either to a single character, like Dpoint, Dcolon, Dstar with their obvious interpretation, or to a keyword written with apostrophe characters, like Dmain or Drule written as **`'main'`** or **`'rule'`**, respectively.

> alice file : header, item section, data section, rule section.
> header:
>   (Dmain; Dmodule, module string), title string, target word size,
>   (from line, end line, source string)*,
>   Dpoint.

The first non-comment line of the ALICE file starts with either **`'main'`** or **`'module'`**, where the latter one is followed by the module string which is the module name (as specified by the `module` pragmat) in quotation marks. It is followed by the title as given by the `title` pragmat, or `"aleph"` if no title was specified. The target word size is the number of bits in an ALEPH word, which is 32 for this implementation. The next block of three items each assigns file names to line numbers for determining the location in error and warning messages. Source lines are numbered consecutively starting from one, not resetting when opening a new source. This list contains the first and last number associated with a file. This way the relative line number and corresponding file name appearing in error and warning messages can be determined from the line number only.

7.1 Alice item section

The item section defines all identifiers appearing in this module, including imported, exported, and the local ones. This section has the following structure:

> item section:
>   (typer, item id, flags, lineno, type specific info, qualifier, tag)*, Dpoint .
> typer: the type of the identifier (see below).
> item id: letter `I` followed by a decimal number.
> flags: an integer containing type specific flags.

lineno: the source line of the definition.
type specific info: see below, can be empty.
qualifier: qualifier in quotation marks, can be "" (empty qualifier).
tag: the identifier enclosed in quotation marks.

The typer is one of the non-formal types defined in the `types.ale` module, written between sharp brackets such as `<charfile>`. Item id specifies how this entry is referred to in this ALICE file: it is the letter `I` followed by an integer. The integer starts at `1` and *must* increase by one for each item line in this section. The flags entry is a decimal number which gives the following flags (defined in module `tags.ale`:

public	set if the item is to be exported
imported	set if the item is to be imported from other module
external	set if it is an external item
ruleflags	eight bits specifying rule type and special handling.

Lineno is an integer which defines the source line of the item for error reporting. Type specific info gives additional information for lists (table, stack, and their static versions) and rules. For lists this part consists of two integers: calibre (block size), and the index of the standard selector (`-1` if none, otherwise this number is at least 1). For rules the type specific into specifies the number of affixes followed by the list of types of the formal affixes in the same syntax as the item typers. For formal lists the typer is followed by two integers: the calibre (`-1` if no block was specified), and the standard selector (`-1` for no standard selector).

Finally both qualifier and tag are strings enclosed in quotation marks. Exported and imported items are matched by their qualifier–tag pairs. For each imported item there must be exactly one exported item in the other modules.

7.2 Alice data section

The data section of the ALICE file contains additional information for list and file declarations, fillings, and contains the initial values for constant and variable declarations.

data section: (data description)*.
data description:
  Dlist, list definition, Dpoint;
  Dfile, file definition, Dpoint;
  Dexpression, expression, Dpoint;
  Dfill, filling, Dpoint;
  Dfront, string, Dpoint;
  Dback, string, Dpoint;
  Dexternal, item, string, Dpoint.

A list definition contains the size information for a stack or for a static stack in the following format:

list definition: item, est type, est size.

The integer est type is one of 0, 1, 2, 3, 4 indicating that the size estimate for the list is [] (empty), [cons], [=cons=], [tag], or [=tag=], respectively. The est size is either a constant when etype is 0, 1 or 2, or is an item otherwise.

A file definition describes the lists in the file area (if present), the string after the = sign in the file declaration, and the direction of the file:

file definition: item, direction, table item, pointer item, [file area].
file area: Dsub, (list item)*, Dbus.

The file area part is optional, and is present only if it is a data file declaration specifying lists. Direction is one of 0, 1, 2, or 3 describing whether the file is for input (1), output (2), or both. The table item is the item identifying the standard string table, while the pointer item is a pointer constant pointing to the corresponding string in the standard string table. The optional file area contains the list items in the order they were given in the file declaration.

An expression describes the defining expression of a constant or a variable declaration.

expression: item, expr.
expr: target integer; item; expr, operator, expr; Dopen, expr, Dclose.

An expression typically consists of a single target integer, but it can be an arbitrary expression, including parentheses, unary and binary operators (including virtual lower and upper bound, calibre, arithmetic and boolean operations). A target integer starts with the letter X followed by a signed decimal number in the range of the target word size.

