22

Writing Search Algorithms in Functional Form

R. M. Burstall
Department of Machine Intelligence and Perception
University of Edinburgh

1. INTRODUCTION

A great many machine intelligence programs perform some sort of search, and a number of investigators have discussed techniques of searching trees.

One well known method, which we will call 'depth-first' search is particularly attractive because it is very easy to program by recursion. The resulting programs are short and rather transparent. It has the disadvantage however that it is by no means the most efficient sequence of searching a tree in many cases. Indeed if the tree is infinite it may cause the search to go right out of control. The attractiveness of 'depth-first' recursive searching was pointed out a long time ago by Newell, Shaw, and Simon, but they also remarked on its pitfalls. More recently Slagle in his 'Deducom' program (*see* Slagle, 1965) used a 'depth-first' recursive search for convenience, but found that it led to severe limitations on the problem-solving ability of his program. A number of examples of problems amenable to tree search methods are given by Golomb and Baumert (1965).

Most machine intelligence tasks are difficult enough to make ease and transparency of programming an important consideration. Hence it would be interesting to discover a programming technique which would preserve the simple program structure associated with recursive depth-first search while allowing a more efficient and flexible sequence of searching. This paper puts forward such a technique. The search method in question which we will call 'controlled' search is that used by the Doran and Michie 'Graph Traverser' program (*see* Doran, 1967, 1968; Doran and Michie, 1966; Michie, 1967).

We first consider different methods of searching, quite apart from the question of programming. We then describe the recursive programming technique for depth-first searching and discuss the possibility of introducing a new form of expression into programming languages to enable us to treat

more general searches without restructuring the program. It turns out that this new form of expression is not really a special extension of programming languages but can be defined in terms of Landin's generalized jump operator. This is much more satisfactory.

2. SEARCH TECHNIQUES

We will consider the problem of searching a tree to find a node with a given property. If it is necessary to search the whole tree, e.g. to find the node which is optimal in some sense, the sequence in which the search is carried out is unimportant. If however we are content to discover just one node with the given property the sequence in which the nodes are searched may be of great interest. We will consider three possibilities and illustrate them using the following simple problem.

The tree is defined in terms of two functions of integers

$$f(n)=n^2-2n+3$$
$$g(n)=2n^2-5n+4$$

The problem is to find a value of n with some property p by repeatedly applying either f or g to a starting value n_0. For example

$$n_0 = 0$$
$$p(n)=30<n\leqslant 40$$

Part of the corresponding search tree is shown in figure 1.

2.1. Depth-first search

To search a tree starting from a node n by depth-first search:

1. Check whether n itself has the required property. If so the search terminates successfully;
2. If n has no successors the search terminates unsuccessfully;
3. Take each successor of n in some specified sequence and perform a depth-first search starting from that node. Continue until one of the successors leads to a successful search.

The sequence of search for the problem stated above is shown in figure 2, making the assumption that the tree is restricted to a finite one by ignoring all nodes greater than 500, and taking $f(n)$ before $g(n)$.

If this restriction is removed the search fails completely as only the topmost branch of the tree will ever be explored.

The effectiveness of a depth-first search can be improved by choosing a suitable sequence for taking the successors of a given node, e.g. smallest first. But this will not remedy the inability of a depth-first search of an infinite tree to recover from a single wrong decision.

