Topic 2
Scheme and Procedures and Processes

September 2008

Substitution model for defined procedure application

Function definition:

```
(define (<fun> <param1> <param2> ...)
  <body>)
```

Function call:

```
(<fun> <expr1> <expr2> ...)
```

The substitution model

- Evaluate expressions <expr1>, <expr2>, ...
- Substitute the value of <expr1> for <param1>, the value of <expr2> for <param2>, ... in a copy of the <body> expression in the definition of <fun> to make a new expression
- Evaluate that expression

Substitution model example

```
(define (square x) (* x x))
```

Evaluation of `(square 2)` by substitution model:

```
(square 2)
(* 2 2)
4
```

Second substitution example

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))
```

```
(sum-of-squares 2 3)
(+ (square 2) (square 3))
(* 2 2)
4
  (* 3 3)
  9
(+ 4 9)
13
```

Third substitution example

```
(define (double-square x y)
  (+ (square x) (square y)))
```

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))
```

```
(sum-of-squares 10 10)
(+ (double-square 10 10) (sum-of-squares 10 10))
(+ (sum-of-squares 10 10) (double-square 10 10))
(* 10 10)
100
  (* 10 10)
  100
(+ 100 100)
200
```
Applicative order and normal order

- Applicative order: evaluate arguments, then apply procedure to values
- Normal order: substitute argument expressions for corresponding parameters in body of procedure definition, then evaluate body

Our substitution model of evaluation uses applicative order.

Conditional statements

Two forms:
1) `(if <test> <then-expr> <else-expr>)` NOT optional in Scheme

Example:
```
(define (absolute x)
  (if (< x 0)
      (- x)
      x))
```

2) `(cond (<test1> <expr1>)
        (<test2> <expr2>)
        ...
        (else <last-expr>))` NOT optional in Scheme

Example:
```
(define (absolute x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        (else (- x))))
```

Comments on conditionals

- A test is considered to be true if it evaluates to anything except `#f`
- A branch of a cond can have more than one expression: `(<test> <expr1> <expr2> . . . <exprN>)`
- `(<test>)` returns value of `<test>` if it is not `#f`
- The else branch must contain at least one expression

(Boolean functions)

- `(and <expr1> <expr2> . . . <exprN>)`<expr>s evaluated in order; return `#f` if any evaluate to `#f`, else return value of `<exprN>`

- `(or <expr1> <expr2> . . . <exprN>)`<expr>s evaluated in order; return the first value that is not `#f`; return `#f` if all are `#f`

(Confession)

Actually, I lied. The `<else-expr>` in `if` statements and the `else` branch in `cond` statements are optional, but the value that is returned is unspecified, so don’t omit them.
• \( \text{not} \ expr \) returns \#t or \#f as appropriate

**NOTE:** define, if, cond, and, or are special forms

**Difference between procedures and mathematical functions**

Mathematical functions are defined declaratively, by stating WHAT the conditions are that their values satisfy.

Example:

\( \text{sqrt}(x) = \text{the unique } y \text{ such that } y \geq 0 \text{ and } y \cdot y = x. \)

**Newton's method for square roots**

General approach:

• If a guess is good enough, return it.
• If not good enough, compute a better guess.
• Repeat
  (As a practical matter, we’ll only compute an approximate value.)

To compute \( \text{sqrt}(x) \) with guess=y, new guess is

\( \frac{y + x/y}{2} \)

I.e., average y with x/y

**Square root of 10**

Guess: 2
\( (2 + (10/2))/2 = 3.5 \)
\( (3.5 + (10/3.5))/2 = 3.1785 \)
\( \ldots \)
\( 3.162277660168 \ldots \)

**Newton's method (code)**

```scheme
(define (sqrt x)
  (compute-sqrt 1.0 x))

(define (compute-sqrt guess x)
  (if (good-enough-sqrt? guess x)
      guess
      (compute-sqrt
        (better-sqrt-guess guess x)
        x)))
```

```scheme
(define (good-enough-sqrt? guess x)\)
  (= guess (compute-sqrt guess x))

(define (better-sqrt-guess guess x)\)
  (/ (+ guess (quotient x guess)) 2)
```
(code continued)

(define (good-enough-sqrt? guess x)
  (< (abs (- x (square guess)))
      0.000001))

(define (better-sqrt-guess guess x)
  (average guess (/ x guess)))

(define (average x y)
  (/ (+ x y) 2))

Notice compute-sqrt is a recursive procedure

• Recursive procedure calls itself

Body
• Base conditions (am I done? – is this a problem so easy I can do right now with no work?)
• Otherwise, call itself on a simpler problem – one closer to the base condition

Each time recursive procedure is called, it is like a new procedure (variables are bound anew).

Brain Teaser – getting recursion

• What do the following procedures print when applied to 4? Note, I am not worried about the value returned, rather, about what is printed.
• First, try to predict what is printed.
• Second, try it in scheme.
• Third, if your prediction is different from what scheme produced, figure out why.
• Fourth, get help if it doesn’t make sense!

The Code

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

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

(define (print x)
  (display x)
  (newline))

Procedures as Black-Box abstractions

• A computing problem is often broken down into natural, smaller subproblems.
• Procedures are written for each of these subproblems.
• A procedure may call itself to solve a subproblem that is a smaller version of the original problem. This is called recursion.

Square root example

Subproblems (subprocedures)

sqrt
  compute-sqrt (also calls itself)
  good-enough?
  square
  better
  average

(primitives abs, /, +, - are also called)
Black box

Inputs → Output

We know what it does, not how

Procedures

Procedures are like Black Boxes. Their definitions (how they work) can be changed without affecting the rest of the program.

Example:
(define (square x)
  (exp (* 2 (log x))))

Procedural abstraction

• A user-defined procedure is called by name, just as primitive procedures are.
• How the procedure operates is hidden.

(square x)
(exp x)

Variables

• All symbols in a procedure definition are variables.
• All symbols in the procedure head (procedure name and parameters) are called bound variables.
• All occurrences of these variables in the body of the procedure definition are bound occurrences.
• All symbols in the body of the procedure that are not bound are called free variables.

Scope

• Bound variables in a procedure definition can be renamed without changing the meaning of the definition.
• The body of a procedure is the scope of the bound variables named in the procedure head.
• Changing the name of free variables will change the meaning of the definition.

Local variables

• The formal parameters of a procedure definition are local variables in the body.
• Other variables can become local variables by defining values for them; they become bound.

(define (area-of-circle radius)
  (define pi 3.14159)
  (* pi radius radius))
Procedure definitions can be nested

(define (sqrt x)
  (define (compute-sqrt guess x)
    (if (good-enough? guess x)
      guess
      (compute-sqrt
       (better guess x) x)))

(sqrt continued)

(define (good-enough? guess x)
  (or (< guess 1e-100)
      (< (abs (- (/ x (square guess))) 1))
      0.000001)))

(sqrt continued 2)

(define (better guess x)
  (average guess (/ x guess)))
  (compute-sqrt 1.0 x))

Locally defined procedures must come first in body of definition.

Block structure

- Procedure definitions are often called blocks
- The nesting of procedure definitions is called block structure
- Variables that are free in an inner block are bound as local variables in an outer block
- Values of local variables don't have to be passed into nested definitions via parameters
- This manner of determining the values of variables is called lexical scoping

Using fewer parameters

(define (sqrt x)
  (define (compute-sqrt guess)
    (define (good-enough?)
      (or (< guess 1e-100)
          (< (abs (- (/ x (square guess))) 1))
          0.000001)))

(fewer params continued)

(define (better)
  (average guess (/ x guess)))
  (compute-sqrt 1.0)))