ALEPH Compiler

L. Csirmaz

1 Preface

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, 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)" according to the manual of the language [1]. 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 [2].

ALEPH is a direct descendant of another extinct language, CDL, standing for Compiler Description Language. CDL was designed by C. H. A. Koster [3, 5] 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 [6]. Both ALEPH and CDL belong to the family of the few languages based on affix, or two-level, or van Wijngaarden, grammars [4, 7]. Affix grammars were developed to provide a formal definition of what an ALGOL68 program is. The appealing intuitive meaning of an affix grammar definition combined with the theoretical simplicity and completeness soon led 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, while the *affixes* of the grammatical symbol are appended to the procedure name using + (plus) symbols. Affix values come from another, very restricted language, and they behave like parameters of the procedure.

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 using 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 exactly the available data types and fixing the data manipulating primitives.

The unusual *call-then-store* mechanism of ALEPH is inherited from CDL. Output parameters are local for the called procedure, 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 [1], 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 possibilities resembling the CDL2 approach. There are several other extensions, changes and restrictions, hopefully all of them in the spirit of the original design of the language.

2 A bird's eye overview of ALEPH

The ALEPH manual [1] 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. 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 the context-free parsers, ALEPH rules can be equipped with *affixes*. Affixes carry auxiliary, context sensitive information. The syntax of ALEPH resembles that of affix grammars as it uses the + sign to separate the procedure (rule) name and the parameters (affixes).

2.1 Data types

The basic data type of ALEPH is *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 by tables and stacks. 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, consequently a pointer uniquely determines the list it points into.

Consecutive locations in a table or in a stack can be grouped together either to a 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*. Strings, on the other hand, behave like black boxes, and can be manipulated by predefined routines only. This way characters in a string are not restricted to single byte (or wide) characters, but can use arbitrary character sets.

Finally, for communicating with the outside world, ALEPH has character and data files. They can be opened, closed, and can send and receive characters (charfile), or words and pointers (datafile).

2.2 Rules and calling a rule

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

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

Each actual affix represents a word (a constant, a variable, or an indexed element), a list (stack or table), or a file. No compound affixes are allowed. A rule either returns a logical value or returns nothing. 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 (thus 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 abbreviations for tail recursion. There are no repetitive statements, iteration is 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 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 reach a return point;
- when a local variable or parameter is used it has a value assigned to it (it is not “uninitialized”);
- an out parameter has been assigned a value when the routine reaches any of its return points;
- if a local variable (or parameter) has been assigned a value, it is actually used.

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

2.3 Externals

Basic data manipulation, such as addition, comparison of words (integers) and similar operations, are done by (standard) external rules. As an example `incr+x` increases the value of its argument by one; and `equal+x+y` tests for equality. All file operations are done by externals. Externals can be redefined; this feature can be used to mimic overloading basic operators.

2.4 Prototypes

Originally ALEPH has no prototypes. It was, however, a natural extension to the language. Prototypes are indispensable when the program is split into smaller modules which can be compiled independently. Each module should have complete information on all objects (rules, variables, stacks, etc.) it imports, and should provide that information on those objects it exports. Prototypes are just the right constructs for this purpose.

2.5 Modules

An ALEPH module provides certain resources to other modules and can be compiled independently. A program can *require* resources provided by certain modules, and can *export* resources defined within it. Accordingly, an ALEPH module has a public and a private part. The public part specifies the exported items using prototypes, and the private part contains the realization of the provided resources. When the module is required, the public part is scanned. When the module is compiled, the compiler checks that all objects this module promises to export are indeed provided by its private part.

The public part of the module may contain other declarations next to the list of export prototypes, and may even require additional modules. Such declarations will be compiled into the invoking module using the module namespace.

An ALEPH module can redefine any resource provided by a required module. To reach the hidden definition, name qualifiers can be added in the usual way.

3 Enhancements and changes

The ALEPH Manual [1], the official definition of the language, has several revisions. All of them were published by the Mathematisch Centrum, Amsterdam, between 1974 and 1982. Implementations frequently add new features, restrict or leave out others. This happened with ALEPH as well; these changes are reflected in the different versions of the Manual. The ALEPH language implemented – and used – by the present compiler is no exception. Making modular programming possible – as opposed to using monolith programs prevalent at the time of the creation of ALEPH – required new features. Other extensions and restrictions came naturally; some of those date back to the time of the first ALEPH compilers. All changes made by this implementation hopefully respect the original design ideas and philosophy.