$f(n)=n^2-2n+3$
$g(n)=2n^2-5n+4$
$p(n)=30<n\leqslant 40$
$n=38$ satisfies $p(n)$

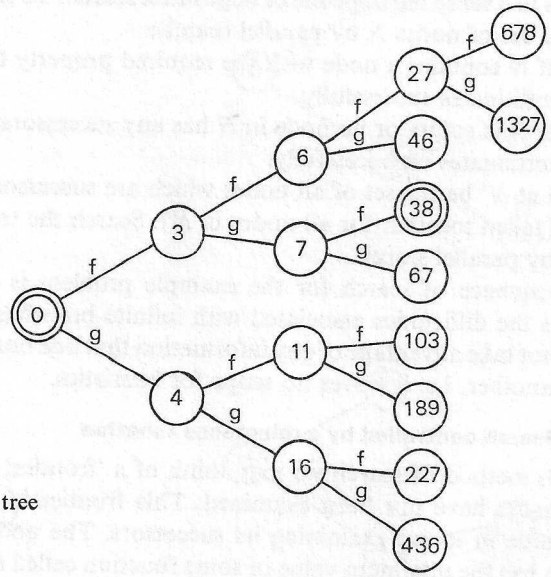

Figure 1. The search tree

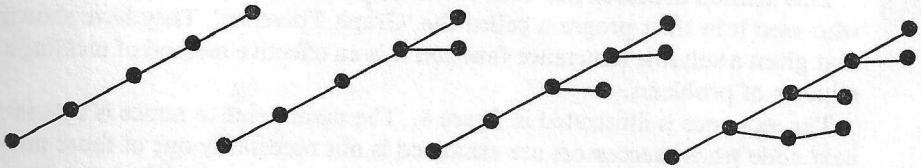

Figure 2. Stages in depth-first search

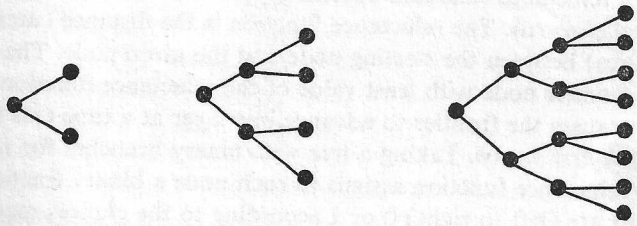

Figure 3. Stages in parallel search

2.2. Parallel search

This is in a sense the opposite of depth-first search. To search the trees starting from a set of nodes N by parallel search:

1. If N contains a node with the required property the search terminates successfully.
2. If N is empty or no node in N has any successors the search terminates unsuccessfully.
3. Let N' be the set of all nodes which are successors of a node in N (taken together for all nodes in N). Search the trees starting from N' by parallel search.

The sequence of search for the example problem is shown in figure 3. It avoids the difficulties associated with infinite branches but is very rigid and does not take advantage of any information that one branch is more promising than another, i.e. it leaves no scope for heuristics.

2.3. Search controlled by a reluctance function

In this method of search we may think of a 'frontier' of those nodes whose successors have not been examined. This frontier is extended by choosing any node in it and examining its successors. The node chosen may be that which has the minimum value of some function called a 'reluctance function', normally chosen on heuristic grounds. (Doran and Michie call this an 'evaluation function' but we use the word 'evaluate' here in another sense which would lead to confusion.)

This method of search has been extensively discussed by Doran and Michie who used it in their program called the 'Graph Traverser'. They have shown that given a suitable reluctance function it is an effective method of tackling a number of problems.

The sequence is illustrated in figure 4. The main point to notice is that the next node whose successors are examined is not necessarily one of those produced at the previous move, i.e. the frontier will be pushed forward for a while in one region, but if the reluctance function indicates that further advances here are not as profitable as had been anticipated some other part of the frontier may be extended.

It is easy to see that by choosing a suitable evaluation function parallel search and depth-first search can be produced as special cases of search controlled by a reluctance function. Specifically:

(i) *Parallel search.* The reluctance function is the distance (number of arcs) between the starting node and the given node. The rule that the frontier node with least value of the reluctance function is taken first causes the frontier to advance one layer at a time (*see* figure 5).

(ii) *Depth-first search.* Taking a tree with binary branches for simplicity the reluctance function assigns to each node a binary fraction whose digits are (left to right) 0 or 1 according to the choices made in obtaining that node from the original node (*see* figure 6).

