The use of ALGOL 68 for trees

H. D. Baecker*

* *Department of Mathematics, Statistics, and Computing Science, The University of Calgary, Calgary 44, Alberta, Canada*

These notes discuss the use of a binary tree programmed in ALGOL 68 and of a tree-structured reference system to the local definitions of ALGOL 68 syntactic elements. Sample procedures for tree manipulation in ALGOL 68 are appended.

(Received January 1969)

Introduction

A variety of techniques have been developed for the construction of symbol tables in the early passes of compilers. Representative discussions of the merits and demerits of the techniques will be found in Wegner (1968) and Mailloux (1968).

The following notes illustrate the possibility of using tree structures to hold symbol tables, sample procedures in ALGOL 68 being appended to illustrate the techniques. In the absence of a working ALGOL 68 compiler at this date (January 1969) the procedures must be regarded as completely unproven.

It is assumed that the purpose of the initial pass or passes of a compiler is twofold; to abstract and codify the declarations appearing in the text, for later application; and to reduce the source text to a stream of codified units of uniform size, each representing an identifier, denotation, operator, etc.

ALGOL 68 declarations

In ALGOL 68 any identifier or indicant, even those belonging to the standard or library preludes, may be redefined anew once in each range. Thus a symbol table must cater for unpredictable multiple occurrences and must provide means for discovering the relevant definition at any applied occurrence of an object. As a single-pass translator for ALGOL 68 is unlikely to be achieved we must hold nested definitions throughout.

Unfortunately the structure of ALGOL 68 makes it impossible to attribute a meaning to a mode identifier declaration before the defining occurrences of mode-indications have been found, so we need two or three passes to accomplish the objectives for the initial passes stated in the Introduction. We shall therefore reduce the source text to a codified string on the first pass. Then the symbol table built up during the first pass will be inspected for identifiers, indicants, etc., defined in the standard and library preludes, which are deemed to exist in range 0. Any such object found during this inspection will have its definition pulled out from a library and made available prior to the second pass, which will detect the defining occurrences of mode, priority, and operator indications, whilst a third pass finds the defining occurrences of identifiers.

Program example

Appendix 1 lists a set of ALGOL 68 procedures, and nonlocal defining occurrences, for manipulating a binary tree that will serve as a symbol table during the first pass of an ALGOL 68 compiler. The structure *definition*, undefined itself herein, holds the information garnered from defining occurrences in the standard prelude and source text, to be made available to later passes of the compiler. Access to these definitions is via the reference array *ITEMS*. For purposes of exposition assume that the block labelled *PASSONE* is overlaid for subsequent passes of the compiler.

Range determination is according to the scheme outlined by Mailloux (1968) for ALGOL 68. At first occurrence in the source text each object is allocated a sequential *accession number* which subsequently serves as a subscript when the definition of an occurrence is sought via the vector *ITEMS*. The latest accession number to be allocated is held in *item*. The appropriate definition for an object is thus found by using the accession number and the range of the occurrence.

The string *word* holds the most recent object to have been assembled by the input routine from the source text. This may serve as a search key at an occurrence or as both search key (for positioning in the tree) and an initialisation value for a new instance of the structure *object* which forms a new tree element. The other fields of *object* are two reference pointers to possible further instances of *object* and two integers, the accession number and kind of the object. Upon a use of that object its accession number is substituted in the output stream.

The kernel procedure is *find*. Using the value of *word* it searches through the tree by comparison with the *name* of *object* encountered. On a low comparison it branches *left*, on high to *right*. If the selected branch is *nil* a terminal node of the tree has been reached and the sought object is not in the tree. When insertion of a new object is to be performed then *next* refers to the instance of *object* from which it is to be appended. If an object has been matched in the tree then *next* refers to the instance of *object* bearing that *name*. The tree is rooted at *root*.

The integer *what* holds the kind of *word*, e.g. identifier, indicant, denotation, separator.