3.1 Program text representation

ALEPH is an Algol-like language 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:

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

Other possibility could be 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.

Characters in ALEPH are not restricted to single bytes as character 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 can be part of a string, but not of a string denotation (strings 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.

3.2 Hexadecimal constants

Wherever 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.

3.3 Relations

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

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

3.4 Lists in scalar context

In an 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, making writing and comprehending stack operations easier. The 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 for details.

3.5 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 the source file
`_module_`	string, module name as set by the pragmat `module=...`
`_title_`	string, the title as set by the pragmat `title="..."`
`_rule_`	string, the rule name if used inside a rule definition.

3.6 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 sign `#` for this purpose, while the official representation `?` is also accepted.

3.7 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.

3.8 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.

3.9 Variable number of affixes

Allowing and handling a variable number of rule affixes is one of the 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 ALEPH programs. 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 the affixes to the right end of the list can repeat indefinitely. The rule declaration `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, then adds `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 its 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 elements 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 rest, and then subtract it from the sum of the first two. Thus the calls

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

put 7 to `x`, 6 to `y`, and -7 to `z`, respectively.

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 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`.

```
'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, see Section 3.17. The assignment `st->x` stores the topmost element of the stack `st` in `x` as discussed in Section 3.4.

```
'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+@.
```

3.10 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;
 [-10:10],action 2;
 [L;1000],action 3;
 [:],      action 4)
```

The area in the square brackets determines whether the alternative following it will be chosen. 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 in an 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 is aborted with an error message.

3.11 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 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.

They have lower priorities than the arithmetic operators.

In expressions integer denotations (both decimal and hexadecimal), constant tags, pointer constants (defined in fillings), virtual bounds and block size (calibre) can be used. List size estimates and repeat numbers (see Section 3.18) are evaluated before the virtual bounds are determined, thus these values cannot depend on these bounds.

3.12 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;+)).
```

Module roots 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 the rule

```
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.

3.13 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 do not change during runtime. 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 subrange. 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 address 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.

3.14 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; 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.)
- 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 3.18). Such a stack can still shrink, but cannot expand beyond its virtual upper limit.

3.15 Table declarers

To distinguish table declarations from prototypes and fillings (see Sections 3.22 and 3.18, respectively), a table declaration must contain an empty size estimate:

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

3.16 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, must contain at least one selector.

The same block element can be identified by several 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 public prototype must contain a selector list pack, while a filling may have such a pack, see Section 3.18. These selector packs must define the same block size and the same standard selector (but could define additional selector names).

3.17 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.

3.18 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 was defined with selectors. If there is a selector pack, both the block size and the standard selector must be the same; the selectors, 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. 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 follows: a colon `:` and a tag which is defined to have the the value of the virtual address of the lastly defined list item. In the example

```
'constant'tsize=10.
'stack'T=(0*tsize:tzero,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 `tzero` 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 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 in the 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 one, six zeroes, and another one. The block can also be written as (1,0*6,1).

3.19 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 an expression.

In general, next to the four rule types *predicate*, *question*, *action*, and *function* specified by the ALEPH Manual, 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, these conditions are 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.

3.20 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 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 limits of 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 should contain only those lists to which pointers are sent to or received from. The order of the lists is significant: when reading from a file the first list in the area is matched to the first list when the file was written.

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 (allowed 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.

3.21 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.

3.22 Prototype

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

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

Prototypes are used to inform the compiler about tags which are defined in other modules, and tags which should be exported. See Section 5 for how modules can be used.

3.23 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 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.
`stdlib=off`	don't include the standard library
`define=tag`	mark `tag` as defined for an `ifdef` pragmat
`library=on/off`	switch library mode
`prototype=none`	specify how prototypes are handled (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`
`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 3.24. Command-line arguments starting with a double dash, such as `--XX=YYYY` are parsed as

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

except that no conditional pragmats are accepted, see Section 3.24. 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`	add `"XXXX"` as a library file
`-o XXXX`	specify the output file
`-I XXXX`	search directories
`-L XXXX`	the 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 which is neither module nor library, and generated in the current directory. The `-y` option marks the following source file to be processed as a library module. The `-I` option specifies the list of search directories for source files, requested modules and library modules. Finally the `-L` option specifies where the compiler should look at the standard library files.

Default value of some of the pragmats is the following:

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

The 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.

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). 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.

3.24 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`, 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` (`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.

3.25 Library mode

Pragmats `library=on` and `library=off` turn the library mode on and off, respectively. This mode determines whether the 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 Macro substitution

Calls to macro rules are processed by textual substitution. Such a replacement can result in a syntactically incorrect 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 has a syntax error:

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

2) There is a problem with the dummy affix `#`.
   Using the same `macro` rule as above, the substitution has incorrect syntax:
   `macro+u+#` becomes `(u->#->u, u->#)`

3) While a macro can have a variable number of affixes, neither `shift affix block` nor `get affix blockno` can be used in a macro text.
   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, it would shift out all affixes and the computation in `math` would not be carried over.
   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` becomes `(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` becomes `(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 local variable (which may be removed during optimization). For 4) the macro substitution mechanism remembers the last substituted formal affix, which gives the correct standard selector. For 5) and other side effects, no warning is, or can be, given, but substitution changes the semantics. So use macros with care.

5 ALEPH modules

An ALEPH module typically starts with a `module=XXX` pragmat which specifies the name, and also the namespace, of the module to be `XXX`. It is followed by the *public part* containing the prototypes of the exported (public) tags defined by this module. The *private part*, called the body, defines the exported items together with the auxiliary, unexported items. The body is enclosed between `ifdef=compile` and `endif=compile` pragmats.

When the module is requested by another ALEPH module or program using the `require="XXX"` pragmat, only the public part – the head – is processed. The prototypes in the head are considered to be external definitions, which are to be imported by the invoking program, and are provided by the module. When the module is compiled, both the head and the body are processed. Prototypes in the head specify which items will be exported, and the compiler can check that they are indeed defined in the body of the module.

The following example defines a sample module which exports the rule `do something` and the stack `LEXT`.