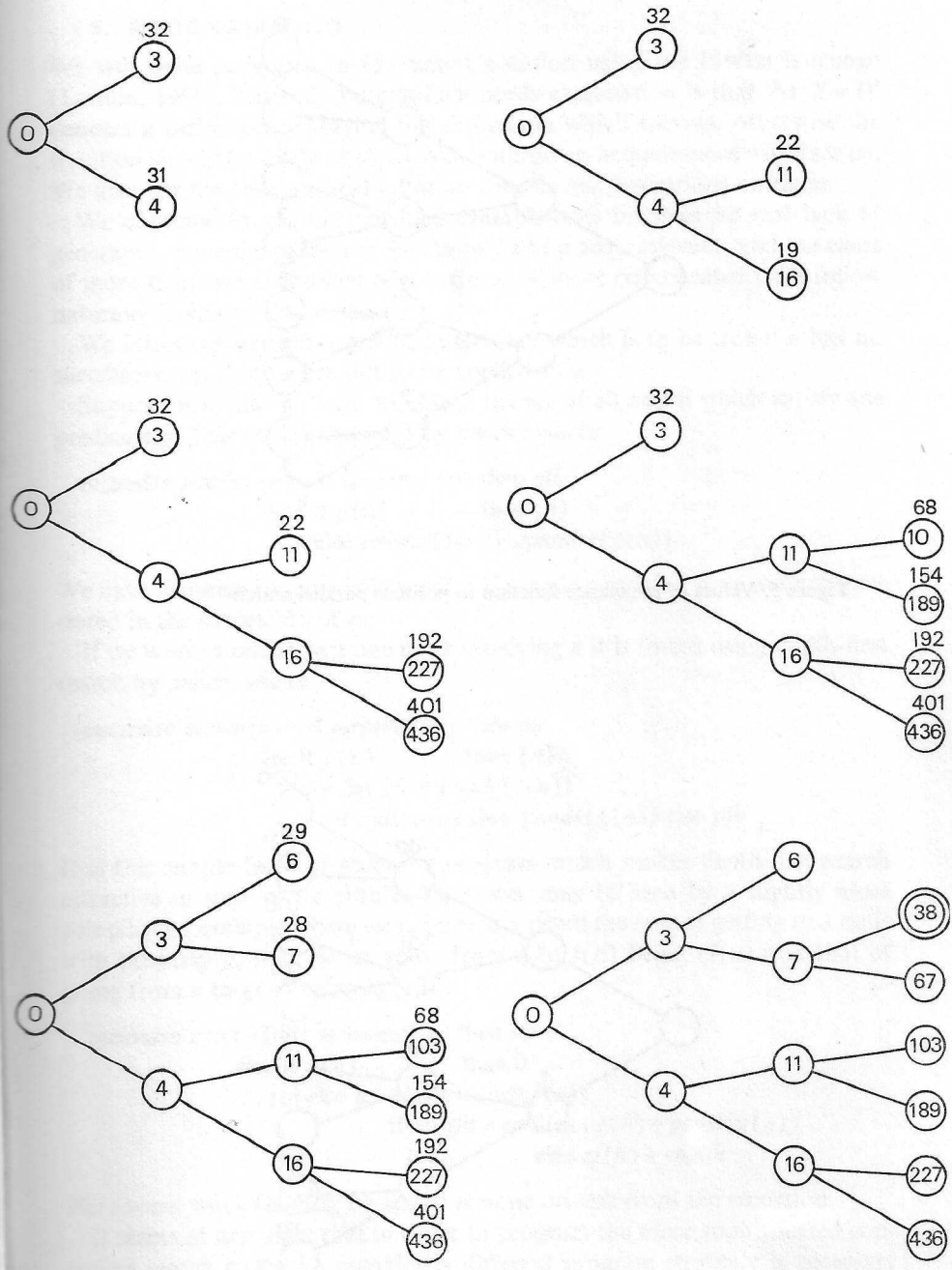

Figure 4. Controlled search. Reluctance function: $r(n) = |n - 35|$.
Values of reluctance are written above the frontier nodes

