CMPU 101 § 55

Computer Science I
Problem-Solving and Abstraction

Prof. Jonathan Gordon
Lecture 12

Locally defined data

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, it adds a new entry to the table, e.g.,

```
> (define w1 0.25)
> (define w2 0.75)
```

Global Environment	
w1	0.25
w2	0.75

We've seen that lambda expressions evaluate to procedures

```
> (lambda (x y) (+ x y))
#<procedure>
```

and the symbols in the argument list of a lambda expression play a special role:

```
> ((lambda (x y) (+ x y)) 1 2)
3
```

When the procedure is applied to a set of input expressions, the expressions in the body of the lambda expression are evaluated in order.

When those expressions are evaluated, occurrences of the argument symbols evaluate to the corresponding input expressions.

What happens if a function has an input with the same name as an entry in the global environment?

```
;; Define a global variable named x.
(define x 42)

;; Define a procedure that uses the symbol
;; x to represent its argument.
(define mult-by-10
  (lambda (x)
    (* x 10)))

> x
42

>(mult-by-10 1)
10
```

We see that x has different values depending on where it's being evaluated.

When it's the argument to `mult-by-10`, it has the value passed to the function.

```
;; Define a global variable named x.
(define x 42)

;; Define a procedure that uses the symbol
;; y to represent its argument -- but uses
;; x in its body.
(define mult-by-10-v2
  (lambda (y)
    (* x 10)))

> x
42

> (mult-by-10-v2 1)
420
```

When x isn't the name of an argument, it's evaluated in the normal way by looking it up in the global environment.

The input (y = 1) is ignored!

When a function defined by a lambda expression is called, a *local environment* is created.

Suppose Scheme evaluates an expression

(⟨***procedure***⟩ ⟨***arg-1***⟩ ... ⟨***arg-n***⟩)

and ⟨***procedure***⟩ is defined by the expression

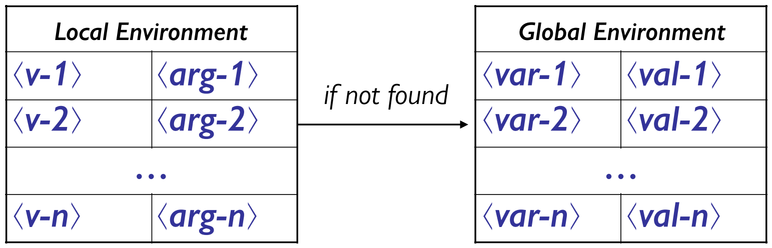
(lambda (⟨***p-1***⟩ ... ⟨***p-n***⟩) ⟨***body***⟩)

Scheme creates a local environment in which each parameter ⟨***p-i***⟩ is assigned the value of the corresponding argument ⟨***arg-i***⟩.

```
(define ⟨procedure⟩
  (lambda (⟨v-1⟩ … ⟨v-n⟩)
    ⟨body⟩)
```

```
(⟨procedure⟩ ⟨arg-1⟩ … ⟨arg-n⟩)
```

Environment for evaluating ⟨*body*⟩:



Local Environment	
⟨*v-1*⟩	⟨*arg-1*⟩
⟨*v-2*⟩	⟨*arg-2*⟩
…	
⟨*v-n*⟩	⟨*arg-n*⟩

if not found

Global Environment	
⟨*var-1*⟩	⟨*val-1*⟩
⟨*var-2*⟩	⟨*val-2*⟩
…	
⟨*var-n*⟩	⟨*val-n*⟩

When we use `define`, it creates an entry in the *current* environment – which may not be the global environment!

```
(define silly-func
  (lambda (x)
    (define y x)
    y))

> (silly-func 42)
42

> y
Error: y: undefined
```

☹

Because this is confusing, you should only use the `define` special form at the global level, i.e., not inside any other expression.

So, how should we create local variables?

Let there be `(let …)`

The `let` special form defines one or more *local variables*:

```
(let ((⟨variable 1⟩ ⟨expression 1⟩)
      (⟨variable 2⟩ ⟨expression 2⟩)
      …
      (⟨variable n⟩ ⟨expression n⟩))
  ⟨body 1⟩
  ⟨body 2⟩
  …
  ⟨body k⟩)
```

Evaluation of `let` special form

1. A new local environment is created inside the current environment.

2. Each symbol ⟨*variable i*⟩ is associated with the result of evaluating the corresponding ⟨*expression i*⟩.

3. The expressions ⟨*body 1*⟩ ... ⟨*body k*⟩ are evaluated in turn.

Whenever one of the symbols ⟨*variable i*⟩ must be evaluated, the corresponding value in the local environment is used.

