Logic and Recursion: The Prolog Twist

Jesse M. Heines
Jonathan Briggs
Richard Ennals

Britain's young Prince William has been born into a famous family. As he grows up he will become aware of the distinguished ancestry which links him with several centuries of British monarchs through both his father and his mother.

According to current reports, the popular Prince still appears to lack the physical and intellectual power to carry out an exhaustive search for his forebears himself. However, he may already know enough to initiate a computer-aided search: he knows his immediate ancestors, his parents, as well as the next level, the parents of his parents.

Given the names of William's father and mother, a database of names connected by parent-child relationships, and an effective search strategy, one should be able to write a relatively simple program to list his geneology.

The key to the programming task is that the problem is *recursive*: for each level of ancestors, the next level can be determined by searching for the ancestors of those ancestors, and so on until no additional ancestors can be found. A recursive problem is one that can be broken down into two or more subprograms, at least one of which is the same as the original problem with a different set of conditions or arguments.

This article approaches the issue of recursion by describing a traditional programming problem and comparing the ways it is implemented in four languages: Basic, Pascal, Lisp, and micro-Prolog. It is a programming exercise and does not purport to be an exhaustive discussion of recursion or the recursive properties of the four languages.

No claims are made that recursion is a "good" programming technique, and we do not discuss its relative strengths and weaknesses as compared to other techniques such as iteration. Our purpose is simply to present an interesting illustration of recursion across four languages to provide insight into the subtle properties of each language.

The Factorial Function

We will use the factorial function to illustrate recursion throughout this article; as it is a classic example of recursive programming. The factorial of any positive integer n is the product of all positive integers from 1 to n and is represented by the symbol n! By definition, the factorial of 0 is 1. Therefore:

$0! = 1$
$1! = 1$
$2! = 2 \times 1 = 2$
$3! = 3 \times 2 \times 1 = 6$
$4! = 4 \times 3 \times 2 \times 1 = 24$

Before proceeding to coding in a particular language, it will help if we can develop a formal specification of the factorial function. This can be done using a logical notation, starting with the definition given above:

[1] 0 has factorial 1

The factorial of a positive integer is the product of that integer and the factorial of the integer to which it is the successor:

[2] u has factorial v if 0 < u and
u is the successor of w and
w has factorial x and
x times u = v

The successor relationship is specified as:

[3] u is the successor of w if w + 1 = u

In traditional computing terms, the simplest way to solve the factorial problem is via iteration. This can be illustrated with the Basic program in Listing 1, using a FOR/NEXT loop to accomplish the iteration.

Listing 1.

```
10 INPUT N                          Enter the number whose factorial is to be computed.
20 LET ANSWER = 1                   Initialize the ANSWER to 1.
30 IF N=0 THEN 70                   Handle the special 0 case.
40 FOR K=N TO 1 STEP -1             Begin looping from N to 1.
50   LET ANSWER = ANSWER * K        Multiply the ANSWER by the loop index.
60 NEXT K                           Iterate over the next index.
70 PRINT ANSWER                     Print the answer.
99 END                              Stop.
```

A flowchart for the iterative solution is shown in Figure 1. This program certainly does the job, but it is not a very clear representation of the factorial function in terms of the formal function specification given above. This lack of clarity is *not* because the program is written in Basic, as will be seen in the next section. It is because the solution uses iteration.

Recursion may or may not be more computer efficient, and it may or may not be easier to program. These issues are beyond the scope of this paper. However, as shown by the flowchart in Figure 2, recursion allows one to code the factorial function in a manner that is a much clearer representation of the formal function specification. The sections that follow illustrate how the recursive solution is programmed in four different languages.

Jesse M. Heines, Digital Equipment Corporation, Educational Services, Burlington, MA 01803.

Jonathan Briggs and Richard Ennals, Department of Computing, Imperial College of Science & Technology, University of London, London SW7 2BZ.

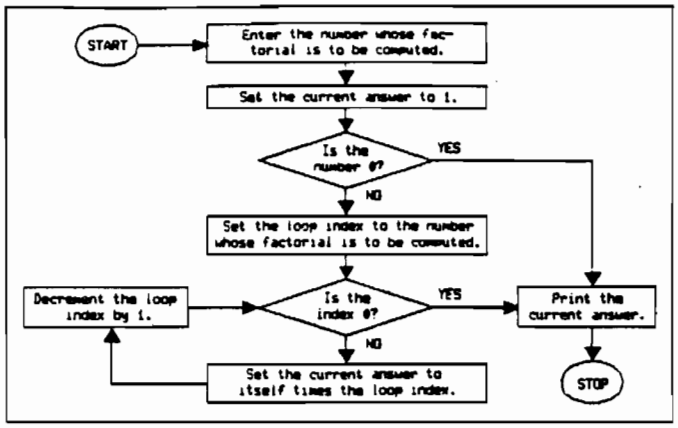

Figure 1. Flowchart of an interactive factorial function.