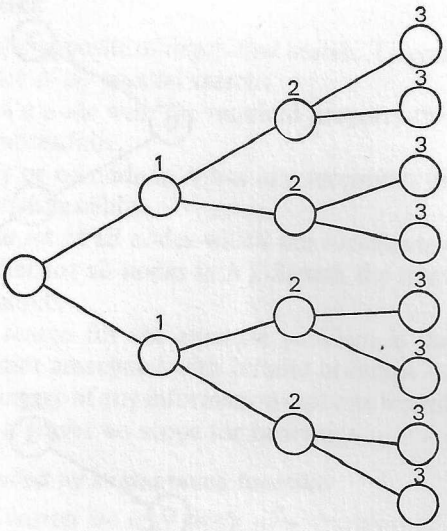

Figure 5. Values of reluctance function to produce parallel search

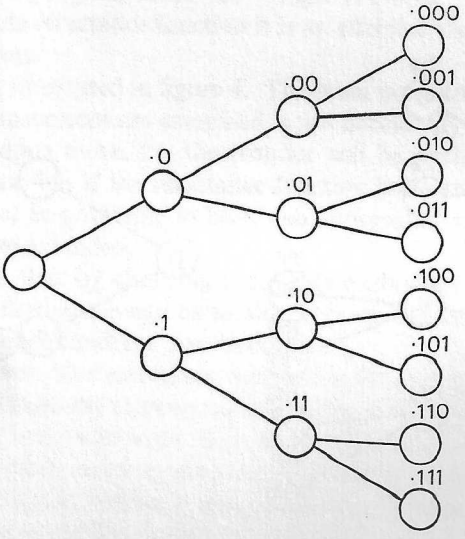

Figure 6. Values of reluctance function to produce depth-first search

3. PROGRAMMING

We will write programs in functional notation using the ISWIM language (Landin, 1966). The only point which needs explanation is that '**let** $X=D$' denotes a definition qualifying the expression which follows, otherwise the notation should be fairly obvious even without an acquaintance with ISWIM. We also use the conventional notation for sets and operations on them.

We continue to use the previous example–this involves no real lack of generality, since the particular functions f and g are irrelevant, and the cases of more than two successors of a node or of more complicated trees follow naturally from our discussion.

We introduce one extra predicate *terminal* which is to be true if n has no successors, i.e. f and g are not to be applied to it.

Suppose first that we wish to obtain the set of all nodes which satisfy the predicate p. This set is computed by *pnodes* where

recursive *pnodes* $(n)=$**if** *terminal* (n) **then** *nil*
else if $p(n)$ **then** $\{n\}$
else *pnodes*$(f(n)) \cup$*pnodes*$(g(n))$

We have assumed for convenience that if a node n satisfies p we are not interested in the successors of n.

If we wish to obtain just one node satisfying p it is found using depth-first search by *pnode*, where

recursive *pnode*$(n)=$**if** *terminal*(n) **then** *nil*
else if $p(n)$ **then** $\{n\}$
else let *pfn*$=$*pnode*$(f(n))$
if *null*(pfn) **then** *pnode*$(g(n))$ **else** *pfn*

It is this simple form of recursive program which makes depth-first search attractive in spite of its pitfalls. Its power may be seen by a slightly more complicated example where we require as a result the cost of getting to a node with property p, the cost of going from n to $f(n)$ being $cf(n)$ and that of going from n to $g(n)$ being $cg(n)$.

recursive *cost*$(n)=$**if** *terminal* (x) **then** ∞
else if $p(n)$ **then** 0
else let *costfn*$=$*cost*$(f(n))$
if *costfn*$=\infty$ **then** $cg(n)+$*pcost*$(g(n))$
else $cf(n)+$*costfn*

Here some work (adding up costs) is done on exit from the recursion.