```
'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 source is read with an implicit initial `compile=on` pragmat. The `module` pragmat in the source defines the module name and namespace to be `sample`, and also automatically sets the `prototype` pragmat to `public`. The module head is parsed, and the compiler marks all prototyped tags to be exported. It means that those 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 read and compiled. The stack `LEXT` and the rule `do something` are compiled, and prepared for export using the namespace of the module.

When the same module is required by another ALEPH module or program by issuing a `require="sample"` pragmat, the module text is parsed with an initial `compile=off`. In this case the `module` pragmat also sets the module name and the namespace to `sample`, but sets the `prototype` pragmat differently, namely to `import`. Next the prototypes are scanned and the items are marked as "to be imported". The compiler complains if any of these tags is not used properly. Reaching `if=compile` the condition fails, thus the remaining part of the module is ignored.

The body of a module can require public items from other modules. It may happen, without any problem, that the body of module `a` requires public items from module `b`, and the body of module `b` requires public items from module `a`.

The public part of a module may also contain additional ALEPH constructs, not only prototypes. A `require` pragmat here makes the imported items automatically available to the invoking program. Declarations are compiled into the invoking program, but using the module's namespace.

Items defined in a module can be redefined by the invoking program, and the new definition can be exported. The next example shows a module which redefines the rule `proc+x+y` exported by another module to print out some tracing information. Other tags exported by the `MOD` module are made automatically available to the program requiring the module `MOD with printing`.

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

Here `MOD::proc` is the original rule as defined by the `MOD` module. Omitting the qualifier would cause the rule to call itself making an infinite recursion.

Similar mechanism allows redefining items in libraries, as definitions in library modules (including the standard library) are used only as a last resort when no other definition has been found.

5.1 Required and included source files

Source files are handled one at a time, they are read, processed and closed before opening the next file. Source files can be specified on the command line, requested by a `require` pragmat, or included by an `include` pragmat.

The pragmat `require="file"` appends the source `file` to the end of files to be processed as a *module*. Each module is processed only once. Library modules (including the standard library) are handled similarly, and processed after all other sources have been finished. Source files added by the `require` pragmats in a library module are treated as libraries.

The pragmat `include="file"` always appends `file` to the end of the source list keeping 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, included sources are processed as many times as they are specified.

When a source is processed as a required module, implicit `compile=off` and `prototype=import` pragmats are executed. When the source was added in the command line, an implicit `compile=on` and `prototype=none` pragmat is executed. The effect of the `module=xxx` pragmat depends on whether compile is on or off. If `compile=on`, then it switches to module compilation and sets `prototype=public`. If `compile=off`, then it reads a module head, and sets `prototype=import`.

