A. Günther et al. (eds.), International Computing Symposium 1973
© North-Holland Publ. Co., 1974.

ALEPH, A Language Encouraging Program Hierarchy [1]

Rob Bosch, Dick Grune, Lambert Meertens
Mathematical Centre, Amsterdam, the Netherlands

An ALEPH program consists of a set of grammar-like definitions of input, actions to be performed and output, properly interrelated. The syntactic and semantic simplicity of ALEPH has three important consequences: aspects of the dynamic behavior of the program can be derived statically, substantial optimization can be achieved through simple algorithms, and portability is high. The aspects of the dynamic behavior include a check on the use of uninitialized variables and a consistency check on user-declared dynamic properties of rules. The optimizability of ALEPH programs allows the programmer to formulate algorithms with all the elegance inherent in a top-down formulation and nevertheless obtain good machine code.

1. Introduction.

ALEPH is a high-level programming language designed to induce the user to write his programs in a well-structured way. The language is suitable for any problem that suggests top-down analysis (parsers, search algorithms, combinatorial problems, artificial intelligence problems etc.).

An ALEPH program is a top-down description of what is to be done: complex actions are defined in terms of (usually) less complex ones, which in turn are defined in terms of still simpler ones, and so on, until a level is reached at which further decomposition is undesirable.

An ALEPH program consists of a set of such definitions, in a notation not unlike the rules of an affix grammar (Koster [1,2], Crowe [3]). In fact, many of the ideas in ALEPH were derived from the theory of affix grammars; for example, repetition is expressed, not by a goto or while statement but by what in a grammar would be called 'right recursion'.

The syntax and semantics of ALEPH are so simple that it is possible to statically derive interesting properties of the dynamic behavior of the program. For example, the compiler can easily verify that no variable will be used before obtaining a value. Thus the use of uninitialized variables is prevented in a natural way, without resorting to the (dangerous) trick of automatic initialization. Also, the compiler can detect logical constructions that imply what is generally called 'backtrack', and provide a message. The signalling of inadverted backtrack appears to be a powerful weapon against messy programming.

The syntactic simplicity of ALEPH programs can be utilized for a different purpose: optimization. The compiler can transform the program into a directed graph and thereby readily detect recursion, thus permitting a more efficient translation of non-recursive constructs. Furthermore, this directed graph can be used for storage optimization. Thus the programmer can formulate algorithms with all the elegance inherent in a top-down formulation and nevertheless obtain good machine code (probably even more compact than he could have safely written himself).

Because the semantic primitives needed for the translation are small in number and simple in nature ('pass parameter', 'call procedure conditionally', etc.), the transfer of the compiler from one machine to another is quite straightforward. As, however, additional semantic primitives may be defined by the programmer (e.g., multilength arithmetic, 'convert to hash code', or whatever he thinks is a primitive of his problem), the portability of the program (as opposed to that of the compiler) is determined by the portability of these programmer-defined primitives.

The present work is a continuation of the research started by C.H.A. Koster, which resulted in the development of CDL (Compiler Description Language) [4]. His CDL-compiler gave us a great deal of experience with affix-grammar-like languages, from which ALEPH has benefitted.

A two-pass compiler is available and an optimizing two-pass compiler is under construction. These compilers, themselves written in ALEPH, are to a large extent machine-independent. Our versions yield assembly language code for the CDC Cyber Computers.

ALEPH is presently being used for the construction of a machine-independent ALGOL68 compiler.

---

1) Report IW 9/73 of the Dept. of Computer Science, Mathematical Centre, Amsterdam.

It should be borne in mind that this paper is not an ALEPH manual: it does not cover the complete language. Instead a motivated account of the most salient points is given. An ALEPH manual is due to appear in a few months at the Mathematical Centre, Amsterdam, The Netherlands.

2. Criteria.

2.1. Goals.

Our main goals in the design of ALEPH were the following:

a. It must allow good programming at a reasonable effort and a moderate price.

b. Since ALEPH is a tool and not a goal in itself the compiler for it must be simple.

c. To allow the application of the algorithms written in ALEPH on a wide range of machines, the compiler must be portable (as far as possible).