Posting definitions

When the source text has been scanned and transformed the procedure *cull* is invoked to post denotations to *ITEMS* and to find identifiers, indicants, operators; etc., that are defined in range 0, that is, in the standard and library preludes. These are posted to *ITEMS*, the size of which is determined by the highest accession number allocated this time.

It is clear that *definition* is a *union*, to cater for the variety of definitions applicable to various types of objects. As an example we give the posting of a denotation to *ITEMS* using the procedure *denotes* (called from *cull*).

Source definitions and applications

Subsequent passes will detect, form, and post definitions occurring in the source text. Once a definition structure for an item has been formed it is posted to *ITEMS* by a call to *post*. This procedure is complicated by the desire to establish the lists dependent from *ITEMS* in ascending order of range number, whilst catering for the possibility that definitions may occur anywhere within a range.

The recursive procedure *seek* will return a reference to the definition applicable within the current range, if none exists it returns the value *nil*.

Conclusion

The main object of these notes has been to explore the facilities of ALGOL 68 as a tree manipulation language, a purpose which it would seem to satisfy well. As a method of handling the syntax of ALGOL 68 the scheme presented here is crude and has already been refined in many respects.

Perhaps the most important conclusion, not demonstrated herein, is that the facilities of ALGOL 68 appear to lend themselves to writing an ALGOL 68 compiler in itself, which will greatly facilitate the propagation of the language.

Acknowledgement

I wish to thank my colleague J. E. L. Peck for his great help in the preparation of these notes.

Appendix 1

```
int range number, max range, range depth;
[1: flex] int current ranges;
struct link = (ref link next, ref definition definition, int
                range);
struct item = (int kind, ref link first);
[1: flex] item ITEMS;
   proc uprange =
      (: range number := current ranges [range depth plus 1]
                       := max range plus 1);
   proc downrange =
      (: range number := current ranges [range depth
         minus 1]);

PASSONE: begin pragma overlay pragma
int item := 0, what;
string word;
struct object = (string name, ref object left, right,
                 int accession number, kind);
ref object root := nil;
  proc find = int:
    begin ref ref object next := root;
    while (ref object : next :≠: nil | word ≠ name of
         next | false) do
       next := (word < name of next | left of next | right
         of next);
    if (ref object : next) :=: nil then
       (ref ref object : next) := object :=
         (word, nil, nil, item plus 1, what) fi;
    accession number of next
    end co find co;
  proc cull = (ref object next) :
    begin
    if (ref object : left of next) :≠: nil then
         cull (left of next) fi;
    comment here insert the case clause (depending on kind
       of next) to insert denotations in ITEMS and to
       search the preludes for definitions for ITEMS
       comment
    if (ref object : right of next) :≠: nil then
         cull (right of next) fi
    end;
  proc denotes = (ref object next) :
    begin string heap hold := name of next ;
    ITEMS [accession number of next] :=
       (kind of next, link := (nil, hold, 0))
    end;
comment here comes the body of PASSONE where calls
       to find are made after each word has been assembled
       from the source text comment
ITEMS := [1 : item] item; comment assign actual value
       to flexible upper bound comment
for i to item do first of ITEMS [i] := nil;
cull (root);
end;

PASSTWO : begin pragma overlay pragma
  proc post = (int accession number, ref definition
       definition) :
    begin ref item the item = ITEMS[accession number];
    ref int the type = type of the item,
    ref ref link next := first of the item,
    link heap hold := (nil, definition, range number),
    bool more ;
    CHECK TYPE :
    if (ref link : next) :=: nil
    then the type := input type
    elsf the type ≠ input type
    then go to INCOMPATIBLE TYPE
    fi ;
    NOW INSERT LINK :
    while (more := (ref link : next) :≠: nil |
       range of next ⩽ range number | false) do
         next := next of next ;
    if more then next of hold := next fi ;
    (ref ref link : next) := hold
    end co post co ;
comment here follows the meaty body of PASSTWO which
       will call post upon the successful elucidation of mode
       and operator definitions comment
end ;

PASSTHREE : begin pragma overlay pragma
proc post = comment as in PASSTWO comment
  proc seek = (ref link where) ref definition :
```

```
begin ref definition j := nil;
    if (ref link : where) :≠: nil ∧ range of where ⩽
        range number
    then j := seek (next of where) fi ;
  if (ref definition :j) :=: nil then for i to range depth
        do
    if current ranges [i] = range of where then
    j := definition of where ; go to out fi fi ;
out : j end ;
comment an initial call to seek is always of the form :
    seek (first of ITEMS[accession number]) comment
comment here comes the mighty meaty body of PASS-
    THREE, including calls to seek to determine modes
    upon the elucidation of identifier definitions and a call
    to post to hang each elucidated definition from
    ITEMS comment
end ;
comment here follow subsequent passes comment
```