It seems at first sight that in order to program the more sophisticated controlled search method a completely different program structure is necessary and the elegant recursive approach must be abandoned. Doran and Michie in their 'Graph Traverser' use an approach which in its essentials may be expressed thus:

Suppose that $\phi(n)=\{f(n), g(n)\}$, i.e. the set of successors of n, and that $r(n)$ is the reluctance function for a node, saying how undesirable it is.

```
pnode(n)=
    let N={n}
    loop: let nmin=least(r,N);
          if ∃ n∈φ(nmin) [p(n)] then goto out;
          N:=N−{nmin}∪φ(nmin); goto loop;
    out: result any(p,φ (nmin))
```

Here *least*(*r*,*nodes*) finds the member of N with least value of r and *any* (p, ϕ(*nmin*)) finds any member of ϕ(*nmin*) satisfying p.

This could of course have been written recursively, although it is essentially an iterative algorithm. Thus:

```
recursive pnode(N)=
        let nmin=least(r,N);
          if ∃ n∈φ(nmin)[p(n)] then any(p,φ(nmin))
             else pnode(N−{nmin}∪φ(nmin))
```

The depth of recursion now corresponds to the total number of nodes examined and not as previously to the distance along the branch from the start to the current node. The problem above of costing the path from the start to the solution node is no longer so easy. It would require that a back pointer be passed on explicitly with each node so that when the solution is found a retrace can be done and the cost can be worked out. This is an important limitation of this way of programming controlled search: dealing with the answer when found needs special programming.

4. ALTERNATIVE EXPRESSIONS

The point of this paper is to show how the controlled search can be programmed in a manner very similar to the recursive method for depth-first search. To do this we may introduce a new kind of expression called an 'alternative expression' which could be thought of as an extension of any programming language of the ALGOL family. This kind of expression involves a special evaluation rule which we will explain informally. In the next section we will show that the rules for alternative expressions can be made quite precise in terms of '*J*' operator, a form of generalized jump (Landin, 1966). Indeed if our programming language is equipped with this operator or an equivalent generalized jump facility, we do not need to introduce alternative expressions as a new feature at all since a function can be defined which has the same effect.

Consider the expression

$$h(x+1, y-2)$$

To evaluate it we must evaluate BOTH $x+1$ AND $y-2$.

But consider the expression

$$h(x)<3 \textbf{ and } k(x)>0$$

To evaluate this it is sufficient to evaluate EITHER $h(x)<3$ OR to evaluate $k(x)>0$ provided that the one evaluated has truth value false (if not, the other expression must also be evaluated).

Now suppose that *fpos*(n) is the smallest number greater than n which has a property p and *fneg*(n) is the largest number less than or equal to n which has p. Then if we just want an expression whose value is *some* number which has p we might write

fpos(0) **alternatively** *fneg*(0) (**alternatively** is a new basic symbol)

Here it would be sufficient to evaluate EITHER *fpos*(0) OR *fneg*(0).

To evaluate this expression we might get the computer to work on both in turn taking as result whichever it managed to evaluate first.

Evaluating *fpos*(0) might well involve evaluating another such **alternative** expression and we would think of the possible calculation as a tree of which it would be sufficient to evaluate completely just one branch. If we do not wish the computer to use its discretion as to which branch to evaluate we must provide some extra information about each branch, i.e. reverting to our controlled search idea we must provide the value of the evaluation function for that branch. We will call this an alternative expression.

$<$*alternative expression*$>$ ::= $<$*expression*$>$ **reluc** $<$*real expression*$>$
alternatively
$<$*expression*$>$ **reluc** $<$*real expression*$>$

fpos(0) **reluc** 5 **alternatively** *fneg*(0) **reluc** 10

We now program a search controlled by an evaluation function in a manner corresponding exactly to a depth-first search (we use r for the reluctance function).

recursive *pnode*(n) = **if** $p(n)$ **then** $\{n\}$
else (*pnode*$(f(n))$ **reluc** $r(f(n))$
alternatively *pnode*$(g(n))$ **reluc** $r(g(n))$)

The idea here is that in the recursive evaluation of *pnode* a lot of alternative expressions will be activated each with two component expressions. The evaluation mechanism is to consider all the component expressions which have been produced by activation of alternative expressions but have not yet been evaluated themselves, and evaluate the one which has least value of the reluctance. This means that the evaluation mechanism has somehow to keep a list of component expressions to be evaluated, with suitable information about their environments, and a link to the alternative expression to which

they belong. We assume at the moment that there is only one such list for a particular program.