The above requirements were augmented by two more requirements of a more practical nature:

d. Since in our institute ALEPH is mainly intended for compiler writing, sorting algorithms, text-editing, etc., emphasis is on facilitating non-numeric symbolic programming. (Note: this text was justified by a text justifier written in ALEPH).

e. Since it was clear that we shall have to do for a long time to come with early and mid third genereation computing equipment, the compiler must not require any advanced hardware.

2.2. Good programming.

Two different approaches were taken for the effecting of such a vague notion as "good programming". Firstly the literature contains ideas about what constitutes good programming (Dahl [7], Dijkstra [5, 8], Wirth [9], to mention a few); many of these ideas were incorporated. Secondly, we often found it much easier to recognize bad programming and forbid it than to recognize good programming and to promote it.

It is not generally possible to disallow bad programming: a language that is powerful enough to formulate any algorithm in it is also powerful enough to formulate it messily. Nevertheless, it is often possible to make the "desirable" construction more convenient than an "undesirable" one: the usage of a language does not so much depend on its possibilities (it's a Turing machine anyway) as on the conveniency of these possibilities. Although it is perfectly possible to write recursive routines in FORTRAN, hardly anybody ever does so since the administration is just too cumbersome and, conversely but analogously, it is perfectly possible to "jump all over the place" in ALEPH but hardly anybody ever does so since the administration is just too cumbersome.

It should be noted that, surprisingly, it is sometimes possible to forbid bad programming: for example, most high-level languages effectively prevent a jump to data.

2.3. Effort.

We require the "good programming" to be available "at a reasonable effort". Consequently, if a feature that is normally present and useful in programming languages is banished from ALEPH, an acceptable alternative should be present.

2.4. Price.

We also require the "good programming" "at a reasonable price". Since the only way to program a machine efficiently is in hard machine code, we should be willing to accept certain losses for writing in a high-level language. These losses, however, must not depend on the style of programming in such a way as to foster bad programming: for example, in many high-level languages it is more efficient to pass information to subroutines in global variables than in parameters. Consequently, the ALEPH compiler will have to do thorough optimization, and, for simplicty, the constructions in the language should allow easy optimization.

2.5. Simplicity.

The required simplicity of the compiler conflicts with the tendency to make ALEPH as high-level as possible and with the need for extensive optimization. Some trade-off is to be expected here.

2.6. Portability.

The greatest problem in portability is the portability of the object code. Our solution is to produce machine-independent object code of an extremely simple nature. This code can be produced internally and converted directly to pertinent machine code (for production) or can be produced externally and then be converted separately by a simple ad-hoc program (during half-bootstrapping).

2.7. Hardware.

Fancy hardware like virtual memory, hardware stack or microprogramming is not supposed available. Consequently, some fairly elaborate optimizations, like check on non-recursivity, are worth while. Nevertheless the object code could still make good usage of the above advanced features.

3. The Language.

3.1. The grammar form.

It is well known that a grammar is an excellent means for specifying clearly and transparently the input to a program. With the same ease with which we specify a list of numbers separated by commas:

```
input: number, rest numbers option.
rest numbers option:
  comma symbol, number, rest numbers option;
  empty.
```

```
(or, in Backus Normal Form,
<input>::= <number><rest numbers option>
<rest numbers option>::=
  <comma symbol><number><rest numbers option> |
  <empty>                                     )
```

we specify a (seemingly much more complicated) parenthesized tree in infix notation:

```
tree: item;
  open symbol, tree, item, tree, close symbol.
item: letter.
```

It is also well known that under a wide variety of circumstances such a grammar can be considered as a program to read the input described: for reading "input", read a "number" and then read a "rest numbers option". For reading a "rest numbers option", either if there is a comma read it, read a "number" and read a "rest numbers option", or you're done.
Of course there is no reason why a grammar should only be used for the definition of input instructions. The grammar

```
sort: split into two lists, sort first list,
  sort second list, merge.
sort first list: is ordered; sort.
sort second list: is ordered; sort.
```

describes a widely-used sorting technique, or rather a family of these. Here the great value of grammars as a programming device becomes prominent: we are forced first to define the general skeleton of the program in a clear way and then to refine the algorithm by filling in the details in a hierarchical descent. After the above definition the actual forms of "split into two lists", "merge" and "is ordered" are still open. The rule for "is ordered" could decide that only one element is ordered, or even that up to three elements are ordered by straightforward rearrangement, without affecting the basic workings of the algorithm. "split into two lists" could just cut the list in the middle and then "merge" would have to be fairly complicated, or it could split the list into two lists in such a way that all the elements in the first list are smaller than those in the second list, "merge" would then be empty and we would obtain Quicksort (Hoare [10]).

