More Procedures

CSE 413, Autumn 2002
Programming Languages

http://www.cs.washington.edu/education/courses/413/02au/

Readings and References

- Reading
  - » Section 1.2-1.2.2, *Structure and Interpretation of Computer Programs*, by Abelson, Sussman, and Sussman
- Other References
  - » Section 3, *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*

Abstraction is a good thing

- The span of absolute judgment and the span of immediate memory impose severe limitations on the amount of information that we are able to receive, process, and remember.
- By organizing the stimulus input simultaneously into several dimensions and successively into a sequence or chunks, we manage to break (or at least stretch) this informational bottleneck.
  - » Miller, 1956. see OtherLinks page for reference

A clean abstraction is a good thing

- One of the interesting and difficult things about software design is deciding how to chop up the system design in a "logical" fashion
- "Common sense" design is not always obvious
- Two useful goals
  - » Increase Cohesion
  - » Decrease Coupling

Cohesion and Coupling

- Cohesion describes the degree to which the various parts of a single conceptual object relate to one another in a logical way
  » a "cohesive design" is a good thing
- Coupling describes the degree to which different conceptual objects are tied together through implementation details and assumptions
  » a "highly coupled design" is a bad thing

Name space pollution

- One common problem that contributes to coupling between modules is naming
- As much as possible, you want to keep the details of your implementation from leaking out into the outside world
  » reduce conflict with other modules and reduce the complexity of your own design
  » make it possible to replace your implementation entirely with a new one that has the same external interface but completely different internals

Procedure names

- Recall that sqrta.scm defined a number of small auxiliary procedures to accomplish the task of calculating the square root
  » sqrt-iter, good-enough?, improve
- None of these procedures are of specific interest to the outside world
  » they interfere with other designs that want to build other procedures with the same names
  » the prefix "sqrt-" is clutter in our own design

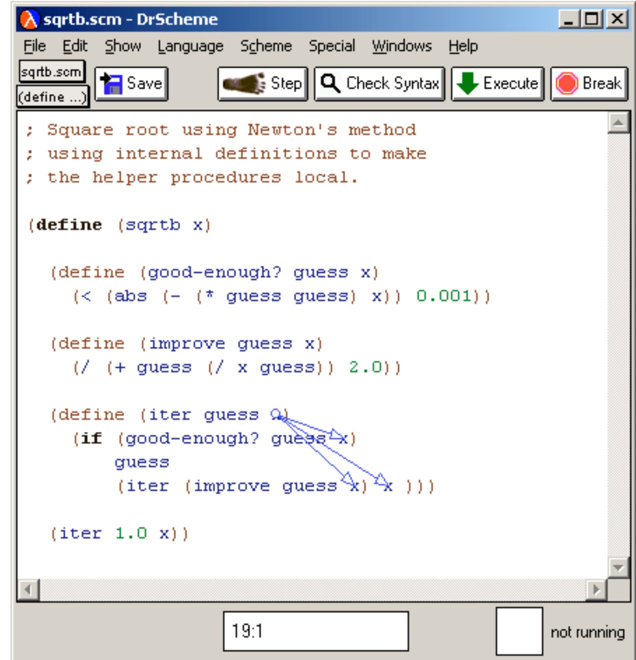
Helper definitions local to procedure

; Square root using Newton's method
; using internal definitions to make
; the helper procedures local.

```
(define (sqrtb x)

  (define (good-enough? guess x)
    (< (abs (- (* guess guess) x)) 0.001))

  (define (improve guess x)
    (/ (+ guess (/ x guess)) 2.0))

  (define (iter guess x)
    (if (good-enough? guess x)
        guess
        (iter (improve guess x) x )))

  (iter 1.0 x))
```

Local names

- The names of the helper procedures are now local to the define statement for sqrt
- The *scope* of the names is the define block
- Notice that the scope of the names of the formal parameters of each local procedure is the body of that procedure
  - » *the parameter names of a procedure are local to the body of the procedure*



Parameter names are local

```
(define (sqrtc x)

  (define (good-enough? ga xa)
    (< (abs (- (* ga ga) xa)) 0.001))

  (define (improve gb xb)
    (/ (+ gb (/ xb gb)) 2.0))

  (define (iter gc xc)
    (if (good-enough? gc xc)
        gc
        (iter (improve gc xc) xc )))

  (iter 1.0 x))
```

; Square root using Newton's method
; using internal definitions to make
; the helper procedures local.

; Replaced guess and x with ga, gb,
; gc and xa, xb, xc to highlight the fact
; that they are not all the same object.

Note that "x" is defined in the outer block and so it is visible to all of the helper procedures.

Do we need to pass x around from procedure to procedure?

Refer to variables in enclosing scope

```
(define (sqrtc x)

  (define (good-enough? ga xa)
    (< (abs (- (* ga ga) xa)) 0.001))

  (define (improve gb xb)
    (/ (+ gb (/ xb gb)) 2.0))

  (define (iter gc xc)
    (if (good-enough? gc xc)
        gc
        (iter (improve gc xc) xc )))

  (iter 1.0 x))
```

- xc is supplied to iter as a parameter.
- The value of that parameter is "x".
- iter calls itself recursively, and supplies the same value of "x" that it was given.
- Therefore, the value of "xc" is always "x", and we don't need to pass it as a parameter to procedure iter.

