Programming the Premature Loop Exit: From Functional to Navigational

C K Yuen
DISCS, NUS
Kent Ridge, Singapore 0511

We wish to study a simple programming problem involving premature loop exits, in Lisp and Pascal, and use it to make a few points about appropriate methods of program structuring. The problem is:

> Inspect an array A[0..N] to find which is true -
>
> (a) It contains no zeros
> (b) It contains only one zero
> (c) It has two zeros separated by an even number of non-zeros
> (d) It has two zeros separated by an odd number of non-zeros.
>
> In (c) and (d) the first occurrence is all we want. For each case, perform certain specific processing.

Below is a short solution written in BaLinda Lisp, a parallel Lisp dialect but used here sequentially only:

```
(DEFUN Find (Zero I)
      (COND ((> I N) (COND ((NULL Zero) ..(a)..)
                           (T ..(b)..)))
            ((= (ARRAY A I) 0)
             (COND ((NULL Zero) (Find I (+ I 1)))
                   ((ODD (- I Zero)) ..(c)..)
                   (T ..(d)..)))
            (T (Find Zero (+ I 1)))))
...
(Find NIL 0)
```

The function Find recurses through successive elements of A. If it finds one 0, the argument Zero is set to retain the index; until then, Zero is NIL. The recursions terminate if I reaches N or if a second 0 is found, in which case the gap between the two 0's is checked. The four different actions for the four conditions are embedded in the search itself, making the program very concise and terse.

Although the program is quite simple and straightforward, in our experience most students find it hard to understand. The situation improves somewhat if we make a retreat from terseness:

```
(DEFUN Find (Zero I)
      (COND ((> I N) (COND ((NULL Zero) 'NoZero)
                           (T 'OneZero)))
            ((= (ARRAY A I) 0)
```

```
              (COND ((NULL Zero) (Find I (+ I 1)))
                    ((ODD (- I Zero)) 'EvenGap)
                    (T 'OddGap)))
             (T (Find Zero (+ I 1)))))
...
(LET ((X (Find NIL 0)))
     (COND ((= X 'NoZero) ..(a)..)
           ((= X' OneZero) ..(b)..)
           ((= X 'EvenGap) ..(c)..)
           ((= X 'OddGap) ..(d)..))
```

It seems that, by putting into the program meaningful text strings, we make it easier for the program reader to figure out what each part does. Note also that here actions (a) to (d) execute in the main program environment, whereas in the previous version they are in the environment of Find. Whether they would have the same effect in both cases can be decided only after issues like variable scoping have been considered, but we do not go into this here.

Unfortunately a problem has crept in with the "user-friendliness": the processing sections (b) and (c) and (d) no longer have the indices of the zeros. If this information is needed in the processing, then we have to make another retreat:

```
(DEFUN Find (Zero I)
      (COND ((> I N) (LIST Zero NIL))
            ((= (ARRAY A I) 0)
             (COND ((NULL Zero) (Find I (+ I 1)))
                   (T (LIST Zero I))))
            (T (Find Zero (+ I 1)))))
...
(LET ((X (Find NIL 0))
      (First (CAR X))
      (Second (CADR X)))
     (COND ((NULL First) ..(a)..)
           ((NULL Second) ..(b)..)
           ((ODD (- Second First) ..(c)..)
           (T ..(d)..)))
```

We have now packaged the two indices into a list, and must later test the two values to choose the four actions (a) to (d). The program is (we guess) slightly harder to understand for the average student; among other reasons, he is puzzled at seeing that Find returns, not four different lists, but only two. However, Most proficient Lisp programmers would see little difference in quality between the three versions, and would not consider any one to be noticeably better or worse. Each version is "functional" since no assignments are used, as well as "structured". We would ourselves favour the shortest version which takes us less time to read and figure out.

Now consider the Pascal versions, starting by recoding the third Lisp program:

```
I:= 0;
First:= -1;
Second:= -1;
Exit:= FALSE;
REPEAT IF (A[I]=0)
       THEN IF (First=-1)
            THEN First:= I
            ELSE BEGIN
                 Second:= I;
                 Exit:= TRUE
                 END;
       IF (I=N) THEN Exit:= TRUE
                ELSE I:= I+1
UNTIL Exit;
IF (First=-1)
THEN ..(a)..
ELSE IF (Second=-1)
     THEN ..(b)..
     ELSE IF (ODD (Second-First))
          THEN ..(c)..
          ELSE ..(d)..
```

Instead of a recursive function, a loop is used, and instead of being returned in a two element list, the indices of the two zeros are assigned to two variables. Since Pascal integers cannot take Boolean values, -1 is used to denote "undefined". These changes are obviously appropriate for Pascal. Another slight difference is that we do not re-enter the loop with I=N+1 and then immediately exit, but set exit if I=N, with the result that if A[N] happens to be the second 0, then Exit is set to TRUE twice. This avoids having to test the value of I at the start of the loop, which is a standard practice in Lisp but rather alien to Pascal programmers.