The formulation of algorithms in the form of a grammar has, in the three years of our experience, proven to be an excellent technique for enhancing their well-structuredness.

Once having decided that the grammar-form will be the basis of our language we must answer three fundamental questions. What is the exact flow-of-control? How do rules communicate? And how is the semantics specified for rules that are not further decomposable (terminal symbols)? Furthermore we shall have to provide data types and some input-output, and for the benefit of the user we shall have to add some syntactic sugar.

3.2. The flow-of-control.

From a formal point of view the rule for "tree" mentioned above should be read as: there is a tree either if there is an item or if there is an open-symbol, followed by a tree, an item, another tree and a close-symbol. The flow-of-control suggested by this is obvious: check for the presence of an item and, if that fails, check for a succession of open-symbol, tree, item, tree and close-symbol and if these are not all present then there is no tree. This interpretation is unacceptable since it can only be implemented through the use of automatic backtracking. Moreover, it is not even adequate. Suppose we want to inspect two objects, if they are both integer add them and if they are both character concatenate them:

```
combine:
  is first integer, is second integer, add;
  is first char, is second char, concatenate.
```

If "is first integer" now succeeds and "is second integer" fails, then we are not at all interested in the second alternative and "combine" should fail right away. In our experience the best programs are those in which in all rules the first members of the alternatives succeed under mutually exclusive circumstances. The first member of an alternative can then be considered as the key to that alternative: if the key fits, the other alternatives are no longer of interest. There is a strong analogy here with LL(1)-grammars (Knuth [11]). In an LL(1)-grammar, if the first symbol is present (the first member succeeds) the rest of the alternative is known to be present (further members cannot fail), thereby completely removing the problem of backtrack. The above example, however, shows that this is too stringent a requirement for a computer language since it would effectively forbid the logical conjunction. So we arrive at the following rule for the flow of control: the

first member selects the pertaining alternative, if any; the rule succeeds if all members in the pertaining alternative succeed and it fails if one of these members fails or if no alternative was selected.

As an important consequence there is only one way to reach a given member M in a given alternative A: all first members of alternatives preceding A must have failed and all members in A preceding M must have succeeded. This simple rule is often used in deriving assertions about the program, both mechanically (e.g., check on status in 3.3. and check on left-recursivity) and by hand.

The above interpretation reintroduces the problem of backtrack. However, not all "two questions in a row" give rise to backtrack: in the simple comparison of three numbers:

equal a b and c: equal a and b, equal b and c.

the first member may succeed and the second fail, without requiring backtrack. Therefore rules are divided into two groups, those that effect global changes ("have side-effects") and those that do not. The rule for constructing alternatives is then: once a rule with side-effects has been called, the rest of the alternative must be guaranteed to succeed.

Although the compiler could itself determine whether a rule has side-effects, this is not done. Instead, this information is given by the user and checked by the compiler, as a form of useful redundancy. Often a conceptual error results in a rule that was thought to be free of side-effects having side-effects.

In the above, rules are used to decide the presence of the desribed constructs, e.g., trees (and, possibly, to process them). In many cases, however, the programmer knows that the construct is present: the tree must be present or something is wrong:

```
tree: item;
  open symbol, tree, item, tree, close symbol;
  error message.
```

Rules are again divided into two groups: those that can fail and those that always succeed. As before, the compiler could find this out, but for reasons explained above the programmer specifies his opinion on the rule, which opinion is then checked by the compiler. And again, this form of redundancy proves to be very useful.

