

Computer Science I

Professor Tom Ellman
Lecture 15

Finding Both Solutions to a Quadratic Equation

```
(define quadratic-solutions (lambda (a b c) ...?...))
```

General Form: $ax^2 + bx + c = 0$

Example: $1x^2 + 0x + -4 = 0$

I.e. $x^2 = 4$

```
Welcome to Dr. Scheme.  
> (quadratic-solutions 1 0 -4)  
(2 -2)
```

Finding Both Solutions to a Quadratic Equation

```
(define quadratic-solutions  
  (lambda (a b c) (list (root1 a b c) (root2 a b c))))
```

```
(define root1  
  (lambda (a b c) (/ (+ (- b) (sqrt (discriminant a b c)))  
                    (* 2 a))))
```

```
(define root2  
  (lambda (a b c) (/ (- (- b) (sqrt (discriminant a b c)))  
                    (* 2 a))))
```

```
(define discriminant  
  (lambda (a b c) (- (square b) (* 4 (* a c)))))
```

Aren't we wasting (Scheme's) time?

- Notice that Scheme evaluates the expression (SQRT (DISCRIMINANT A B C)) twice.
 - Once in ROOT1.
 - Again in ROOT2.
- How can we avoid making Scheme evaluate the same expression twice?

(LET ...)

- Defines a *local variable*.
- Evaluates an expression to get a value.
- Assigns the value to the variable.
- A subsequent expression may use the variable many times.
- The variable's value is computed only once.

Finding Both Solutions to a Quadratic Equation

```
(define quadratic-solutions  
  (lambda (a b c)  
    (let ((dsqrt (sqrt (discriminant a b c))))  
      (list (root1 a b dsqrt) (root2 a b dsqrt)))))
```

```
(define root1  
  (lambda (a b dsqrt) (/ (+ (- b) dsqrt) (* 2 a))))
```

```
(define root2  
  (lambda (a b dsqrt) (/ (- (- b) dsqrt) (* 2 a))))
```

```
(define discriminant  
  (lambda (a b c) (- (square b) (* 4 (* a c)))))
```

General form of LET Expressions

Defining a single local variable:

```
(let ( (<variable> <expression> ) ) <body> )
```

Defining an arbitrary number of local variables:

```
(let ( (<variable-1> <expression-1>)  
      (<variable-2> <expression-2>)  
      ...  
      (<variable-3> <expression-N> ) )  
  <body> )
```

Behavior of LET Expressions

- Scheme evaluates each <EXPRESSION-I> to get a value.
- Scheme assigns the value to the corresponding <VARIABLE-I>.
- Then Scheme evaluates <BODY>.
 - The expression <BODY> may refer to any of <VARIABLE-1> ... <VARIABLE-N>.
 - Scheme uses the values it just assigned to these variables when evaluating <BODY>.

One More Thing about LET Expressions

- Each <EXPRESSION-I> must not contain a reference to <VARIABLE-I>.
- Each <EXPRESSION-I> must not contain a reference to any of the other variables <VARIABLE-1> ... <VARIABLE-N>.

REVERSE and REVERSE-HELPER

```
(define reverse  
  (lambda (lst) (reverse-helper lst '())))  
  
(define reverse-helper  
  (lambda (lst answer)  
    (if (null? lst)  
        answer  
        (reverse-helper (cdr lst)  
                          (cons (car lst) answer))))))
```

REVERSE with Local Definition of REVERSE-HELPER

```
(define reverse  
  (lambda (lst)  
    (letrec ((reverse-helper  
              (lambda (lst answer)  
                (if (null? lst)  
                    answer  
                    (reverse-helper (cdr lst)  
                                    (cons (car lst) answer))))))  
      (reverse-helper lst '()))))
```

General form of LETREC Expressions

Defining a single local recursive procedure:

```
(letrec ( (<variable> <expression> ) ) <body> )
```

Defining an arbitrary number of local recursive procedure:

```
(letrec ( (<variable1> <expression1>)  
        (<variable2> <expression2>)  
        ...  
        (<variable3> <expressionN> ) )  
  <body> )
```

Behavior of LETREC Expressions

- The same as the behavior of LET expressions.
- Except that a <VARIABLE-I> and an <EXPRESSION-I> may define a local recursive procedure.
- Therefore <EXPRESSION-I> may be a lambda expression containing a recursive call to the procedure named by <VARIABLE-I>.

Why Define a Local Function?

- Bundling a main procedure and a helper procedure into a single package.
- Allowing us to re-use the name of the helper procedure elsewhere in the program.
- Greater efficiency!

Counting the Number of Occurrences of an Item in a List

```
(define count (lambda (item lst) ...?...))
```

```
Welcome to Dr. Scheme.  
> (count 'e '(a k q r e d t e))  
2  
> (count 'e '(e (e) e))  
2  
(count 'e '())  
0
```

COUNT and COUNT-HELPER

```
(define count  
  (lambda (item lst) (count-helper item lst 0)))  
  
(define count-helper  
  (lambda (item lst cnt)  
    (cond ((null? lst) cnt)  
          ((equal? item (car lst))  
           (count-helper item (cdr lst) (+ 1 cnt)))  
          (else (count-helper item (cdr lst) cnt)))))
```

COUNT with Local Definition of COUNT-HELPER

```
(define count  
  (lambda (item lst)  
    (letrec ((count-helper  
              (lambda (item lst cnt)  
                (cond ((null? lst) cnt)  
                      ((equal? item (car lst))  
                       (count-helper item (cdr lst) (+ 1 cnt)))  
                      (else (count-helper item (cdr lst) cnt))))))  
      (count-helper item lst 0))))
```

We are wasting (Scheme's) time again!

