Recursion – A Mathematical Notion

Iterative Definition

- An iterative definition of a function is one that defines all the steps to execute explicitly one by one in order to get the final result

  – Fact(n) = n * (n-1) * (n-2) * … * 1

  – Search for a value in a set of numbers
    - Iterate over all numbers in the set (this is a loop)
      – compare the current number with the value
      – If found, return true
    - Return false

Recursive Definitions (1/3)

- Definition

  A recursive function is one that is defined by calling itself (one or many times) on a smaller subset of input

- Factorial
  - Fact(n) = Fact(n-1) * n
  - Fact(0) = 1

- Fibonacci
  - Fib(n) = Fib(n-2) + Fib(n-1)
  - Fib(0) = Fib(1) = 1

Recursive Definitions (2/3)

- Correctness
  - Stems from the axiom of induction (P is a predicate)
    1. Base case (P(0) is true)
    2. Induction step (P(n) => P(n+1) is true)

    Then for all n, P is true !

Recursive Definitions (3/3)

- Properties
  - All recursive definitions must END at some point !
  - There must be a « way out » of the sequence of recursive calls

- Factorial
  - $0! = 1$

- Fibonacci
  - $Fib(0) = 0$, $Fib(1) = 1$

Factorial Example

- Implement Factorial

- Let's have a look at the inner mechanisms !
  - Fact(4)

  - How is a recursive definition executed using a stack ?

Fibonacci Example

- Implement it

- Is it efficient ?
  - Execute Fib(10) !
  - What’s going on ?

Why Recursiveness ?

- Is it always possible and easy to find an iterative solution ?
  - « Towers of Hanoi » problem [next slide]

  - Is it easy to define an iterative solution ???

Towers of Hanoi (1/2)

- 3 pegs « A », « B » and « C »
- 5 disks of differing diameters are placed on peg « A » so that a larger disk is always below a smaller one

Goal : Move all disks to « C » using « B » as an auxiliary

Constraints :

1. *Only the top disk on any peg can be moved*
2. *A larger disk may never rest on a smaller one*

Towers of Hanoi (2/2)

- What if we had a solution for moving 4 disks from one peg to another ?
- A solution for 5 disks could probably use the solution for 4 disks !
  - Can you find it ?
  - Implement it !
    - void towers(int n, char frompeg, char topeg, char auxpeg)

Recursive Chains (1/3)

- A recursive function need not call itself directly
- « a » calls « b » and « b » calls « a » !

```
a(…) { … b(…)  }
b(…) { … a(…)  }
```

Recursive Definitions (2/3)

- Algebraic expressions
  1. An **expression** is a *term* followed by a « + » sign followed by a *term*, or a *term* alone
  2. A **term** is a *factor* followed by an asterisk followed by a *factor*, or a *factor* alone
  3. A **factor** is either a *letter* or an *expression* enclosed in parentheses

Recursive Definitions (3/3)

- Write a program that reads a character string and then prints « valid » or « invalid »
  - int getsymb(char *str, int length, int *ppos)
  - int expr(char *str, int length, int *ppos)
  - int term(char *str, int length, int *ppos)
  - int factor(char *str, int length, int *ppos)

Recursive vs Iterative

- Pros
  - Recursive definitions are in general natural expressions of the solution to implement (tower of hanoi, algebraic expressions, factorial, …)
- Cons
  - They are less efficient than iterative solutions
    - Memory consumption of stack frames
    - Slower due to function calls and especially unnecessary calls (e.g. fibonacci)

Simulating Recursion (1/3)

- Use of a stack to do the simulation
- What happens when a function is called ?
    1. Passing arguments
    2. Allocating and intializing local variables
    3. Transferring control to the function
        1. Save the return address
        2. Restitute the return value (if any) to the calling function

Simulating Recursion (2/3)

- Each time a recursive function calls itself
  - An entirely new data area (frame) is allocated
    - Arguments
    - Local and temporary variables
    - Return address
  - Return value is in a global variable (register)

- Be careful ! This is associated to every function CALL !

Simulating Recursion (3/3)

- Simulation can be used to find an optimized version of the recursive algorithm
  - Removing superfluous variables and stack operations
  - Tail recursion is to be transformed into an iteration

  - This could lead however to adding bugs to the program !
  - A recursive solution could become as efficient as a non recursive one