Using `let`, we can give more meaningful names to expressions:

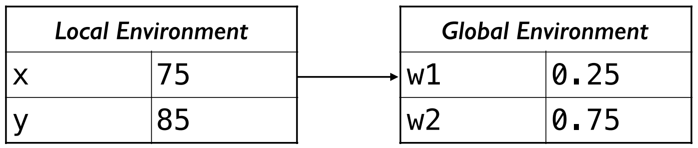
```
(define distance
  (lambda (pt1 pt2)
    (let ((x1 (first pt1))
          (y1 (second pt1))
          (x2 (first pt2))
          (y2 (second pt2)))
      (sqrt (+ (expt (- x2 x1) 2)
               (expt (- y2 y1) 2))))))

> (distance '(1 2) '(3 4))
2.8284271247461903
```

Local `let` environments supersede the global environment

```
> (define x 1000)
> (define y 100)
> (define z 10)
> (+ x y z)
1110
> (let ((x 3)
        (y 4))
    (+ x y z))
17
```

```
(define w1 0.25)
(define w2 0.75)
(define weighted-average
  (lambda (x y)
    (+ (* w1 x) (* w2 y))))
(weighted-average 75 85)
```

Environment for evaluating `(+ (* w1 x) (* w2 y))`:

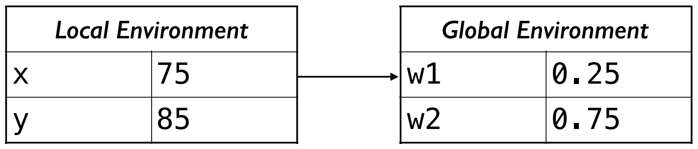


Local Environment	
x	75
y	85

Global Environment	
w1	0.25
w2	0.75

```
(define w1 0.25)

(define w2 0.75)

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

(weighted-average 75 85)
```

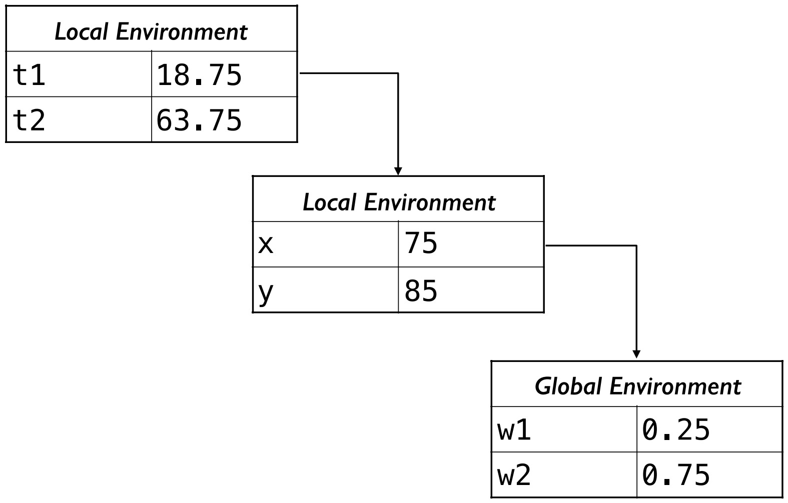
Environment for evaluating `(* w1 x)` and `(* w2 y)`:



Local Environment	
x	75
y	85

Global Environment	
w1	0.25
w2	0.75

Environment for evaluating `(+ t1 t2)`:



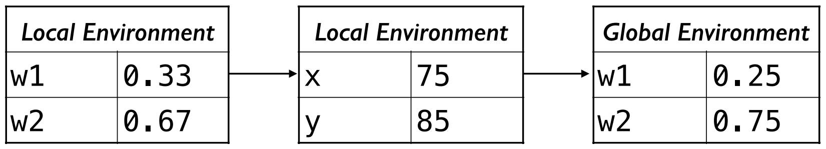
Local Environment	
t1	18.75
t2	63.75

Local Environment	
x	75
y	85

Global Environment	
w1	0.25
w2	0.75

```
(define w1 0.25)

(define 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.

`let` is just a convenient abbreviation

We can re-write any `let` statement in terms of the lambda expressions we've been using all along:

```
> (define z 1000)

> (let ((x 3)
        (y 4))
    (* x y z))
12000

> ((lambda (x y)
     (* x y z))
   3 4)
12000
```

The local variables we define in a `let` expression are all computed with respect to the *parent environment*, which might be the global environment, a `lambda` expression, or another `let` expression:

```
> (define x 100)

> (let ((x 44)
        (y (* x 2)))
    (list x y))
(44 200)
```

Often we want to carry out some complex computation incrementally, building up intermediate values that we'd like to store in separate local variables.

To do this with `let` statements, we need to nest them so each variable is defined in a local environment that inherits from the previous one.

We defined a function to calculate the distance between two points.

(Assuming a flat grid; a round planet makes things harder!)

What if we want to print our mileage from the start as we go on a trip?

```
(define print-mileage
  (lambda (start dest1 dest2)
    (let ((miles1 (distance start dest1)))
      (let ((miles2 (+ miles1 (distance dest1 dest2))))
          (let ((avg (/ (+ miles1 miles2) 2)))
            (printf "Leg 1: ~A~%Leg 2: ~A~%Avg: ~A~%"
                    miles1 miles2 avg)))))))
```

We can get the same effect using `let*` instead of nested `let` expressions.

In a `let*` expression, the values used to initialize a local variable can refer to the local variables that have already been given values (i.e., that appear earlier in the `let*` expression).

