Topic 3
Linear Recursion and Iteration

September 2008

Processes generated by procedures

- Procedure: defined by code in a language
- Process: activities in computer while procedure is run

Recursion and iteration

- Recursive procedure: procedure calls itself in definition
- When recursive procedure runs, the activated process can be either iterative or recursive (in language like Scheme, Lisp and some other languages)

So which is it?

- Process is iterative if, whenever procedure calls itself, the value returned by that call is just returned immediately by procedure. (Example: **`compute-sqrt`**)
- Process is recursive if, for at least one instance when a procedure calls itself, the returned value from that call is used in some more computation before the procedure returns its value.

Computing factorial

```
n! = n * (n-1) * (n-2) * . . . * 1
```

Formally:

```
n! = 1                if n = 1
   = n * (n-1)!       if n > 1

; takes a pos integer and returns its fac
(define (fac n)
  (if (= n 1)
      1
      (* n (fac (- n 1)))))
```

Fac process is recursive

- Stack is needed to remember what has to be done with returned value.
- (fac 3)

```
(* 3 (fac 2))
     ( * 2 (fac 1))
           1
     (* 2 1)
(* 3 2)
6
```

Iterative factorial

- Instead of working from `n` **downward, we can work from `1` upward.**
- `; iterative version of factorial`
- `(define (fac n)(ifac 1 1 n))`
- `; helping fn for iterative version of factorial`

```
(define (ifac val cur-cnt max)
  (if (> cur-cnt max)
      val
      (ifac (* cur-cnt val)
            (+ cur-cnt 1)
            max)))
```

Ifac process is iterative

- **No stack needed to remember what to do with returned values.**

- `(fac 3)`

```
(ifac 1 1 3)
(ifac 1 2 3)
(ifac 2 3 3)
(ifac 6 4 3)
6
```

Some compilers are smart

- A smart compiler will convert `fac` **into something like:**

A: if cur-cnt > max, return val
val = cur-cnt * val
cur-cnt = cur-cnt + 1
goto A

Tail recursion

- When the value of a recursive procedure call is returned immediately, it is an instance of **tail recursion.**
- **Smart compilers know to write iterative loops whenever they find tail recursion.**
- **Some smart compilers: Scheme, Common Lisp, gcc, IBM JIT, C#**

Tree recursion

- A procedure that calls itself two or more times before returning a value is a **tree recursive procedure.**

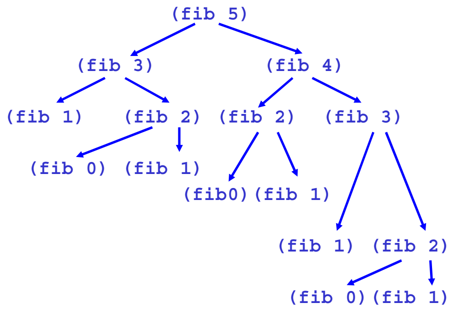
- **(If a procedure calls itself only once before returning a value, the procedure is generally a linear recursive procedure.)**

Fibonacci numbers

fib(n) = 0 if n = 0
= 1 if n = 1
= fib(n-1) + fib(n-2) if n > 1

```
; takes a pos int and returns the
; fibonacci #
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Tree of subproblems



(fib 5)

(fib 3) (fib 4)

(fib 1) (fib 2) (fib 2) (fib 3)

(fib 0) (fib 1)

(fib0) (fib 1)

(fib 1) (fib 2)

(fib 0) (fib 1)

Tree Recursion

- Even though it looks bad, it often is a very natural way to solve a problem.
- While it seem inefficient, there may be a way to create an iterative version.
- However, the recursive version often helps you think about the problem easier (and thus, that is what we will do – especially helpful if the data structure calls for it)

Fib (iterative version)

```
; takes a pos int and returns its
; fibonicci #
(define (fib n) (ifib 1 0 n))

; helping function - counts up
(define (ifib next-fib cur-fib cnt)
  (if (= cnt 0)
      cur-fib
      (ifib (+ next-fib cur-fib)
            next-fib
            (- cnt 1))))
```

Thinking about problems Recursively

- Let’s look at some problems that seem “hard” that are made much easier if we think about them recursively.

Towers of Hanoi

- Assume 3 pegs and a set of 3 disks (different sizes).
- Start with all disks on the same peg in order of size.
- Problem is to move them all to another peg, by moving one at a time, where no larger peg may go on top of a smaller peg.

Example

Move 3 disks, from peg 1 to peg 3 using peg 2 as extra.

(move-tower 3 1 3 2)

- Move top disk from 1 to 3
- Move top disk from 1 to 2
- Move top disk from 3 to 2
- Move top disk from 1 to 3
- Move top disk from 2 to 1
- Move top disk from 2 to 3
- Move top disk from 1 to 3

Recursive Program

- Are we done? Does this represent a base case?
- Otherwise,
- Formulate a problem that calls the same procedure again (wishful thinking) with an easier problem (one closer to the base conditions)

```
(define (move-tower size from to extra)
  (cond ((= size 0) #t)
        (else
         (move-tower (- size 1) from extra to)
         (move from to)
         (move-tower (- size 1) extra to from)
         )))
```

```
(define (move from to)
  (newline)
  (display "move top disk from ")
  (display from)
  (display " to ")
  (display to))
```

Problem reduction

To find a general solution to a problem:

- Break problem into subproblems that are easier to solve
- Combine solutions to the subproblems to create solution to original problem

Repeat on the subproblems until the problems are simple enough to solve directly

---

Procedure reduction

- Identify what subprocedures will be needed
- Write code for the procedure to combine the values returned by the subprocedures and return as the value of the procedure
- Repeat this process on each subprocedure until only predefined procedures are called
- Generally, simple cases are tested for first before any recursive cases are tried

Change counting problem

- This is an example of a procedure that is easy to write recursively, but difficult to write iteratively.
- Problem: Count the number of ways that change can be made for a given amount, using pennies, nickels, dimes, quarters, and half-dollars.

- E.g., number of ways to change 10 cents (5 coins)
- (10 pennies) or (1 nickel + 5 pennies) or (2 nickels) or (1 dime)

Strategy

- Divide ways of making change into disjoint sets that will be easier to count
- \# of ways =

\# of ways without using any coins of largest available denomination

\+

\# of ways that use at least one coin of largest available denomination

CC Examples

- (cc 5 2) ; count change 5 cents using 2 coins
- 1 (5 pennies) + 1 (1 nickel)

- (cc 10 2) ; count change 10 cents using 2 coins
- 1 (10 pennies) + 2 ways of making (cc 5 2)

- Number of ways of making change for a using n-1coins + Number of ways of making change for a - (value n) using n coins

Base Cases

- (cc amt numb-coins)

- If (= amt 0) 1
- If no coins or (< amt 0) 0

Change counting code

```
(define (cnt-chng amt) (cc amt 5))
(define (cc amt k)
  (cond ((= amt 0) 1)
        ((or (< amt 0) (= k 0)) 0)
        (else (+ (cc amt (- k 1))
                 (cc (- amt
                        (vk k))
                     k)))))
```

(continued)

```
(define (vk kinds);value of largest
  (cond ((= kinds 5) 50)
        ((= kinds 4) 25)
        ((= kinds 3) 10)
        ((= kinds 2)  5)
        ((= kinds 1)  1)
        (else 0)))
(cnt-chng 100)   292
```