One or two remarks are worth making. There is no reason why a program should not contain more than one alternative expression. In this case the tree of possible evaluations will not be homogeneous–it will have nodes, some of which correspond to activations of one alternative expression, some to activations of other alternative expressions.

Likewise there is no reason why an alternative expression should have just two components. We may allow more or even less than two. We may easily want the number of components to be determined dynamically.

We have not included in our last definition of *pnode* the case where *n* is a terminal node. One way of dealing with this would be to insert there an alternative expression with no components i.e. the other branches of the tree already in existence are to be followed but this node does not give rise to any new branches.

It is a bad principle, however, to extend a programming language to give some new features unless we are quite sure that the extension is of sufficient generality. Can we make use of some more generally desirable feature of programming languages and avoid the need for 'alternative expressions'?

5. USE OF THE *J* OPERATOR

In another paper presented at this symposium* Landin puts forward a jump operator '*J*' which enables his functional programming language ISWIM to handle departures from the normal sequence of evaluation, such as error exits. The case of a search which is to be terminated as soon as it is successful is closely analogous to an error exit, and we now show how *J* may be used to program 'alternative expressions'. This means that we withdraw the above proposal for alternative expressions (which was merely an explanatory device) and offer instead an equivalent library function called *oneof*.

We keep the component expressions to be evaluated together with associated values of their reluctance function on a list called jobs. They are ranked in ascending order of the values of their reluctance function.

How are we to represent the component expressions? If we simply write down the expressions they will get evaluated straight away, which is not our intention. What we need is a function which when applied to a list of jobs will produce the required value. It is this function which is stored on the jobs list. Before it is stored however, we must apply *J* to it. This means that if it ever gets applied it will return its result as the result of the '*oneof*' expression of which it is a component.

Thus the task of the function *oneof* is to add to a list of jobs supplied to it as a parameter the two component functions supplied to it with their associated values, but only after applying *J* to them. It must then take the first component function off the jobs list and apply it to the rest of the jobs list.

* See note at end of this paper.

The function *oneof* is as follows:

```
oneof(f,rf,g,rg,jobs)=
    let jobs=addjob(J(f),rf,jobs);
    let jobs=addjob(J(g),rg,jobs);
    let ((jh,rh),jobs)=next(jobs); N.B. next produces the head and tail
      jh(jobs)                     of a list.
   where recursive addjob(jf,rf,jobs)=if null (jobs) then ((jf,rf)) else
         let ((jh,rh),jobs)=next(jobs);
           if rf≤rh then (jf,rf)::(jh,rh)::jobs        N.B. :: is an infixed
                    else (jh,rh)::addjob(jf,rf,jobs)   operator for 'cons'.
```

Given *oneof* we can now write *pnode* again very easily

```
recursive pnode(n)=pnoden
    where pnoden(jobs)=if p(n) then {n}
                       else oneof(pnode(f(n)),r(f(n)),
                                  pnode(g(n)),r(g(n)),jobs)
```

To find a node satisfying p starting from a node $n0$ we evaluate

```
pnode(n0)(nil)
```

The action is not difficult to grasp once we realize that the functions denoted by *jf*, *jg*, *jh* above always jump back as soon as they are executed to the *oneof* expression which gave them birth, returning their value as the result of that expression.

We have written *oneof* to take two components but there is no difficulty in writing it to take a list of any number of components (we supply a list of function-reluctance pairs and use *maplist*).

The difficulty about terminal nodes can then be dealt with by invoking *oneof* and giving it an empty list of component jobs. This allows existing branches of the tree to be processed further without creating any new ones.

Thus the definition of *oneof* as a library function enables us to carry out controlled searches in an elegant and transparent recursive manner without extending the language with any new form of expression or special evaluation mode. Although our example has been a simple one, much more complex searches may be programmed using the same function *oneof*.