Refer to variables in enclosing scope

```
(define (sqrtd1 x)

  (define (good-enough? ga xa)
    (< (abs (- (* ga ga) xa)) 0.001))

  (define (improve gb xb)
    (/ (+ gb (/ xb gb)) 2.0))

  (define (iter gc)
    (if (good-enough? gc x)
        gc
        (iter (improve gc x))))

  (iter 1.0))
```

- xa is supplied to good-enough? as a parameter.
- The value of that parameter is always "x".
- Therefore, we don't need to pass it as a parameter to procedure good-enough?.

- xb is supplied to improve as a parameter.
- The value of that parameter is always "x".
- Therefore, we don't need to pass it as a parameter to procedure improve.

All x parameters replaced with global x

```
(define (sqrtd2 x)

  (define (good-enough? ga)
    (< (abs (- (* ga ga) x)) 0.001))

  (define (improve gb)
    (/ (+ gb (/ x gb)) 2.0))

  (define (iter gc)
    (if (good-enough? gc)
        gc
        (iter (improve gc))))

  (iter 1.0))
```

```
; Square root using Newton's method.
; Removed all the x parameters since they
; refer to the globally available x in
; every case.
```


Lexical scoping

- The preceding changes to the sqrt definition are examples of the use of *lexical scoping*
- Free variables (those that are not bound by the parameter list or a local define) are taken to refer to bindings made by enclosing procedure definitions
- The bindings are looked up in the environment in which the procedure was defined

Recursion and Iteration

- Definitions
  - » procedure (the text definition)
  - » process (the actual live action events)
- A recursive procedure (one that calls itself) does not necessarily generate a recursive process (one that has an open deferred operations remaining for each call)
- Many languages make the two always equivalent, but it is not necessary

Two implementations of factorial

```
; linear recursive

(define (facta n)
  (if (= n 1)
      1
      (* n (facta (- n 1)))))
```

We don't know what (facta (- n 1)) is until we have worked our way all the way down to facta(1). All the multiplications are deferred operations.

```
; iterative

(define (factb n)
  (define (iter prod count)
    (if (> count n)
        prod
        (iter (* count prod) (+ count 1))))
  (iter 1 1))
```

We are counting up. We know what 1*1 is, and we know what 1+1 is. So we can go directly from (iter 1 1) to (iter 1 2) to (iter 2 3) to (iter 6 4) etc.

Difference

- The key difference between the linear recursive process and the iterative process is this
  - » recursive - there are operations not yet completed which must be remembered by the system running the program - generally on a stack
  - » iterative - all of the state for the block of code can be captured in a finite set of variables - these variables are the arguments to the iterating function

Two implementations of simple counter

```
(define (print x)
  (display x))

; iterative process

(define (count1 x)
  (cond ((= x 0) (print x))
        (else (print x)
              (count1 (- x 1)))))

; linear recursive process

(define (count2 x)
  (cond ((= x 0) (print x))
        (else (count2 (- x 1))
              (print x))))
```

```
> (count1 4)
43210
> (count2 4)
01234
>
```

why?

Fibonacci Numbers

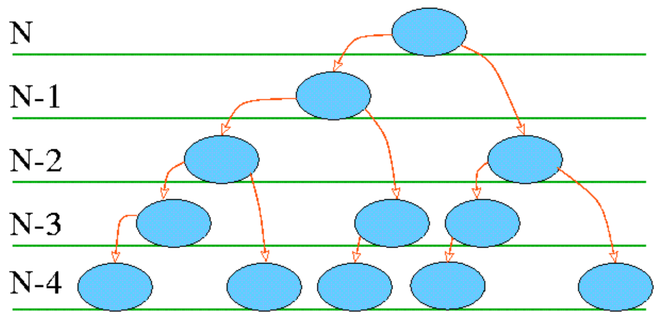
- Recall definition of Fibonacci numbers F_n

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Leonardo Pisano Fibonacci (1170-1250)

  - » First two are defined explicitly
  - » Rest are sum of preceding two
  - » $F_n = F_{n-1} + F_{n-2}$ (n > 1)
  - » sequence sometimes starts with 1, not 0

Recursive Calls of Fibonacci Procedure



- Re-computes fib(N-i) multiple times

Two implementations of Fibonacci

```
; tree recursive

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

; iterative

(define (fib-b n)
  (define (iter a b count)
    (if (= count 0)
        b
        (iter (+ a b) a (- count 1))))
  (iter 1 0 n))
```


Two implementations of Fibonacci

```
// tree recursive
int fib(int i) {
  if (i < 0) return 0;
  if (i == 0 || i == 1)
    return 1;
  else
  return fib(i-1)+fib(i-2);
}

// iterative
int fib_iter(int i) {
    int fib0 = 1, fib1 = 1, fibj = 1;
    if (i < 0) return 0;
    for (int j = 2; j <= i; j++) {
      fibj = fib0 + fib1;
      fib0 = fib1;
      fib1 = fibj;
    }
    return fibj;
}
```