# A Functional FOR Loop

C K Yuen
DISCS, NUS
Kent Ridge, Singapore 0511
email: YuenCK@ISCS.NUS.SG

The idea of this article arose during the design of BaLinda K, a version of BaLinda Lisp with a superficially C-like syntax. ("K" is hard "C".) BaLinda Lisp has no iterative construct, and recursive calls must be used to implement looping. The question was then posed as to whether BaLinda K, which is meant to be more user friendly to imperative minded programmers, should have looping facilities. It was decided to have no WHILE/REPEAT loops, but FOR loops turn out to be a less negative case.

Consider the simple example of summing an array:

```
(DEFUN Loop (Sum I)
       (COND ((> I N) Sum)
             (T (Loop (+ Sum X[I]) (+ I 1)))))

Loop (0. 1)
```

Several matters make the program unnatural to imperative programmers. First, the exit condition is tested at the start of the loop rather than the end, because otherwise X[I] might go beyond the array bound. Second, the arithmetic is not performed in the loop body but in the recursive call, by passing a new argument Sum to the next iteration. Third, the loop index increment (+ I 1) is also embedded in the recursive call. Finally, the loop initialization, I=1 and Sum=0, is embedded in the initial Loop call, which comes after the loop definition. While these might seem to be very minor problems to Lisp programmers, it is by no means simple to convert others to this way of thinking. Writing complex nested loops this way is much harder still.

Now consider the alternative of using a Pascal REPEAT loop:

```
Sum:= 0;
I:= 1;
REPEAT Sum:= Sum+X[I];
       I:= I+1
UNTIL I>N;
```

Loop initialization has been moved to the front, and the summing action is separately shown, but the programmer still has to explicitly put in the exit test and the index increment. Programming this way does not seem to be much simpler than recursion. In contrast the FOR loop solution

```
Sum = 0.
FOR I = 1 TO N DO
    Sum = Sum+X[I];
```

provides a more significant improvement, by simpiflying the looping control.

Having a loop construct in a recursive language is not just a trivial matter of adding some preprocessing, because there are certain special requirements for loops in functional

programming. Consider the two Pascal loops: Neither the loop as a whole nor the individual iterations return any result, and information carrying from one iteration to the next is achieved through side effect, by changing Sum. Though these practices are natural in imperative languages, they are contrary to functional principles.

To conform to these, the BaLinda K FOR loop uses the CONTINUE statement to call the next iteration, with the possibility of passing over arguments so that the results of one iteration may pass to the next. The FOR part starts the first iteration, establishing initial values of the arguments as well as specifying iterative changes and termination conditions.

To take an example, summing an array is done as:

```
{ FOR Sum = 0.,
      I = 1 TO N RETURN Sum
  DO CONTINUE (Sum+X[I], -)
 }
```

Sum and I are both initialized in FOR..DO, but a new value of Sum is passed in CONTINUE, whereas the change for I (increment by 1) is already specified in the FOR part and so is not required in CONTINUE. Its place is taken by the "don't care" symbol - (also used in tuple IN/RD statements to avoid retrieving unwanted fields). In both the Lisp and K programs, Sum and I are internal to the loop and their values are not accessible from the outside. Hence, in BaLinda K an explicit RETURN construct is needed to obtain the results of loops.

The next example is a slightly more complex loop which multiplies elements of list X into those of array Y, summing the products to get the inner product of X and Y:

```
{ FOR Sum = 0.,
      Z = ZIP (X) RETURN Sum,
      I = 1 TO N RETURN Sum
  DO CONTINUE (Sum+Z*Y[I], -, -)
 }
```

Here Z "zips" through the elements of X (a method also used in some functional languages), while I goes from 1 to N. Reaching either the end of X or Y causes the return of Sum. We see that the ZIP feature allows the FOR loop to be used like a WHILE.