References

Mailloux, B. J. (1968). *On the Implementation of ALGOL 68*, Amsterdam: Mathematisch Centrum.

Wegner, P. (1968). *Programming Languages, Information Structures, and Machine Organisation*, New York: McGraw-Hill Book Co.

Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A. (1969). *Draft Report on the Algorithmic Language ALGOL 68*, MR 100, Amsterdam: Mathematisch Centrum.

Book review

Machine Intelligence 4, by Ed. B. Meltzer and D. Michie, 1969; 508 pages. (*Edinburgh University Press*, 100s.)

The Annual Machine Intelligence Workshops held at Edinburgh University show how dedicated organisers can keep up high standards year after year. This record of the 1968 Workshop has 26 contributions, a similar number to that of the 1967 meeting; yet, the majority of authors are newcomers to the Workshops, and the standard, if at all changed, seems to be rising. This year there is one dominant impression, and that is the marked swing towards the use of more precise technical methods. The hopeful optimism that once felt that a few simple heuristics backed up by machine power could solve significant problems has given way to a realisation that Machine Intelligence might have to rely on difficult techniques. This can be seen most clearly in the applications papers. Here there are two trends, one going towards the use of very special features of the problem, and another attempting to use formal theorem proving techniques. The first trend is seen in the latest of the series of papers by Miss C. J. Hilditch on pattern recognition, and also in the paper by Buchanan, Sutherland and Feigenbaum, which is entitled 'Heuristic Dendral'. This latter paper describes a program for generating hypotheses for the structure of organic chemicals using experimental data, whilst the program is called heuristic, it actually uses quite complex special knowledge. Amongst the papers proposing the application of theorem proving are those of Darlington on the possible use of theorem proving in information retrieval, and Green on its use in a question answering system. The whole relationship between logic and problem solving by machine is considered at length in a profound, and stimulating, paper by McCarthy and Hayes, this points out that the program which carries out the problem solution is itself open to logical analysis. This paper is likely to become required reading in the field.

The solid central core of the 1968 Workshop is in the half-dozen papers on theorem proving. Almost all techniques in use today rely on Herbrand's theorem with steps decided by methods related to J. A. Robinson's Resolution Principle. Perhaps it is worth noting that the accumulated papers on mechanical theorem proving in the Machine Intelligence series of volumes are a unique reference source, since they give a comprehensive coverage of existing methods.

Programming techniques have often benefitted as a side-effect of work in Artificial Intelligence. Today this connection is being exploited in a much more systematic way. For example, Foster and Elcock are implementing a high level language based on logical assertions, rather than instruction steps.

There is the usual sprinkling of new ideas which might blossom in the future. Two examples of this are the proposal of J. A. Robinson on the mechanisation of higher order logic, and the algebraic analysis of program structure put forward by Burstall and Landin. Altogether, this book is not one to be left on the library shelf, rather it is one to be kept in constant use by an individual, a very rare thing for the proceedings of a conference.

J. J. Florentin (London)