- COUNT-HELPER takes ITEM as its first argument.
- When COUNT-HELPER calls itself, it uses ITEM as the first parameter sent to the recursive invocation of COUNT-HELPER.
- The parameter ITEM gets passed from one invocation of COUNT-HELPER to the next without being changed.
- How can we avoid this useless effort?

COUNT with Local Definition of COUNT-HELPER

```
(define count
  (lambda (item lst)
    (letrec ((count-helper
              (lambda (lst cnt)
                (cond ((null? lst) cnt)
                      ((equal? item (car lst))
                       (count-helper (cdr lst) (+ 1 cnt)))
                      (else (count-helper (cdr lst) cnt))))))
      (count-helper lst 0))))
```

Values of Non-Local Variables

- The (new) definition of COUNT-HELPER includes a reference to ITEM.
- The variable ITEM is no longer a parameter to COUNT-HELPER.
- ITEM is called a “non-local variable”.
- Where does ITEM get its value?
- From the variable called “ITEM” that is a parameter of the surrounding COUNT procedure definition.

The Global Environment

- The *global environment* is a table that holds the values assigned to variables.
- Each time Scheme processes a (DEFINE ...) expression typed into the top level prompt, Scheme adds a new entry to the table.

Welcome to Dr. Scheme.
 > (define <VAR-1> <VAL-1>)
 > (define <VAR-2> <VAL-2>)
 ...
 > (define <VAR-N> <VAL-N>)

| Global Environment | |
|--------------------|-------|
| VAR-1 | VAL-1 |
| VAR-2 | VAL-2 |
| ... | |
| VAR-N | VAL-N |

Welcome to Dr. Scheme.
 > (define w1 0.25)
 > (define w2 0.75)

| Global Environment | |
|--------------------|------|
| W1 | 0.25 |
| W2 | 0.75 |

Local Environment

- Suppose Scheme evaluates an expression: (<PROCEDURE> <ARG-1>...<ARG-N>).
- Suppose that <PROCEDURE> is defined by the expression (LAMBDA (<P-1>...<P-N>)<BODY>).
- Scheme creates a *local environment* in which each parameter <PI> is assigned the value of the corresponding argument <ARG-I>.
- Scheme evaluates <BODY> using the local environment in combination with the global environment.

```
(define <PROCEDURE> (lambda (<V-1> ... <V-N>) <BODY>))
```

```
(<PROCEDURE> <ARG-1>...<ARG-N>)
```

Environment for Evaluating <BODY>:

| Local Environment | | Global Environment | |
|-------------------|-------|--------------------|-------|
| V-1 | ARG-1 | VAR-1 | VAL-1 |
| V-2 | ARG-2 | VAR-2 | VAL-2 |
| ... | | ... | |
| V-N | ARG-N | VAR-N | VAL-N |

(define weighted-average (lambda (x y) (+ (* w1 x) (* w2 y))))

(weighted-average 75 85)

Environment for Evaluating (+ (* w1 x) (* w2 y)) :

| Local Environment | |
|-------------------|----|
| X | 75 |
| Y | 85 |

| Global Environment | |
|--------------------|------|
| W1 | 0.25 |
| W2 | 0.75 |

LET and LETREC Create New Local Environments

- When Scheme encounters a LET or LETREC it creates a new local environment.
- The new local environment hold definitions for the variables defined in the LET or LETREC expression.
- The new local environment is said to be “inside” of the current local or global environment.

(define weighted-average (lambda (x y) (let ((t1 (* w1 x)) (t2 (* w2 y))) (+ t1 t2))))

(weighted-average 75 85)

Environment for Evaluating (* w1 x) and (* w2 y) :

| Local Environment | |
|-------------------|----|
| X | 75 |
| Y | 85 |

| Global Environment | |
|--------------------|------|
| W1 | 0.25 |
| W2 | 0.75 |

Environment for Evaluating (+ t1 t2) :

| Local Environment | |
|-------------------|-------|
| T1 | 18.75 |
| T2 | 63.75 |

| Local Environment | |
|-------------------|----|
| X | 75 |
| Y | 85 |

| Global Environment | |
|--------------------|------|
| W1 | 0.25 |
| W2 | 0.75 |

(define weighted-average (lambda (x y) (let ((w1 0.33) (w2 0.67)) (+ (* w1 x) (* w2 y))))

(weighted-average 75 85)

Environment for Evaluating (+ (* w1 x) (* w2 y)) :

| Local Environment | |
|-------------------|------|
| W1 | 0.33 |
| W2 | 0.67 |

| Local Environment | |
|-------------------|----|
| X | 75 |
| Y | 85 |

| Global Environment | |
|--------------------|------|
| W1 | 0.25 |
| W2 | 0.75 |

Nesting Environments

- Each new local environment lies inside the previous one.
- To find the value of a variable, Scheme looks at the innermost environment first, and proceeds outward until finding a value for the variable.
- A new local variable may therefore make an existing local or global variable inaccessible.