RECURSION

Recursion is a programming technique that enables the solution to certain problems to be computed in terms of solutions to smaller instances of the problem. Recursion offers precise and succinct descriptions of many algorithms, especially those that are naturally expressed in terms of solutions to subproblems. A procedure that successively calls itself a subprocedure is called a *recursive procedure* and the procedure call to the recursive procedure is called a *recursive call*. Recursive calls are usually made with smaller and simpler instances of the problem. In order to terminate the recursive calls, every recursive procedure must have a solution directly defined for at least one so-called *base case.* The base cases provide the foundations upon which the recursive solutions are computed.

For example, a function to compute x^y (assuming $x > 0$, $y \geq 0$, and x and y are integers) can be expressed recursively as shown in Fig. 1. In this example, the base case is $x^0 = 1$.

Such a function can be succinctly encoded in a programming language that supports recursion. Most modern programming languages (e.g., Pascal, C, C++, and Java) support recursion. Figure 2 shows a function written in Java which recursively computes x^y. However, if recursion is not directly supported by the language, as is the case with Fortran, recursion can be simulated using a stack. A stack is ideal for implementing recursion because it enables all of the subproblems to be stored and retrieved in the order in which they need to be solved. Recall that the solution to a recursive problem cannot be computed until the solutions to all of its subproblems are known. Using a stack, each new subproblem that is generated can be pushed onto the stack. When a base case is reached it provides the necessary information to solve the subproblem currently on the top of the stack. Hence, once a base case is reached, the subproblems stored on the stack can be successively popped and solved. As computation continues, additional subproblems may be encountered and the stack will then grow until a base case is again reached. The solution to a base case is again used to pop and solve successive subproblems. Once the last subproblem is popped off the stack and solved, the stack will be empty and the original problem will be solved.

```
power(x,y)   = 1                    if y = 0
             = power(x, y−1) * x    if y > 0
```

Figure 1. A recursive definition for computing x^y.

```
public static int power(int x, int y) {
    if (y == 0// precondition: x > 0) {
        return(1);
    } else {
        return(power(x, y − 1) * x);
    }
}
```

Figure 2. A Java function for computing x^y recursively.

Figure 3 illustrates how the computation of 2^7 proceeds using the recursive Java function `power(x,y)`. The left-hand side of this figure shows that each successive call is computing a solution to a smaller and smaller problem until the base case is reached. The base case is then solved directly (in this example $x^0 = 1$) and its value is returned to the point from which it was called in order to compute the solution for 2^1. This is in turn computed by using the result from computing $2^0 = 1$ and multiplying it by 2. Each successive computed value is returned to the point from which it was called until the value for the initial call is returned (at the top right of the figure).

Figure 4 provides a different illustration of how 2^7 would be computed using the example Java function `power(x,y)`. Again, we see that the function depends on a call to itself to compute the solution to a subproblem. In this instance, 2^7 is computed by first computing 2^6 and then multiplying that result by 2. However, the solution to 2^6 first requires computing a solution to 2^5. The value of 2^5 first depends on computing a solution for 2^4 and so on. The subproblems become progressively smaller until the base case is reached, for which the solution is known. At that point the known solution is returned and is used in computing the solution for the larger problem. The results of each subproblem are returned and used to solve progressively larger problems until the original problem has been solved.

Finally, note that there is a distinction between a function that is implemented recursively and a *recursive function.* The latter has a precise meaning in theoretical computer science, where it refers to the class of functions that can be computed by any computer. See COMPUTABILITY.

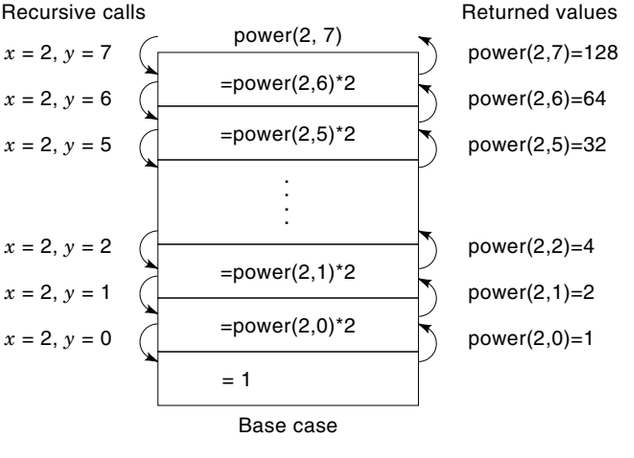

Figure 3. An example of recursive calls and return values.

```
power(2,7) = (power(2,6) * 2)
           = ((power(2,5) * 2) * 2)
           = (((power(2,4) * 2) * 2) * 2)
           = ((((power(2,3) * 2) * 2) * 2) * 2)
           = (((((power(2,2) * 2) * 2) * 2) * 2) * 2)
           = ((((((power(2,1) * 2) * 2) * 2) * 2) * 2) * 2)
           = (((((((power(2,0) * 2) * 2) * 2) * 2) * 2) * 2) * 2)
           = (((((((1 * 2) * 2) * 2) * 2) * 2) * 2) * 2)
           = ((((((2 * 2) * 2) * 2) * 2) * 2) * 2)
           = (((((4 * 2) * 2) * 2) * 2) * 2)
           = ((((8 * 2) * 2) * 2) * 2)
           = (((16 * 2) * 2) * 2)
           = ((32 * 2) * 2)
           = (64 * 2)
           = 128
```

Figure 4. Computing `power(2,7)`.

ACKNOWLEDGMENTS

T. Brecht and S. McIlraith gratefully acknowledge the support of the Natural Sciences and Engineering Research Council (NSERC). T. Pitassi's research is supported by NSF Grant CCR-9457782, US-Israel BSF Grant 95-00238, and Grant INT-9600919/ME-103 from NSF and MŠMT (Czech Republic).

BIBLIOGRAPHY

T. A. Standish, *Data Structures, Algorithms and Software Principles in C,* Reading, MA: Addison-Wesley, 1995.

T. A. Standish, *Data Structures in Java,* Reading, MA: Addison-Wesley, 1998.

N. Wirth, *Algorithms + Data Structures = Programs,* Englewood Cliffs, NJ: Prentice-Hall, 1976.

A helpful book on the subject of thinking recursively is:

E. S. Roberts, *Thinking Recursively,* New York: Wiley, 1986.

TIMOTHY BRECHT
University of Waterloo

SHEILA MCILRAITH
Stanford University

TONIANN PITASSI
University of Arizona