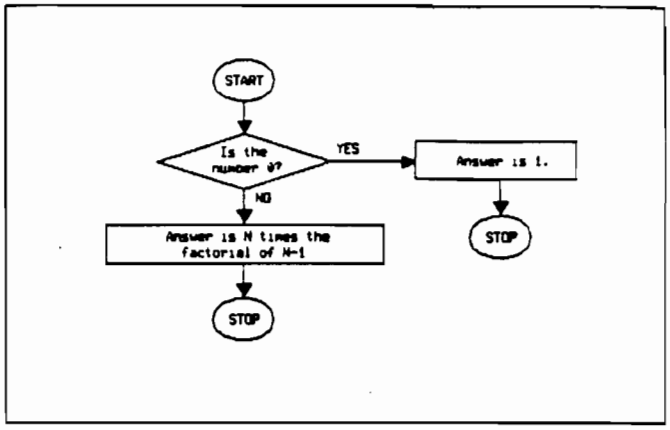

Figure 2. Flowchart of a recursive factorial function.

A Basic Implementation

Not all languages allow recursion, but this is really more a property of the language *implementation* than the language *design*. As stated earlier, a detailed technical discussion of the language implementation requirements for recursion is beyond the scope of this paper. But even Basic, that most maligned of computer languages, allows recursion in some implementations.

Just to demonstrate that it can be done, Listing 2 shows the code for a recursive implementation of the factorial function written in Digital Equipment Corporation's Basic+2.

Listing 2.

```
10 DEF FNF(X)                          Begin the function
                                         definition.
20   IF X=0  THEN  LET FNF=1           Handle the 0 case.
30   IF X<>0 THEN  LET FNF=X*FNF(X-1)  Handle all other
                                         cases.
40 FNEND                               End the function
                                         definition.
```

When the symbol FNF is used alone on the left side of the equal sign in lines 20 and 30, it represents the result that the function returns. When FNF (x-1) is used on the right side of the equal sign in line 30, it represents a recursive call to function FNF with a new argument (the original argument minus 1). The function is initially called with a statement such as:

```
100 LET F = FNF (4)
```

This function can be made more readable and efficient by using other advanced features of Basic+2, but we needn't proselytize further here.

A Pascal Implementation

While recursion may be somewhat foreign to Basic programmers, it should be familiar to Pascal programmers. Following is the code for a recursive Pascal definition of this function:

```
function factorial (x:integer) :    integer;
                                    (declare the start of an
                                    integer function with one
                                    integer argument)
begin                               (begin definition block)
    if x = 0 then factorial := 1    (handle the 0 case)
         else factorial := x*       factorial (x-1);
                                    (recurse using the orig-
                                    inal argument minus 1)
end;                                (end definition block)
```

As in the Basic implementation, when the symbol "factorial" is used alone on the left side of the :=sign it represents the result that the function returns. When it is used on the right side of the := sign within its own function definition, it represents a recursive function call. A Pascal statement for calling this function would have the form:

```
f := factorial (4);
```

A Lisp Implementation

With the recursion concept firmly in mind, we are now ready to go a bit further by looking at two interesting languages that are often used for logic and artificial intelligence applications. The first is Lisp:

```
(defun factorial (x)    ; begin the function
                        ; definition
  (cond                 ; begin a conditional block
   ((equal x 0) 1)      ; if the "equal" relation is
                        ; true when x is compared to
                        ; 0, return 1
   (t (times x (factorial     (difference
                               x 1))))))
                        ; in all other cases, return
                        ; the product of x and the
                        ; factorial of the difference
                        ; between x and 1
```

This code may be difficult to follow for readers who are not used to "prefix" notation, but this should be only a minor stumbling block. Prefix notation puts the operation first, followed by the two arguments. Thus, (equal x 0) means "if x equals 0," and (difference x 1) means "x minus 1." The comment following the last line of the code translates the recursive call into English. A Lisp statement to call this function equivalent to the two previous illustrations would look as follows:

```
(setq f (factorial 4))
```

which sets variable f to the value returned by the function call (factorial 4).

One of the reasons for showing the Lisp implementation is that it is relatively easy to trace Lisp functions to demonstrate the recursive calling sequence. A trace of the above call to compute the factorial of 4 is shown below. The first number in each line is the recursion level. The number is parentheses at the end of each ENTER FACTORIAL message is the *argument* with which the function is called. The number at the end of each EXIT FACTORIAL message is the *result* returned by that recursion level.

```
(1 ENTER FACTORIAL (4))
  (2 ENTER FACTORIAL (3))
    (3 ENTER FACTORIAL (2))
      (4 ENTER FACTORIAL (1))
        (5 ENTER FACTORIAL (0))
        (5 EXIT FACTORIAL 1)
      (4 EXIT FACTORIAL 1)
    (3 EXIT FACTORIAL 2)
  (2 EXIT FACTORIAL 6)
(1 EXIT FACTORIAL 24)
```

Logic and Recursion, continued...