A filling specifies the list fillings as specified in the module text. It follows the complicated structure of fillings, and includes definitions of constant pointers.

filling: item, source line, (fill item, [repeater], initializer*)*, Dpoint.
fill item: target integer; target string; item; compound fill.
compound fill: Dopen, (target integer; item)*, Dclose.
repeater: Dstar, (integer; item).
initializer: Dcolon, item.

All items in an initializer are pointer constants, and they get their values after all virtual addresses have been calculated. Consequently repeater values cannot depend on pointer constants, and there is a strict limit on how large a repeater can be. A target string is a letter T followed by a string denotation between quotation marks, to be packed into an ALEPH string in the target architecture.

The Dfront and Dback lines repeat the strings from the `front matter` and `back matter` pragmats. Finally, Dexternal lines associate an external item with the string specified in the external declaration of that item.

7.3 Alice rule section

The rule section of the ALEPH file contains the compiled form of the rules defined in the module. This section is a sequence of rule definitions, and each such definition consists of a head, followed by a sequence of nodes describing which other rules are to be called, and at which node should the computation continue.

> rule section: (rule definition, Dpoint)*.
> rule definition: rule head, (node definition, Dcomma)*.
> rule head: item, min local, max local, node count.
> node definition: call node; extension; classification.

The item in the rule head is the rule name, it is followed by three integers. If the rule has no local variables, then both min local and max local are zero. Otherwise local variables of the rule are numbered from min local to max local, including bounds. (Actually, if min local is not zero, then it is one more than the total number of in, out, and inout affixes of the rule.) Finally, node count is the number of nodes following the rule head.

In the rule body nodes are denoted as `N1`, `N2`, etc, where the number after `N` goes from `1` until the node count inclusive. Local variables are denoted as `Ld` where `d` goes from min local to max local. Finally, the formal variables are denoted as `Fd` where `F1` is the first formal affix, and the number goes ahead until the number of the last formal affix. Formal affixes are in a one-to-one correspondence of the rule type description in the item section.

7.3.1 Call nodes

A call node corresponds to a rule call and has the following format.

> call node: Dnode, node, item, C1, C2, C3, (affix)*, false node, true node.
> false node: Dout, node label.
> true node: Dout, node label.
> node label: 0; -1; -2; node.

The node after the Dnode symbol and in the node label is a node identifier starting with `N`, followed by an integer. The item, starting with `I`, identifies the rule to be called. It is followed by three integers C1, C2, and C3 giving information on the required stack size when calling the rule; these numbers are discussed later. After the description of the actual affixes the continuation is specified: false node and true node identifies the nodes where the execution should continue when the call returns false and true (fail and success), respectively. False node can be the number zero when the call cannot fail; it can be `-1` denoting true exit, and `-2` for false exit from this rule. In other cases these are node identifiers starting with `N` followed by the node number. The true node is never zero.

The affix part describes the actual affixes of the call.

> affix: target integer; item; formal; local; limit type, item;
> Ddummy; indexed affix; anchor; Danchor.

limit type: Dupb; Dvup; Dlwb; Dvlwb; Dcalibre.
indexed affix: Dsub, item, affix, Dbus, selector.
anchor: Danchor, integer.

A target integer is a (signed) decimal integer starting with an `X`. An item, a formal or a local start with the letter `I`, `F`, and `L`, respectively, followed by an integer as discussed above. When the actual affix is a limit, the item following the limit type is either a global item, or a formal affix. Upper and lower bounds can be either dynamic or static, see Section 4.13, as indicated by the different names. The Ddummy symbol indicates the dummy actual affix meaning that the result in the corresponding out affix can be discarded, see Section 4.5. An indexed affix describes an indexing. The item after the Dsub symbol is either a global or formal list which is to be indexed. The affix gives the actual index (which can also be an indexed affix), and selector is an integer giving the offset within the indexed block: for the rightmost element of the block it is 1, and it increases by one.

A formal or local in the affix list might be preceded by a Dcolon symbol. It signals that in that affix the called rule returns a value which is not used later, thus the caller can safely skip retrieving and storing that value.

If the called rule has a variable number of affixes, then an anchor appears at the `@` affix position. The integer after Danchor specifies the number of the following *affix blocks* (not the number of the affixes in those blocks) as provided by this rule call. This number is strictly positive if the last actual affix in this call is *not* `@`. If the last actual affix is `@`, then this number is the negative of the provided blocks (which, in this case, can be zero); and only in this case is the last actual affix before the false node a single Danchor.