The two division criteria can be combined, yielding four groups:

```
can fail,     has side-effects:    'predicate'
can fail,     has no side-effects: 'question'
cannot fail,  has side-effects:    'action'
cannot fail,  has no side-effects: 'function'
```

In this terminology "tree" should be an 'action'. Now the item between the two trees may be missing, so a programmer might write:

```
'action' tree: item;
    open symbol, tree, rest tree;
    error message.
'action' rest tree:
    item, tree, close symbol; error message.
```

The compiler would find two errors (given suitable definitions for the missing rules). "Rest tree" is not an 'action', and the "close symbol" causes backtrack (over tree and item). These two errors, admittedly simple as they are, would probably not be detected in most conventional programming languages and would be called by many people "conceptual errors" rather than "syntactic errors".

3.3. The parameter mechanism.

All the above grammars are context-free and as such they are inadequate to express actual algorithms. What is needed is a way of communication between the notions in a rule. Formally such a way is provided by the affixes in an affix-grammar (Koster [2]): ALEPH uses a parameter mechanism that is very much akin.

All formal variables (parameters) are local to the rule they belong to, as are the local variables. Some formal variables are prefilled at call entry with the values of the actual parameters (corresponding to ι-bound-affixes), some formal variables are still uninitialized at call entry but their values will be used by the calling rule (corresponding to δ-bound-affixes), and some are both (not corresponding to an affix type). All local variables are uninitialized at call entry. The rule is obliged to give values to those parameters that will be used by the caller. However, if the rule fails, the caller will never need these values: they will not even be passed back at call exit, so that in that case the rule does not have to provide them. This "copy-maybe-restore" mechanism has the advantages of the standard "copy-restore" (transparency, efficiency of parameter access, no machine addresses on the stack) and moreover provides a one-level backtrack free of charge: a rule may tentatively mess up its parameters, and if it then decides to fail, nothing needs to be restored (since only copies were spoiled).

Since the status (initialized or not) of all formal and local variables is known at call entry; since this status before the execution of a member, together with the parameter description of that member, determines the status after the execution; and since there is only one way to reach a certain point in a rule, the compiler can readily construct the status at all points and perform a reliable check on the use of uninitialized variables.
This again proves to be very helpful in detecting (logical) errors.

For an example we return to the list of numbers separated by commas mentioned in 3.1., and we suppose that we want to read them, add them and print the sum:

```
'action' input — res:
  number + res, rest numbers option + res,
    result + res.
'action' rest numbers option + >res> — nmb:
  comma symbol, number + nmb, sum + nmb + res,
    rest numbers option + res;
  +.
'action' number + res>:
  get int + input file + res;
  error + bad number, 0 -> res.
'action' sum + >x + >y>:
  add + x + y + y; error + overflow.
```

The pluses affix the affixes to the rules. Co-ordering with pluses is used rather than sub-ordering with parentheses. The use of parentheses would have implied the possibility of nesting: this nesting, however, is not allowed. Moreover, parentheses are already being used extensively in a different way (see 3.6.1.).
The minus signals a local variable. The right arrow-head in front of "res" indicates that "res" will be prefilled, the one at the back of "res" indicates that after the call the value will be returned to the caller.
The local variable "res" is uninitialized at the colon in "input", from the declaration of "number" it follows that it will not use the value of "res" (which would have been illegal) but will return a value to it. So, at the first comma "res" is initialised and may be affixed to "rest numbers option" which uses its value.

The above notation precludes the introduction of operators and type procedures in ALEPH, and in fact they do not exist in ALEPH. Although we readily concede that operators and type procedures often allow a very elegant formulation of an algorithm, we also feel that they tend to lead to unjustifiable simplifications. By the nature of it, an operator or type procedure yields only one result (if we disregard messy tricks). Now, it is doubtful if, e.g., the result of the inversion of a matrix can be expressed in one matrix, and it is simply not true that the result of the addition of two integers can be expressed in one integer (since overflow may occur). Especially the latter fact is poorly appreciated both in high-level languages and in hardware. In the worst case what is called the "add-instruction" is in fact a bit-shuffler that happens to yield the sum in about 75 percent of the cases. In a slightly better case the program comes to a grinding halt or some pre-attached program is called, with all the misery inherent in interrupts. In fact there is no add instruction: all there is is an add request, which, like any other request, can fail to be satisfied and which is a 'question' in the sense of the above. This is correctly recognized by that hardware that sets an overflow bit, which bit is then, more often than not, boldly ignored by the high-level language.