```
(define print-mileage
  (lambda (start dest1 dest2)
    (let* ((miles1 (distance start dest1))
           (miles2 (+ miles1 (distance dest1 dest2)))
           (avg (/ (+ miles1 miles2) 2)))
      (printf "Leg 1: ~A~%Leg 2: ~A~%Avg: ~A~%"
              miles1 miles2 avg)))))))
```

When a `let` special form is evaluated, the first thing that happens is that all of the initializing expressions are evaluated with respect to the parent context.

Only after that is the local environment created, containing entries for each of the variable–value pairs.

In contrast, when a `let*` special form is evaluated, the local environment is built incrementally as each initializing expression is evaluated.

Each initializing expression is evaluated with respect to the version of the local environment that's been created so far.

```
(let* ((a 1)
       (b (* a 2))    ; {a=1}
       (c (* a b 2))  ; {a=1, b=2}
       (d (* a b c))) ; {a=1, b=2, c=4}
  (list a b c d))
```

Locally defined procedures

When we've defined procedures at the global level, we've used `define` to provide a *name* for the procedure and a `lambda` expression to provide a specification for *what the procedure does*.

We can do the same thing using a `let` expression to define a *local function*.

A local function is convenient when you don't expect to use it elsewhere, e.g.,

```
(define cube-list
  (lambda (lst)
    (let ((local-cube-func
             (lambda (x) (* x x x))))
      (map local-cube-func lst))))

(cube-list '(1 2 3 4))
```

To define a local variable that's a recursive function, we need the `letrec` special form.

Why can't we use `let` or `let*` to do this?

General form of `letrec` expressions

Defining an arbitrary number of local variables including recursive procedures:

```
(letrec ((⟨variable 1⟩ ⟨expression 1⟩)
         (⟨variable 2⟩ ⟨expression 2⟩)
         ...
         (⟨variable n⟩ ⟨expression n⟩))
  ⟨body⟩)
```

Evaluation of `letrec` special form

1. The local environment is created first – before any of the initializing expressions have been evaluated.

2. Each of the variables is given a special placeholder value, `#<undefined>`.

3. Each of the initializing expressions is evaluated in turn with respect to the local environment.

Any variable can refer to any other variable – but it might get the value `#<undefined>` if that variable is defined after it!

`letrec` can do anything `let*` can do:

```
(let* ((x 3)
       (y (* x 4))
       (z (* y 1000)))
  (list x y z))
```

```
(letrec ((x 3)
         (y (* x 4))
         (z (* y 1000)))
  (list x y z))
```

But `letrec` can also deal with locally defining recursive functions, where the variable being defined is used in the expression being given for it.

When we used accumulators to solve problems, we often wrote a helper function whose only use was to be called by the corresponding wrapper function.

In these cases, it makes sense to define the helper function as a local procedure.

Old way

```
(define fact-helper
  (lambda (n acc)
    (if (< n 1)
        acc
        (fact-helper (- n 1) (* acc n)))))

(define fact
  (lambda (n)
    (fact-helper n 1)))
```

New way

```
(define fact
  (lambda (n)
    (letrec ((helper
               (lambda (m acc)
                 (if (< m 1)
                     acc
                     (helper (- m 1)
                             (* acc m))))))
      (helper n 1))))
```

reverse and reverse-helper

```
(define reverse-helper
  (lambda (lst answer)
    (if (null? lst)
        answer
        (reverse-helper (rest lst)
                        (cons (first lst)
                              answer)))))

(define reverse
  (lambda (lst)
    (reverse-helper lst '())))
```

reverse with local definition of reverse-helper

```
(define reverse
  (lambda (lst)
    (letrec ((helper
               (lambda (lst answer)
                 (if (null? lst)
                     answer
                     (helper (rest lst)
                             (cons (first lst)
                                   answer))))))
      (helper lst '()))))
```

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) ...?...)


> (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 (first lst))
           (count-helper item (rest lst) (+ 1 cnt)))

          (else
           (count-helper item (rest lst) cnt)))))
```

`count` with local `helper`

```
(define count
  (lambda (item lst)
    (letrec ((helper
               (lambda (item lst cnt)
                 (cond ((null? lst)
                        cnt)

                       ((equal? item (first lst))
                        (helper item (rest lst)
                                 (+ 1 cnt)))

                       (else
                        (helper item (rest lst)
                                cnt))))))

      (helper item lst 0))))
```

We are wasting (Scheme's) time again!

`helper` takes `item` as its first argument.

When `helper` calls itself, it uses `item` as the first parameter sent to the recursive invocation of `helper`.

The parameter `item` gets passed from one invocation of `helper` to the next without being changed.

How can we avoid this useless effort?

`count` with local helper (referencing a non-local variable)

```
(define count
  (lambda (item lst)
    (letrec ((helper
               (lambda (lst cnt)
                 (cond ((null? lst)
                        cnt)

                       ((equal? item (first lst))
                        (helper (rest lst) (+ 1 cnt)))

                       (else
                        (helper (rest lst) cnt))))))

      (helper lst 0))))
```

Values of non-local variables

The (new) definition of `helper` includes a reference to `item`.

The variable `item` is no longer a parameter to `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.

Acknowledgments

This lecture incorporates material from:

- Tom Ellman
- Luke Hunsberger