5.2 Using the namespace

The actual namespace affects only definitions (declarations and import prototypes) by adding the namespace automatically as the qualifier to the defined tag. The namespace is empty in the main program; and it is the same as the name of the module otherwise. Specifying the namespace explicitly is necessary, for example, when a module uses a callback function defined outside the module. The sample module `quicksort` below sorts the elements of the stack `st` between the pointers `from` and `to`. For comparing two stack elements it uses the callback function `qless+x+y`; it returns true if the element pointed by `x` is "smaller than" the element pointed by `y`. The skeleton of the module can be

```
'pragmat'module=qsort.         $ module name
'action'qsort+>from+>to+[]st[].$ the sorting routine
'pragmat'prototype=reverse.    $ handle qless
'question'qless+>x+>y.
'pragmat'prototype=reverse.    $ do it back
'pragmat'if=compile.           $ module body start
'action'qsort+>from+>to+[]st[]:
   $ use qless for comparison ...
'root'+.
'pragmat'endif=compile.
```

The first `prototype=reverse` ensures that the `qless` prototype is handled correctly. When the module is compiled then `qless` is marked to be imported (that is, `prototype=import` instead of the default `public`). When the module is requested, `qless` is to be exported (instead of the default `import`). The second `prototype` pragmat restores the original value; it can be omitted if there are no more prototypes in the module.

When using the `qsort` module, the rule `qless` must be exported as was required by the `qless+>x+>y` prototype. The caveat is that declarations in the invoking program use a different namespace. Thus the program must specify `qless` using the quicksort module name as qualifier:

```
'question'qsort::qless+>x+>y: $ use this namespace
     x>y.                     $ sort in reverse order
'root'qsort+<<A+>>A+A.        $ and use it
```

5.3 Redefining library tags

The definition of an identifier – which is either an import prototype or a declaration – is determined as follows. First, the actual source is scanned for a definition. If found, it is the definition, and the search is finished. If not found, then modules required from this source are checked; if not found, then modules required from the required modules checked, and so on. There must be a unique definition of the first level where such a definition is found, but there might be other definitions at higher levels, which are ignored. This feature can be used to redefine a rule provided by a module as in Section 5.2, and can also be used to redefine library routines.

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

Internally, the assignment (transport) `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, see Section 3.25). 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=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=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 Implementation details

The target code of this implementation is standard C. It is assumed that the basic ALEPH data type translates to `int` which is 32 bits long. There is no assumption on the size of C pointers. The compiler has been written so that the word size of the generated code can be changed. Two more aspects should be addressed when using different target word size: what is the character set and how strings are represented on the target machine when running the generated code. The details given here are for the case when the target word size is 32 bit.

6.1 Tables, stacks

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

Table and stack structures have the following fields.

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

The ALEPH list element L[idx] 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.

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

6.2 Data file

As discussed in Section 3.20, ALEPH datafiles store integers (computer words), and pointers to lists. An external datafile is a sequence of 1024*sizeof(int) blocks, 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 an indicator bit for the word at position $32 \le$ i ≤ 1022, this bit is at word B[int(i/32)] and position (i&31) (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 values in 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 the ranges are disjoint), and all pointer values, with the exception of nil and eof must be positive.

When reading a datafile, it has 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).

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

Appending to an existing ALEPH datafile is not allowed as it raises several problems. The first block should be read, checking if it has the same metainformation as the current file, position 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, and 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 zones. 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 difference and pass it as a pointer. Handle `nil` and `eof` separately. If the zone is not found (the corresponding list was not supplied when opening the datafile), then fail and skip this input.

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

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

6.3 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. Input is assumed to be proper UTF-8 encoded, 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; 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

6.4 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
`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 three zeros.

6.5 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.

6.6 Externals

External declarations are allowed in library mode only (see Section 3.25). The interpretation of the string denotation in the external declaration depends on the type of the defined tag.

6.6.1 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.

6.6.2 External table and stack

A list structure is reserved in the global integer array `a_DATABLOCK` as explained in Section 6.1. 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.

6.6.3 External files

The corresponding charfile or datafile structure is reserved in the global array `a_DATABLOCK`. For the description see Sections 6.2 and 6.3. 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.

6.6.4 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 6.5. 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 string `@@make` is an exception; it is handled internally by the linker when generating transput (assignment).

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.

References

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

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

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

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

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

[6] 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`

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