

## 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

$$n! = n * (n-1) * (n-2) * \dots * 1$$

Formally:

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

```
(define (fac n)(ifac 1 1 n))
(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 `ifac` 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#

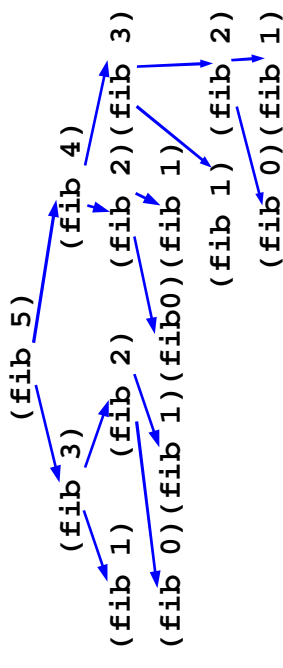
## Tree recursion

- 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
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```

## Tree of subproblems



## Some theory

- $\text{fib}(n)$  = the closest integer to  $\phi^n / \sqrt{5}$  where  $\phi = (1 + \sqrt{5}) / 2 = 1.6180\dots$
- $\phi$  is golden ratio ( $\phi^2 = \phi + 1$ )

## Fib (iterative version)

```
(define (fib n) (ifib 1 0 n))
(define (ifib next-fib cur-fib
  cnt)
  (if (= cnt 0)
      cur-fib
      (ifib (+ next-fib cur-fib)
            next-fib
            (- cnt 1))))
```

## 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.

## 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

## 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
```