There are a few requests that can always be fulfilled: e.g., it is always possible to set one variable equal to the value of the other. Indeed the assignment is written with the aid of an operator: "0 -> res" in the example above. Note that this instruction is necessary to sustain the claim that "number" always assign a value to its formal variable "res>": we are not allowed to let the program carry on with a "ghost" value, even after an error-message.

3.4. Primitive rules.
Rules are specified by their decomposition into other rules. This process must end somewhere; it can end in one of three ways:
a. The required action is a primitive of ALEPH, e.g., assignment.
b. The required action is known to the compiler under a standard name, e.g., the 'predicate' "get int" and the 'question' "add" in the example above.
c. The required action is part of the problem but cannot be decomposed (e.g., the activation of particular hardware) or must be described on a lower level for reasons of efficiency (e.g., the calculation of a hash address from a given string).

In cases a) and b) there is no problem for the user and only a one-time problem in transferring to another machine: the primitives must be reprogrammed. Case c) is exceedingly rare but must be catered for. Rules can be declared 'external' in ALEPH under specification of the parameters and the concerning semantics must be supplied by external means, e.g., at the level of machine code (in which case, of course, there is no portability).

3.5. Data types.
The language defined so far does not rely in any way on the properties of the data types (except perhaps that rules as data would be inconvenient and would violate simplicity requirements). We are still at liberty to define the data types we need. For our applications and for reasons of simplicity we have resticted ourselves to integer data (already introduced above) and stacks of these. The latter have the usual property that top elements may be added, inspected and removed. In addition, they have the following properties:
a. All elements can be reached, thus the stack can act as an array. Arrays in the standard sense cannot be allowed since they may contain mixed initialized and uninitialized variables.
b. Bottom elements can be removed, thus the stack can act as a queue. If the queue walks out of physical memory it is simply pushed back by the runtime system and since all references to a stack go through its base address only this base address needs to be updated. Bottom elements cannot be added: a deque (Knuth [12]) is much more complicated to implement, is hardly ever useful and in emergencies can be simulated by two queues.
c. Each stack has its own private piece of the virtual address space (which in total extends from minus the maximum integer to plus the maximum integer), so that if an integer is used as an index to a stack, it identifies that stack. Thus dynamically complicated objects can be efficiently unraveled by extracting stack identification from the given index.

The above data types are easy to implement and constitute very convenient tools for data handling that have proved their value in practice, especially in combination with data-description-like rules for the processing of data. For example, a list (in "list stack") whose elements consist of items (called "item") and indices to the next element (called "next") is processed by:

```
'action' list + >handle:
  process + item×list stack[handle],
    rest list + next×list stack[handle].
'action' rest list + >handle:
  was + list stack + handle, list + handle; +.
```

where "process" must be given by the user and "was" is a 'question' known to the compiler which tests whether "handle" is an index to "list stack" (if it fails there are no more elements).

Although these data types are safer than the usual data types in languages (all reachable variables have a value and most logical errors are caught immediately by indices being applied to the wrong stack), they unfortuantely lack the rigour and reliability of the flow-of-control explained in 3.2. and 3.3. (runtime checking is still necessary and the "dangling reference" problem is not solved). The reason is simply that the state of the art in grammars and in hierarchical programming is much more advanced than that in data structures. Even the presently most advanced data structures, those of ALGOL 68 (van Wijngaarden [13]) cannot be grafted in a simple way to ALEPH: we would lose the advantages mentioned above, the ALGOL 68 solution to the "dangling reference" problem (scope checking) still needs dynamic checking and is not readily applicable to ALEPH, and indices can still be out of bounds. We hope and expect that many of these trouble-spots can be mended in the near future.

3.6. Syntactic sugar.
In 3.6.1. and 3.6.2. some examples are given of features solely intended to make the language more convenient to use.