The constants C1, C2, and C3 for the called rule are computed as follows. If the last actual affix of this call is *not* `@`, then C3 is zero, C1 is the total number of the actual affixes, and C2 is the number of those affixes which either have type out or inout (they contain output values), or are in some repeat block (independently of their types). If the last actual affix is `@`, then C1 and C2 are computed as before for affixes *before* the anchor affix position (for affixes which are not in the repeat blocks). The value of C3 is the actaul number of affixes in the repeat blocks, except for the last actual affix `@`. Thus in this case `C3` is zero exactly when the the only actual variable affix is `@`.

7.3.2 Extension nodes

An extension node tells that a (global or formal) stack should be extended by a block, and specifies the content of the block.

extension: Dextension, node, item, width, (field data)*, Dout, node.
field data: affix, (Dto, offset)*.

The node after the Dextension symbol is the node identifier; it starts with the letter `N` followed by an integer. The closing node after the Dout symbol is the next node, which can be either -1 or another node identifier (since an extension cannot fail). The item is the (global or formal) stack to be extended, width is an integer giving the number of words in the extension block. The field data defines the content of the block: offset is an integer between 1 and width, and each offset occurs exactly once. The same value, described by the affix can be stored at several offsets. The block content (the values of the affixes in the field data) should be computed *before* the upper limit of the stack is adjusted.

7.3.3 Classification nodes

A classification node contains the affix in the classifier box together with all areas and their destination nodes. While in an ALEPH program the classifier cannot be a constant, due to macro substitution, the linker should be prepared for this case as well.

classification: Dbox, node, affix, (area description)*.
area description: Darea, source line, zones, Dout, node.
zones: zone, (Dsemicolon, zone)*.
zone: list item; interval; value.
interval: (value;), Dcolon, (value;).
value: constant item; target integer.

The node after the Dbox symbol identifies the classification node. It is followed by the affix specifying the classifier's value. Each area starts with Darea followed by the source number of that area. This is specified for the potential messages when the area cannot be selected. The area ends with the success destination node. The area is a semicolon separated list of zones; the area succeeds if any of the zones succeeds. Each zone is either a (global) list item standing for its complete virtual address space, a constant item, a target integer, or an interval where both the lower and upper bound may be missing.

After determining the constant values and the virtual limits of lists, the linker checks that each area can be reached by some value of the affix; gives an error message if this is not the case. There is no "otherwise" part of a classification. If none of the areas succeeds, then the program run is aborted with an error message. The linker gives a warning if this might happen.

8 Target code

The target code is produced exclusively by the ALEPH linker from the intermediate ALICE code of the main program, modules, and standard library. In this implementation the target code is standard C. It has been chosen as a C program can be easily compiled into an executable binary, and C also provides the basic tools for creating the running environment, memory management and minimal file transput routines.

Code generation assumes that the basic ALEPH data type, the *word* has 32 bits; it is translated to `int` in C. Some efforts have been made to allow code generation for a different word size architecture. In this respect two more issues should also be addressed: what character encoding is used, and how strings are represented in the target words. The details given here assume that the target word size is 32 bit, and strings are packed into UTF-8 encoded bytes. The present implementation makes no assumption on the size of C pointers.

8.1 Tables, stacks

An ALEPH table, stack, datafile and charfile is represented as an *index* to a global array called `a_DATABLOCK`. This array is split into structures specific to the required data type. Given the index `idx` of this array, the C macros `to_LIST(idx)`, `to_CHFILE(idx)` and `to_DFILE(idx)` cast the pointer to `a_DATABLOCK[idx]` to a pointer to the corresponding list, character, or datafile structure, respectively.

The table and stack structures have the following fields.

`int *offset`	the zero address virtual element of the list
`int *p`	pointer to the beginning of the allocated memory block
`int length`	number of words in 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 `L[idx]` using the virtual address `idx` (which points to the *virtual* address space of `L`) is translated to the C construct `to_LIST(L)->offset[idx]`. 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 a list can be moved freely in the memory as long as the pointers `offset` and `p` are adjusted properly.

The `unstack` and `unstack to` externals adjust the actual upper bound only. The `release` external actually frees the complete allocated memory, while `scratch` only sets the actual upper bound to its lowest possible value but keeps the allocated memory. The `request space+L+n` external checks that there are at least `n` additional allocated words above the actual upper bound of `L`, and if not, then it allocates additional memory. This may result in moving the whole list to somewhere else in the actual memory, when `offset` is adjusted accordingly.