A FOR loop may also terminate prematurely, e.g., in the above if the array or list has a zero then an error is found requiring immediate loop exit. This is handled by using RETURN inside the loop:

```
{ FOR Sum = 0.,
      Z = ZIP (X) RETURN Sum,
      I = 1 TO N RETURN Sum
  DO { IF Z==0.     => RETURN NIL;
       Y[I]==0. => RETURN NIL;
       T         => CONTINUE (Sum+Z*Y[I], -, -)
 }    }
```

which is equivalent to

```
(DEFUN Loop (Z I Sum)
      (COND ((NULL Z) Sum)
            ((> I N) Sum)
            ((= (CAR Z) 0.) NIL)
```

```
            ((= Y[I] 0.) NIL)
            (T (Loop (CDR Z) (+ I 1)
                     (+ Sum (* Y[I] (CAR Z)))))))

(Loop X 1 0.)
```

This might also be compared with the Common Lisp version:

```
(CATCH Zero
       (LOOP (INITIALLY (I 1)
                        (Z X)
                        (Sum 0.))
             (WHILE (AND (<= I N) (NOT (NULL Z))))
             (DO (IF (= (CAR Z) 0.)
                     (THROW Zero NIL)
                     (IF (= Y[I] 0.)
                         (THROW Zero NIL)))
                 (SETQ Sum (+ Sum (* (CAR Z) Y[I])))
                 (SETQ I (+ I 1))
                 (SETQ Z (CDR Z)))
             (FINALLY Sum)))
```

which is rather like Pascal except for the provision of premature exit by CATCH-THROW. The Scheme DO loop is a little simpler in structure than the Common Lisp loop but the difference is minor.

To give a larger example, below is a breadth-first 8-queens program in BaLinda K:

```
{ FOR Init = '((1)(2)(3)(4)(5)(6)(7)(8)),
      Length = 2 TO 8 RETURN Init
   DO CONTINUE
      ({ FOR Soln = NIL,
             Row = ZIP (Init) RETURN Soln
          DO CONTINUE
             ({ FOR Result = Soln,
                    New = 1 TO 8 RETURN Result
                DO { IF { FOR Old = ZIP (Row) RETURN T,
                              Column = Length-1 DOWNTO 1
                          DO { IF Old==New OR Length-Column==ABS(New-Old)
                                    => RETURN NIL
                             }     }
                           => CONTINUE (CONS (CONS (New Row), Result), -);
                       T => CONTINUE (Result, -)
                  }     },
               -)
        },
       -)
};
```

A full explanation of the algorithm may be found in [1]. Note that the innermost FOR loop does not require a CONTINUE statement as no arguments need to be passed from iteration to iteration.

The program may be compared with the Lisp version:

```
(DEFUN NoCheck (I J Row)
      (COND ((= I 0) T)
            ((= J L) NIL)
            ((= (- K I) (ABS (- L J))) NIL)
            (T (NoCheck (- I 1) (CAR Row) (CDR Row)))))
```

```
(DEFUN Extend (Soln Row L)
      (COND ((> L 8)
             (COND ((NULL Soln) NIL)
                   (T (Extend (CDR Soln) (CAR Soln) 1))))
            ((NoCheck (- K 1) (CAR Row) (CDR Row))
             (CONS (CONS L Row) (Extend Soln Row (+ 1 L)))))
            (T (Extend Soln Row (+ 1 L)))))

(DEFUN Init (Soln K)
      (COND ((> K 8) Soln)
            (T (Init (Extend (CDR Soln) (CAR Soln) K 1) (+ 1 K)))))

(Init '((1)(2)(3)(4)(5)(6)(7)(8)) 2)
```

and the Miranda version[2]:

```
NoCheck Row L = ALL [(J<>L)AND((K-I)<>ABS(L-J)) | (I,J)<-ZIP([1..#Row], Row)]
                WHERE K = #Row+1

Queens 1      = [[1][2][3][4][5][6][7][8]]

Queens (K+1)  = [Row++[L] | Row <- Queens K; L <- [1..8]; NoCheck Row L]
```

Note that the Lisp Extend function implements two loops through its two recursive calls, such that, though only three functions are defined, four loops are present. Note also that the Miranda version tests all elements of Row, whereas the Lisp and K versions exit from the inner loop whenever a check is found and so would only test a part of Row.

We ourselves have found the queens program with FOR loops to be no easier to write than the Lisp version. For us the main advantage seems to lie in the more "integrated" structure, with loop nesting rather than separate functions.

The examples demonstrate that our FOR loop fully meets the requirements of functional programming, but follows closely the structure and appearance of the FOR loop of imperative languages. Though it is a little more elaborate because of the need to explicitly specify returned results, the added features fit in naturally, and should not cause significant learning difficulties.

## References


[1] C K Yuen and M D Feng, Breadth-first search in the eight queens problem, ACM SIGPLAN Notices, 29, No. 9 (1994): 51-55.

[2] R Bird and P Wadler, Introduction to Functional Programming, Prentice-Hall, 1988, p. 161.