**Recursive Modules**

Demonstrate recursion by using a simple function and we develop rules for writing recursive modules from this example. For example, the computer trying to multiply 8 by 3, assuming that the computer can do addition but not multiplication. The problem of multiplying 8 by 3 can be split into two problems as follow:

Problem 1 : Multiply 8 by 2.

Problem 2 : Add 8 to the result of problem 1.

Because your computer can add, so it can solve problem 2 but not problem 1. Problem 1, however, is simpler than the original problem. We can split problem 1 into two problems, 1.1 and 1.2, leaving three problems to solve, two of which are additions.

Problem 1.1 : Multiply 8 by 1.

Problem 1.2 : Add 8 to the result of problem 1.1.

Problem 2 : Add 8 to the result of problem 1.

Even though your computer cannot multiply, you can program it to recognize that the result of multiplying any number by 1 is that number. Therefore solving problem 1.1 (the answer is 8) and problem 1.2 gives the solution to problem 1 (the answer is 16). Solving problem 2 gives the final answer, 24.

int Multiply(int M, int N)

{

if (N = = 1)

return M; // STOPPING CASE

else

return M + Multiply(M, N-1); //RECURSIVE STEP

}

Problem 1 : Multiply M by N-1.

Problem 2 : Add M to the result.

The first of these problems is solved by calling MULTIPLY again with N - 1 as its second argument. If the new second argument is greater than 1, there will be additional calls to function MULTIPLY. We call this case (N > 1) the recursive step because it contains a call to function MULTIPLY.

Note that the two different uses of the identifier MULTIPLY in the recursive step. The one without parameters defines the function result; the other calls the function recursively.

When the condition N = 1 is finally true, the statement

return M // stopping case

executes, so the answer is M. We call this case (N = 1) the stopping case because it is always the last case reached.

**Tracing of a Recursive Function**

Hand-tracing an algorithm’s execution demonstrates how that algorithm works.

**Recursive Procedure**

**ReverseChar** is a recursive procedure that reads in a string of length N and prints it out backward. If the procedure call statement

ReverseChar (5)

is executed, the five characters entered are displayed in reverse order. For

example, if the characters a, b, c, d, e are entered in sequence when this procedure

is called, the lines

**a, b, c, d, e**

e, d, c, b, a

appear on the screen. The bold letters are entered as data and the second row of letters is the output. If the procedure call statement

ReverseChar (3)

is executed instead, only three characters are read, and the lines

a, b, c

c, b, a

will appear on the screen.

void ReverseChar(int N)

/\* Displays a string of length N in the reverse order from which it is

entered\*/

{

char Next ; //next data character

if (N == 1) // stopping case

{

scanf("%c", &Next);

printf("%c", Next);

}

else

{

scanf("%c", &Next);

ReverseChar (N - 1); // recursion

printf("%c", Next);

}

}

**Recursive Mathematical Functions**

Many mathematical functions are defined recursively. An example is the factorial of a number n (represented as n!):

Definition of n!

0! give a result 1

n! is equivalent to n \* (n - 1)!, for ALL n > 0

Thus,

4! = 4 \* 3!

3! = 3 \* 2!

2! = 2 \* 1!

so combining the above three statements,

4! = 4 \* 3 \* 2 \* 1!

It is easy to implement this recursive mathematics definition as a recursive function in Pascal. For example, the Function Factor in Figure 9-6 computes the factorial of its argument N. The recursive step return N \* Factor(N - 1); implements the second line of the factorial definition. This statement means that the result of the current call (argument N) is determined by multiplying the result of the next call (argument N-1) by N. Refer to the definition of factorial n!.