When a stack is to be extended by a block of n words in an extension, the C routine a_extend(L,n) is called first. This routine makes sure that L has an additional n words of free space at the top (by calling request space), and, additionally, returns the address of the first free slot at the top of L. Subsequently the free block is filled by assigning values to B[0] to B[n-1] where B is the address returned by a_extend. Finally, the actual upper bound of L is increased by n.

8.2 Data file

As discussed in Section 4.20, an ALEPH datafile stores integers (words), and pointers to lists. Such a datafile is realized as a sequence of blocks of size 1024*sizeof(int), and each block B[0..1023] 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]	actual data.

In the bitmap part there is a signle indicator bit for each word of the block at index $32 \le \mathtt{i} \le 1023$. This bit is at word B[int(i/32)] and 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 relative value zero; the eof (end of file) indicator is a pointer with relative value -1; all other pointer values must be positive and belong to one of the datafile zones.

The first few data values in the first block of the datafile contain the *zone list*. Each zone occupies three words: virtual lower and upper bounds, and the numerical position of the list. The size of the list is in B[32], data for the first zone is in B[33], B[34], B[35], followed by data for the other zones. The list must fit into the first data block. The lower and upper bounds (inclusive) are strictly increasing (thus these ranges are disjoint and positive), therefore all pointer values, with the exception of nil and eof, must be positive.

Items in a datafile have positional data. The last 10 bits in this file position identify the index within the block; this value must be between 32 and 1023. Other bits of the position identify the block in which the actual value can be found. This file position is stored internally, thus there is no overhead in determining it. The file position can be retrieved for both input and output datafiles, but one can only set the position for an opened input datafile. No check is made to make sure that the position is valid (so it can be set after the eof indicator). Since a word contains 4 bytes and this address must be positive, an ALEPH datafile cannot be larger than about 8 Gbytes.

When opening a datafile for output, the first block is created with the specified zones sorted according to their virtual address ranges; the file pointer is set just before the very first empty space.

Appending to an existing ALEPH datafile is not implemented as it raises several problems. The first block should be read and checked if it has the same metainformation as the current file declaration requires, position it to 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 order of the lists in the corresponding declarations. When the next input is requested, it is checked whether it is a pointer or not. If it is numerical, pass as it is. If it is a pointer, check which list it is in, add the correction 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. This error condition can be retrieved from the `file error` associated with the ALEPH file.

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 for 32 words
`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

The string pointers `st1` and `st2` identify the string supplied by the file declaration. This string is used as a file name when the file is used without explicitly opening.

8.3 Character file

While datafiles use direct file input and output, character files use streams, namely the `fgetc()` and `fputc()` C library procedures without the `ungetc()` facility. The input is assumed to be proper UTF-8 encoded file, incorrect codes are silently ignored. The ALEPH rule `get char` 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 as well, but they use `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

The `aheadchar` stores the next (unread) UTF-8 character when a look-ahead was made. This happens, for example, when an integer is read from the file by the standard rule `get int`. This rule should stop at the first non-digit character, but not consume that character.

8.4 Strings

ALEPH strings use Unicode characters, and they are stored using UTF-8 encoding as C strings with `\0` as the last byte. If the string is in list `L` pointed by the (virtual) index `idx`, then the content of the list block is

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

The empty string is stored as a block of `(0,0,3)`.

8.5 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 a *question* or *predicate*, then it is `int`. The compiled C routine returns `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 in its destination, and do it only if the called routine reports success. According to this requirement, formal affixes are transformed into C parameters as follows. First, assume that the called rule has no variable affix block. Affixes which are neither out nor inout ones (that is, file, stack, table, or in) are passed as integers in their original order. A local integer array is declared for the out and inout affixes, and this array, containing the value of these affixes in their original order, is passed as the last parameter. Before returning, the called routine supplies the output values in this array, which values are then stored by the caller.

