A NOTE ON LIST-PROCESSING

IN ALGOL 68

Y. Wallach, D. Andermann

Technical Report No. 8

July 1970
A Note on List-Processing In Algol 68

Abstract:

This paper intends to show the possibilities of list processing in Algol 68. Examples are given of simulation of Lisp in Algol 68 and construction of trees and their evaluation.

Simulation of Lisp

In order to simulate Lisp we declare two modes:

\[ \text{mode } \text{tom} = \text{string}; \]  
\[ \text{mode } \text{list} = \text{struct (union (ref tom, tom, ref list) head, ref list tail)}; \]

Thus whenever we declare say:

\[ \text{tom } A; \]

we have really set up a pointer to a string where "atomic" information is stored.

The declaration \[ \text{list } B; \] creates a pointer of mode \[ \text{ref list} \] to a cell where (1) points either to a \[ \text{tom} \] or to another list cell or is a \[ \text{tom} \].

The pointer (2) points to another list - cell.

The NIL of Lisp will be defined as a denotation of mode \[ \text{list} \] by the following identity declaration:

\[ \text{list } \text{NIL} = ("NIL", \text{nil}); \]
In order to build up lists we declare the following procedure:

\[
\text{proc CONS = (ref union (com, list) A, list B) ref list:}
\]

\[
(\text{list z; head of z : = A;}
\]

\[
\text{tail of z : = B; z});
\]

This again sets up a list cell and connects

its first element to a \text{com} or a \text{list} called \text{A}

and the second to \text{list} \text{B} and returns a pointer (ref list);

\[
\text{fig 3}
\]

Thus invoking the procedure CONS as in;

\[
z1 : = \text{CONS (A, NIL)};
\]

creates the structure on the right or in Lisp-notation this is equivalent to (A).

\[
\text{fig 4}
\]

In the same way, the following call:

\[
y : = \text{CONS (A, CONS(B, CONS (C, CONS (D, NIL)))})
\]

produces the following structure:

\[
\text{fig. 5}
\]

called \text{(A,B,C,D)} in Lisp.

Finally, if two lists: \text{Y = (A, B)}

and \text{z = (C, D)} are to be connected, this is done

by invoking:

\[
X : = \text{CONS (Y, z)};
\]
In order to dissect a given list into its first element and the remaining list, we define the following two procedures:

\[
\text{proc TL} = (\text{list } P) \text{ list: tail of } P; \\
\text{proc HD} = (\text{list } P) \text{ list: head of } P;
\]

Applying these procedures to the various figures would result in:

**fig. 1:** The result is undefined since the actual parameter is not of mode list.

**fig. 4:** TL (z1) yields the pointer to NIL, HD (z1) yields the pointer to atom A.

**fig. 5** HD (Y) yields the pointer to the atom A, TL (Y) yields the pointer to the list (B,C,D).

**fig. 6** HD (x) and TL (x) yield pointers to the lists Y and z respectively.

Since every procedure in Algol 68 is defined as a recursive procedure, they may be nested.

Thus for the list of figure 5 invoking:

HD (TL(TL(Y))) yields C whereas

TL (TL(TL(TL(Y)))) yields the pointer to NIL.
The remaining three primitive functions of Lisp may be simulated by the following procedures:

\[
\begin{align*}
\text{proc ATOM} &= (\text{union} \ (\text{list}, \ \text{tom}) \ A) \ \text{bool} : \ \text{tom} :: A; \\
\text{proc EQ} &= (\text{tom} \ x, \ y) \ \text{bool} : X = Y; \\
\text{proc NULL} &= (\text{ref} \ \text{list} \ x) \ \text{bool} : \ \text{if} \ \text{HD} (x) = \text{NIL} \\
\text{then} \ \text{true} \ \text{else} \ \text{false} \ \text{fi};
\end{align*}
\]

**Example:**

Suppose the following list is given:

\[X = (((A, B), C), D, E)\]

This list is represented in figure 7.