A Micro-Prolog Implementation

Prolog is different from the three languages already discussed. Although there have been a variety of implementations since 1972, Prolog has received considerable attention in recent months, partly due to its adoption by the Japanese as the starting point for their Fifth Generation machines, which are to be based on logic programming.

The difference between Prolog and the other three languages is that Prolog is *declarative*, while the others are *procedural*. That is, rather than telling the computer *how* to compute the factorial of a number, in Prolog one tries to tell the computer what the factorial of a number *is*.

Micro-Prolog is an implementation of Prolog for microcomputers. Since 1980 it has been available for microcomputers with the Z80 microprocessor and the CP/M operating system. It is currently being implemented for a wide range of other microcomputers such as the Sinclair Spectrum, BBC micro, Apple, and Commodore 64.

Returning to our initial formal specification of the factorial function, we can see that little work is required to code the function in micro-Prolog. The sentence [1] in the specification can be used pretty much as it stands, adding only a hyphen in the relationship name:

[4] 0 has-factorial 1

This is the simplest type of micro-Prolog statement because no *conditions* are involved. More complex micro-Prolog statements can be thought of as *rules* that define conditions under which certain assertions are true. Sentence [2] in the formal specification can be rewritten into such a rule using the micro-Prolog convention where variables are X, Y, Z, x, y, z, X1, Y1, Z1, x1, y1, z1, etc., and the built-in arithmetic of the TIMES program and the LESS relation. (Note that the uppercase and lowercase letters represent different variables in this version of micro-Prolog.)

[5] X has-factorial Y if
    0 LESS X and
    X is-the-successor-of Z and
    Z has-factorial x and
    TIMES (X × Y)

This definition of the "has-factorial" relation is recursive because the relation name "has-factorial" appears as one of the conditional clauses within its own definition. Since X is defined by the previous clause to be the successor of Z, Z must be X-1. Thus, the recursive call in the clause "Z has-factorial x" is actually computing the factorial of X-1.

The specification of "is-the-successor-of" given in Sentence [3] is coded in micro-Prolog using the built-in SUM program:

[6] X is-the-successor-of Y if SUM (Y 1 X)

The micro-Prolog query for finding the factorial of 4 is:

```
Which (x 4 has-factorial x)
```

This is read, "Which values of x are there such that the relation 4 has-factorial x is true?"

Such a query follows the same pattern as queries to databases in micro-Prolog. Indeed, in Prolog, there is no distinction between program and database. A program consists of statements about relationships between individuals, which may be in the form of facts or rules.

The Promise of Logic Programming

Given only this exposure to recursion in Lisp and micro-Prolog, one might prefer Pascal or even Basic. The syntax of Lisp and micro-Prolog may appear unfamiliar, and some might even find it initially difficult to conceptualize something like the factorial function in terms of rules. The syntactic objection might be waved off by saying "one gets used to it," but the conceptual barrier warrants more explanation.

First, one must realize that neither Lisp nor Prolog was designed for mathematical calculation. The reader is referred to the texts listed at the end of this paper for a full discussion of their design considerations. Second, if all one wants to do is answer the question, "What is the factorial of n?," the authors agree that one might as well simply use a calculator with the appropriate function key.

But now the Prolog twist. We have seen that the factorial of 4 can be computed in Basic by using the statement:

```
LET X = FNF (4)
```

and in Pascal by:

```
x := factorial (4);
```

and in Lisp by:

```
(setq x (factorial 4))
```

and finally in micro-Prolog by:

```
Which (x 4 has-factorial x)
```

In the three procedural languages, one cannot use the functions defined in this paper to find out what number 24 is the factorial of. That is, one cannot write statements such as:

```
24 = FNF (x)
24 := factorial (x);
(setq 24 (factorial x)
```

Since the Prolog definition is based on logic, however, one can *theoretically* use it to solve the problem by posing the query:

```
Which (x x has-factorial 24)
```

This is the promise of logic programming: rule-based systems comprehensively defined can be run backwards as well as forwards. Unfortunately, implementation constraints on present machines restrict numerical applications at the present time.

The major problem is that the built-in micro-Prolog relations LESS, SUM, and TIMES are not declarative. They are machine-coded functions that perform their operations using standard procedural techniques. If they were declarative, one should, indeed, be able to get an answer to the backwards question posed above.

Interestingly enough, it is possible to write declarative versions of the LESS, SUM, and TIMES relations. The code for accomplishing this is provided below.