3.6.1. Flow-of-control.
When we read the short program given in 3.3, we can easily see that it is overly recursive. The recursive call of "rest numbers option" in "rest numbers option" puts a copy "res'" of "res" on the stack, works on "res'" and then restores "res" to "res'": it could as well have worked on "res" directly. Moreover, the said call puts a return link on the return link stack that points directly to a "return over return link stack" instruction (since "res" needs no longer be restored and the present call is the last one in an alternative) so it could as well be left out. All that is left of the call is the (re)-activation of "rest numbers option" and as such it corresponds to a simple and clean jump. The user is allowed to write:

```
'action' rest numbers option + >res> - nmb:
  comma symbol, number + nmb, sum + nmb + res,
    :rest numbers option; +.
```

Conversely, he may use the jump only as a last member of an alternative in an 'action' or 'function' and it is then considered shorthand for a recursive call with the same parameters as the original.

Although the compiler would have found this optimization, the user, by indicating this simplification himself, has gained something: "rest numbers option" is now only called in one place, in "input", and can be substituted there. The same holds for "sum", so that the program reduces to:

```
'action' input — res: number + res,
  rest numbers option — nmb:
   (comma symbol, number + nmb,
      sum:
       (add + nmb + res + res;
        error + overflow),
      : rest numbers option;
    +),
  result + res.

'action' number + res>:
  get int + input file + res;
  error + bad number, 0 -> res.
```

3.6.2. Data types.

In addition to formal and local variables ALEPH allows global variables. Although we are aware of their undesirability and of the great opprotunities they afford in bad programming (Wulf, Shaw [6]), we do not see a way to do without them in the present framework. Some information (like, e.g., a character counter on the input in a compiler) must eventually be available to virtually all rules (since, again in a compiler, virtually all rules can cause a call to the error-routine which prints a diagnostic message including said character counter). Consequently, this information must be passed as a parameter to all these rules. The same in essence applies to all I/O information. By way of experiment we rewrote a fair-sized ALEPH program (concerning mode-handling in ALGOL 68) under the elimination of global variables (except I/O information) and found that the average number of affixes per rule went up from 1.5 to 4.5. We consider this too high a price: only a profoundly different approach to data types may yield a solution.
It should be noted, however, that the misuse of global variables is limited by their tendency to cause backtrack errors upon careless handling.

Global variables must be initialized upon decalration. Their values can be changed by any rule. It is also possible to declare initialized constants whose values cannot be changed. Aside from the convenience of this feature it also aids in good programming. It appears that the occurrence of a hard integer denotation in a rule is generally unjustified. Tallying hard integers in some sample programs has taught us that only roughly 1 in 50 integers is used in its integer meaning. For the rest they were either variables of the problem that happened to be constant most of the time (like linewidth of the printer, number of bits in a character, etc.) or terminators in data structures where "nil" should have been used. We seriously contemplate disallowing hard integers in rules and only allowing them in initializations.

4. References.

[1] Koster, C.H.A., On the construction of ALGOL-procedures for generating, analysing and translating sentences in natural languages, MR 72, Mathematical Centre, Amsterdam (1965).

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

[3] Crowe, D., Generating Parsers for Affix Grammars, Comm. ACM 15, 728-734 (1972).

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

[5] Dijkstra, E.W., Notes on Structured Programming, Rep 70 Wsk 03, Math. Dept. Technical University, Eindhoven (1970).

[6] Wulf, W., "Global Variable Considered Harmful", SIGPLAN Notices 8 (2), 28-34 (1972).

[7] Dahl, O-J., Dijkstra, E.W., Hoare, C.A.R., Structured Programming, Academic Press, London (1972).

[8] Dijkstra, E.W., Go To Statement Considered Harmful, Comm. ACM 11(3), 147 (1968).

[9] Wirth, N., Program development by stepwise refinement, Comm. ACM 14(4), 221 (1971).

[10] Hoare, C.A.R., "Quicksort", Computer J. 5 (1), 10-15 (1962).

[11] Knuth, D.E., Top-down syntactic analysis, Acta Informatica 1, 79-110 (1971).

[12] Knuth, D.E., The Art of Computer Programming, Vol I, pp. 235-239, Addison-Wesley, London (1969).

[13] van Wijngaarden, A. (ed.), Report on the Algorithmic Langauge ALGOL 68, Numer. Math. 14, 79-218 (1969).