The function \(Ff(X)\) is supposed to find the first atom of a list \(x\), ignoring all brackets. It may be defined in the following way:

\[
\begin{align*}
\text{proc Ff} &= (\text{list} \ x) \ \text{ref} \ \text{tom} : \ \text{if} \ \text{ATOM} (x) \\
\text{then} \ X \ \text{else} \ \text{Ff} (\ \text{HD}(X)) \ \text{fi};
\end{align*}
\]

calling \(\text{Ff}(x)\) where \(x\) is as in figure 7 would result in the call-chain:

\[\text{Ff}(x) \rightarrow \text{Ff}(y) \rightarrow \text{Ff}(z) \rightarrow A \text{ being the final result.}\]

**The Construction and Evaluation of binary trees.**

Let us now take an example, which deals with evaluating an arithmetic expression given in prefixed ("polish") notation.
This procedure in Algol 68 would look like:

```plaintext
proc EVALUATE = (union (tom, list) L) real :
    begin list M, N;
        if Atom (L) then L else
        M = HD (HD (TL(L))) ; N = HD(TL(HD(TL(L))));
        if HD(L)=="+" then EVALUATE (M) + EVALUATE (N)
        elsif HD(L)=="-" then EVALUATE (M) - EVALUATE (N)
        elsif HD(L)=="*" then EVALUATE (M) * EVALUATE (N)
        elsif HD(L)=="/" then EVALUATE (M) / EVALUATE (N)
        fi ; fi ;
    end ;
```

This is not general enough and besides we may ask whether we have used the facilities of Algol-68 to best advantage. The answer seems to be negative. Algebraic expressions, when rewritten in prefixed i.e. parenthesisless form are best pictured as a binary tree. This may be seen for the expression 15 / ( a - b ) or in prefixed form: / 15 - a b in figure 8.

---

```
fig. 8
```

---
The mode we now need is one with the information stored in four fields:

a) The operator / operand field.
b) The left pointer field.
c) The right pointer field.
d) The type of the node.

This mode will be declared as follows:

\[
\text{mode node} = \text{struct (ref union (string, real) inf, ref node l, r, int ch)}; \quad (16)
\]

The field \(ch\) has the following meaning:

a) If the value is 1 the node represents a constant.
b) If the value is 2 the node represents a variable.
c) If the value is 3 the node represents an operator.

We will need also a special node which will show that the symbol is terminal - let's call it "t".

In order to build a tree corresponding to figure 8, we shall declare:

\[
\text{proc TREE = (union (string, real) S, ref node LF, RT, int J) ref node :}
\]

\[
\begin{align*}
\text{node M ; inf of M :} & = S ; \\
\text{l of M :} & = LF; \text{ r of M :} = RT; \\
\text{ch of M :} & = J; M \\
\end{align*}
\]

end ;

and invoke it repeatedly:

\[
\begin{align*}
\text{node bb, aa, cc, dd, all;}
\text{bb:} & = \text{TREE ("b", t, t, 2);}
\text{aa:} & = \text{TREE ("a", t, t, 2);}
\text{cc:} & = \text{TREE ("-", aa, bb, 3);}
\text{dd:} & = \text{TREE (25 , t, t, 1);}
\text{all:} & = \text{TREE ("/", dd, cc, 3);}
\end{align*}
\]
The built structure is represented in figure 9.

The same result could be produced without the intermediate pointers
aa, bb, cc, dd as in:
all := TREE("/", TREE("25", t, t, 1), TREE("-", TREE("a", t, t, 2),
TREE("b", t, t, 2), 3), 3);

Both programs produce the data structure of figure 9.
Having produced a tree we might return to the problem of evaluating it, say by a
procedure call such as:
RESULT := EVAL (all);

The function "EVAL" will be declared as follows:
proc EVAL = (ref node P, [1:] real z) real:
begin real X, Y, node T := P;
case ch of T in inf of T, if inf of T = "a" then z [1]
elsef inf of T = "b" then z [2]
else z [3]
out X := EVAL (1 of T, z) ; Y := EVAL (r of T, z);
esac ;
If inf of T = "4" then X+Y
elsif inf of T = "-" then X-Y
elsif inf of T = "*" then X*Y
elsif inf of T = "/" then X/Y
else 0 fi: end

If we now examine what happens when we call: EVAL (all);
we see that the function is invoked recursively when the field ch of the node
has the value 3 (i.e. it is an operator), thereby causing the evaluation of the
left and right branches.
The procedure has two parameters, one being the pointer to the root of the
tree and the other an array with unspecified upper bound where the value of
variables are stored.

Conclusion

An essay was done to show the use of Algol 68 in simulating Lisp, constructing
binary trees and their evaluation. The procedures were not tested on an actual
computer because of the lack of an Algol 68 compiler, but were tested manually
by the authors according to their understanding of [1].
BIBLIOGRAPHY

1. A. Van Wijngaarden (Editor), B.J. Mailloux, J. E. L. Peck, C. H. A. Koster:
   Report on the Algorithmic language