NOTE ON LANDIN'S *J* OPERATOR

P.J. Landin gave a talk at the Machine Intelligence Workshop about his *J* operator, a generalized jump facility. Unfortunately the written version was not available in time for publication in this volume. Since my paper makes use of this operator, Mr Landin has kindly agreed to my including a short explanation of it here. He hopes to publish a fuller account shortly and the reference may be obtained from him at Queen Mary College, University of

London. Thus, the work described below is due to Landin, but the explanation is mine and he is not responsible for any defects in it.

In a previous paper Landin (1966) introduced the notion of a 'program point'; this is analogous to a function (or ALGOL procedure) but has a non-standard mode of exit. Instead of returning control to the point from which it is called, it returns control to the point from which the function in which it is defined is called. An example in ALGOL format may help.

```
integer procedure f(x); integer x;
        integer program point f1(y); real y; f1:=entier(y×100);
        real procedure g(z); integer z;
        . . . . . .
        if trouble >0 then f1(1/z) end; z:=z+1;
        . . . . . .
        end of g;
        . . .
        v:=g(3/x);
        . . .
end;
. . .
u:=f(0); u:=u+1;
```

Here $f1$ is an integer program point (i.e. an integer procedure with non-standard exit) defined directly within the integer procedure f. Its job is to return an answer which would do as the answer to its parent f in case computing the answer to f ran into some trouble, e.g. during the internal procedure g. Thus, if g calls $f1$ during execution of f, the subsequent statement '$z:=z+1$' is *not* executed, but instead an immediate exit from its parent procedure f takes place, the result of f being $f1(1/z)$, i.e. $entier(1/z\times 100)$. This value is assigned to u and the next statement executed is '$u:=u+1$'. Thus the remainder of g and f have been short circuited, with a forced exit returning the result of $f1$ as the result of its parent f.

In ISWIM we have

```
let f(x)=
    let program point f1(y)=entier(y×100)
    and g(z)=
        . . .
        if trouble >0 then f1(1/z); z:=z+1;
    . . .
    v:=g(1/x);
. . .
u:=f(0); u:=u+1;
```

A possible improvement to this device, suggested by Landin in his talk, is to replace $f1$ by an ordinary integer procedure, say $f0$, and have a special

operation J which when applied to $f0$ converts it into a program point. The parent of $f1$ would then be determined by the procedure in whose body the J occurred (i.e. the innermost such procedure). Thus we could write for example

```
let f0(y) = entier(y × 100);
...
let f(x) =
    let f1 = J(f0)
    and f(z) =
        ...
        if trouble > 0 then f1(1/z);
etc.
```

The point is that J incorporates a 'fire escape' leading to the exit point of its parent procedure, and attaches this fire escape to the procedure to which it is applied, for use instead of that procedure's normal exit mechanism.

REFERENCES

Doran, J.E. (1967), An approach to automatic problem-solving. *Machine Intelligence 1*, pp. 105–23 (eds Collins, N.L. & Michie, D.). Edinburgh: Oliver and Boyd.

Doran, J.E. (1968), New developments of the Graph Traverser. *Machine Intelligence 2*, pp. 119–35 (eds Dale, E. & Michie, D.). Edinburgh: Oliver and Boyd.

Doran, J.E. & Michie, D. (1966), Experiments with the Graph Traverser program. *Proc. R. Soc.* (A), **294**, 235–59.

Golomb, S.W. & Baumert, L.D. (1965), Backtrack programming. *J. Ass. comput. Mach.*, **12**, 516–24.

Landin, P.J. (1966), The next 700 programming languages. *Communs Ass. comput. Mach.*, **9**, 157–66.

Michie, D. (1967), Strategy building with the Graph Traverser. *Machine Intelligence 1*, pp. 105–23 (eds Collins N.L., & Michie, D.) Edinburgh: Oliver and Boyd.

Slagle, J.R. (1965), Experiments with a deductive question-answering program. *Communs Ass. comput. Mach.*, **8**, 792–8.