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

```
+t[]+>i       (int t,int i)
+""f+o>       (int f,int A[1])
+>io>+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[]`, otherwise it is free to change (and use) these values if the routine fails. In the variable block `V[]`, however, values corresponding to not out or inout affixes cannot be changed, and the value of an inout affix should change only if the routine returns with success.

8.6 Externals

External declarations are allowed in library mode only (see Section 4.25). The interpretation of the string denotation in the external declaration depends on the type of the defined tag.

8.6.1 External constant and variable

External constants cannot be used in expressions or other places where a constant tag is required. In the C external variables and constants can appear as rule parameters; they are replaced by the string specified in the external declaration.

8.6.2 External table and stack

A list structure is reserved in the global integer array `a_DATABLOCK` as explained in Section 8.1. The string in the external declaration is used as the name of a C procedure which is responsible for initializing this structure. The routine is called with three arguments: the index of the associated structure, a constant string with the name of the list, and the calibre. The routine must fill the actual and virtual limits and the calibre. There is a (relatively small) virtual address space set aside for external lists. The first free virtual address is in `a_extlist_virtual`; the address can go up to `max int`. The routine should update this value to reflect its reservation. The routine is also responsible for allocating memory and initializing the content of external tables.

8.6.3 External files

The corresponding charfile or datafile structure is reserved in the global array `a_DATABLOCK`. For the description see Sections 8.2 and 8.3. The string in the external declaration is used as the name of a C procedure which is responsible for initializing the structure. The procedure is called with two arguments: the index of the structure and the name as a character string.

8.6.4 External rules

How an external rule is handled depends on the string denotation. If it starts with a (lower or upper case) letter, then the external rule is assumed to be a C procedure with exactly the same parameter passing mechanism as the compiled rules, see Section 8.5. There must be a header file providing the prototypes of these external procedures, it can be added to the generated code using a `front matter` pragmat. Several standard library rules are implemented this way.

If the first character in the string denotation of the external rule is an underscore _, then another calling mechanism is used: all affixes, independently of their types, are passed as parameters. Such external rules are typically defined as C macros; an example is the `incr+>x>` external rule whose string denotation is `_a_incr`. The ALEPH rule call `incr+ptr` translates to `a_incr(ptr)`. The standard library header file contains the C macro definition

```
#define a_incr(x)   x++
```

which makes the final translation.

The dummy affix # translates to nothing, thus it 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 does some additional work. If the string denotation of the external rule starts with a `@`, then this character is discarded. For each out argument, depending on whether it is the dummy symbol or not, a `0` or a `1` character is appended to the remaining string. Finally, dummy symbols are discarded from the argument list. In the standard library the external rule `divrem` has two out affixes, and its string denotation is `@a_divrem`. Accordingly, four C calls could be generated: `a_divrem11` with four parameters when both the quotient and remainder is used, the three parameter `a_divrem01` and `a_divrem10` when the quotient or remainder is discarded; and the two parameter `a_divrem00` when no result is requested at all.

The external rule strings `@@make` and `@@waitfor` are exceptions; these rules are handled internally by the linker when generating code for transput (assignment) and for the `wait for` rule.

If all out arguments of a function are discarded, then the rule is not called at all. Similarly, if the returned value of a question is not used, then the question is not called. It might cause problems when the function or the question performs some conditional tasks before calling an exit rule (Section 4.19), as those checks will not be carried over.

References

[1] A.V. Aho, J.D. Ullman, *Principles of Compiler Design* Sseries in computer science and information processing, Addison-Wesley, 1977

[2] D.Grune, R. Bosch, L.G.L.T.Meertens, *ALEPH Manual* CWI, IW17/74, Stichting Mathematisch Centrum, Amsterdam, Fourth printing, 1982

[3] D.Grune, *On the design of ALEPH*, CWI Tract 13, Centre for Mathematics and Computer Science, Amsterdam, 1982

[4] C.H.A.Koster, *A Compiler Compiler*, CWI Report MR127/71, Mathematical Centre, Amsterdam, 1971

[5] C.H.A.Koster, *Affix Grammars*, in: J.E.L.Peck (Ed.), *ALGOL 68 Implementation*, North Holland, Amsterdam, 1971

[6] C.H.A.Koster, *Using the CDL compiler*, in F.L.Bauer and J.Eickel (Eds.) *Compiler Constructions* LNCS 21, Springer, 1974

[7] C.H.A.Koster, J.G.Beney, P.A.Jones, M.Seutter, *CDL3 manual*, available as `https://ftp.science.ru.nl/cdl3/cdl3-manual-1.2.7.pdf`

[8] A.van Wijngaardeen, *The generative power of two-level grammars*, in J.Loecks (Rd.), *Automata, Languages and Programming*, LNCS 14, Springer, 1974

[9] A.van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker, *Revised Report on the Algorithmic Language ALGOL 68*, Springer Science & Business Media, 2012