We begin by declaring successor relationships:

```
1 is-the-successor-of 0
2 is-the-successor-of 1
3 is-the-successor-of 2
4 is-the-successor-of 3
5 is-the-successor-of 4
6 is-the-successor-of 5
```

Using these relationships, we can then declare the rules by which one determines whether one number is less than another:

```
X is-less-than Y if
    Y is-the-successor-of X
X is-less-than Y if
    Z is-the-successor-of X and
    Z is-less-than Y
```

The first rule handles cases where Y is one greater than X, while the second rule uses recursion to handle all other cases. Given these relationships, we can now define the "sum" relation declaratively:

```
sum (0 0 0)
sum (0 X X) if
    0 is-less-than X
sum (X 0 X) if
    0 is-less-than X
sum (X Y Z) if
    0 is-less-than X and
    0 is-less-than Y and
    X is-the-successor-of x and
    sum (x Y y) and
    Z is-the-successor of y
```

The first rule says that the sum of 0 and 0 is 0. The second rule says that the sum of 0 and X is X if 0 is less than X. The third rule says that the sum of X and 0 is X if 0 is less than X. The fourth rule says that the sum of X and Y is Z if 0 is less

than both X and Y, X is the successor of some number x, and the "sum" relation is true such that x plus Y is y when Z is the successor of y. In essence, the recursive call in the last rule breaks down the problem of finding the sum of 2 and 3 as follows:

```
2 + 3 = (1 + 1) + 3
      = 1 + (1 + 0) + 3
      = (1 + 0) + (1 + 0) + 3
      = (1 + 0) + (1 + 0) + (2 + 1)
      = (1 + 0) + (1 + 0) + 2 + (1 + 0)
      = (1 + 0) + (1 + 0) + (1 + 1)
        + (1 + 0)
      = (1 + 0) + (1 + 0) + 1 + (1 + 0)
        + (1 + 0)
      = (1 + 0) + (1 + 0) + (1 + 0)
        + (1 + 0) + (1 + 0)
```

and returns the answer 5 by finding the successor of the successor of the successor of 0!

The "times" relation can be defined declaratively in terms of the above "sum" relation:

```
times (0 0 0)
times (0 X 0) if
     0 is-less-than X
times (X 0 0) if
     0 is-less-than X
times (X Y Z) if
     0 is-less-than X and
     0 is-less-than Y and
     X is-the-successor-of x and
     times (x Y y) and
     sum (Y y Z)
```

Interpretation of this definition is similar to "sum." We now have fully declarative definitions of the "less," "sum," and "times" relations. Note that these relations are written with lowercase letters to distinguish them from the built-in relations. A fully reversible micro-Prolog factorial function can then be written by replacing the built-in functions with our declarative versions:

```
X has-factorial 1 if
     X is-less-than 1
X has-factorial Y if
     0 is-less-than X and
     X is-the-successor-of Z and
     Z has-factorial x and
     times (X x Y)
```

This definition now allows us not only to ask the forward question:

```
Which (x 3 has-factorial x)
```

and achieve the answer 6, but also to ask the backward question:

```
Which (x x has-factorial 6)
```

and achieve the answer 3.

As one might expect, the processing time needed to compute either of these answers is very slow. If you try it yourself, we recommend that you stick to the factorials of 0, 1, 2, and 3. Computing the factorial of 4 requires declaration of successor relationships up to 24 and takes several minutes to compute.

This performance problem, however, is partially due to the sequential nature of the systems on which micro-Prolog is currently implemented. When parallel machines are commonplace in the Fifth Generation, perhaps the promise of logic programming will make the Prolog twist a valuable technique rather than just an interesting exercise.

Further Reading

Clark, K.L., J.R. Ennals, and F.G. McCabe, 1982. *A micro-Prolog Primer.* Logic Programming Associates, London, England.

Clocksin, William F., and Christopher S. Mellish, 1981. *Programming in Prolog.* Springer-Verlag, Berlin, Heidelberg, New York.

Digital Equipment Corporation, 1982. *VAX-11 Basic Language Reference Manual.* Digital Equipment Corporation, Maynard, Massachusetts.

Ennals, Richard, 1983. *Beginning micro-Prolog.* Ellis Horwood, Ltd., West Sussex, England.

Jensen, Kathleen, and Niklaus Wirth, 1974. *Pascal User Manual and Report.* Springer-Verlag, Berlin, Heidelberg, New York.

Kowalski, R.A., 1979. *Logic for Problem Solving.* Elsevier North Holland, Inc., New York, Amsterdam, Oxford.

Winston, Patrick Henry, and Berthold Klaus Horn, 1981. *Lisp.* Addison-Wesley Publishing Company, Reading, Massachusetts.