Compared with the Lisp programs, the Pascal code is rather verbose, but perhaps we could make it more terse by embedding? If anyone thinks so, then it must produce some disappointment when the first Lisp version is recoded in Pascal:

```
I:= 0;
First:= -1;
Exit:= FALSE;
REPEAT IF (A[I]=0)
       THEN IF (First=-1)
            THEN First:= I
            ELSE BEGIN
                 IF (ODD (I-First))
                 THEN ..(c)..
                 ELSE ..(d)..
                 Exit:= TRUE
                 END;
       IF (I=N)
       THEN BEGIN
            IF (First=-1)
            THEN ..(a)..
            ELSE IF (NOT Exit)    <---
                 THEN..(b)..;
            Exit:= TRUE
            END
            ELSE I:= I+1
UNTIL Exit;
```

The actions (a) to (d) are embedded inside the loop, but the result is not at all concise. The variable Second is no longer required, but an extra Boolean test (indicate by <---) has crept in: in the earlier version, if the second 0 happens to be the last array element, then Exit is set to TRUE twice, which does no harm, but here it would also cause the execution of (b), if not prevented by the additional test. Also, readability is less good: it takes some effort to realize that only one of the actions (a) to (d) is performed, and once only because of the setting of Exit to TRUE. Clearly the embedding version is inferior to the non-embedding version.

So why is it that embedding works with recursion but not with iteration? The reason seems to be that, because in recursive programs repetition is explicitly indicated by a recursive call, the reader immediately sees that any part not followed by such a call is executed once only. We can see clearly that this applies to actions (a) to (d) in the Lisp program. In a loop, repetition is assumed unless something in the loop sets the exit condition, and some mental effort is needed to detect this and to see that because of the exit, the four alternatives are mutually exclusive. Further, for such a program, embedding does not shorten the code, because after performing the loop terminating actions, one still has to explicitly set the loop exit condition.

Is it then impossible to achieve the same kind of terseness in Pascal? Actually for this problem we can, but only at the cost of violating one of the great rules of structured programming, by using GOTOs:

```
FOR First:= 0 TO N DO
    IF (A[First]=0)
    THEN BEGIN
         FOR Second:= First+1 TO N DO
             IF (A[Second]=0)
             THEN IF (ODD (Second-First))
                  THEN GOTO EvenGap
                  ELSE GOTO OddGap;
         GOTO OneZero
         END
NoZero: ... GOTO ..
OneZero: ... GOTO ..
EvenGap: ... GOTO ..
OddGap: ...
..
```

This is concise because each FOR loop automatically increments its index once per iteration, and terminates when this reaches N, making explicit exit tests and index increments unnecessary, and the four conditions (a) to (b) are detected in different parts of the loop, in which case a jump is immediately made to the appropriate action outside the loop. The jump causes the loops to terminate, and also

causes the values of First and Second to be retained for use outside the loop, making it unnecessary to return results or explicitly set index variables. This version took less time to write and verify than any of the other versions, Lisp or Pascal.

It is curious that we descended from "structured" to "unstructured" programming in order to reproduce some of the advantages of functional programming: like Mohamed and the mountain, if we do not find it good to put code where it is needed, we would jump to where the code is; hence the commonality between embedding and GOTOs. Either method links the detection of a condition requiring certain processing immediately to the processing itself; or to put it differently, directly navigates from need to action. Where the relation between needs and actions is simple, the two methods are concise and effective, and do not produce badly structured code.

Where the relation is more complex, it is necessary to navigate through the program some other way. The first Pascal program and the last Lisp program do it by retaining information in variables and using their values to select code for execution, a method we once compared to navigation by cargo[1][2]. This has several advantages. First, by giving meaningful names and values to the variables, the need-action relation is made clearer. Second, the information can be packaged in an appropriate data structure to facilitate handling. Third, partial needs detected at different parts of the program, and partial actions performed at different times, are easier to organize. However, for simple situations the method can look unnecessarily complex, and the temptation to use a short cut is hard to resist. Further, one can have badly structured control information, just as one can have badly structured GOTO links. Good programming must adopt a judicious combination of the three techniques, taking into account the limitations of each.

References

[1] Yuen, C K, The programmer as navigator, ACM SIGPLAN Notices, 18, no. 9 (1983): 70-78.

[2] Yuen, C K, Further comments on the premature loop exit problem, ACM SIGPLAN Notices, 19, no. 1 (1984): 93-4.