Lecture 19: Stateful Evaluation

cs1120 Fall 2009
David Evans
http://www.cs.virginia.edu/evans

Menu

- Stateful Evaluation Rules
- Exam 1

Names and Places

A **name** is a **place** for storing a value.

define creates a new place

(set! *name expr***)** changes the value in the place *name* to the value of *expr*

mcons creates a mutable pair containing two new places

(set-mcar! *pair expr***)** changes the value in the **mcar** place of *pair* to the value of *expr*

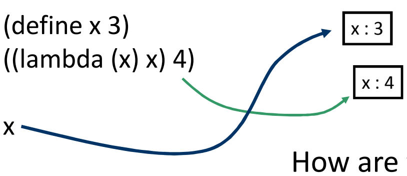
(set-mcdr! *pair expr***)** changes the value in the **mcdr** place of *pair* to the value of *expr*

Lambda and Places

(lambda (x) …) also creates a new place named **x**

The passed argument is put in that place

```
> (define x 3)
> ((lambda (x) x) 4)
4
> x
3
```

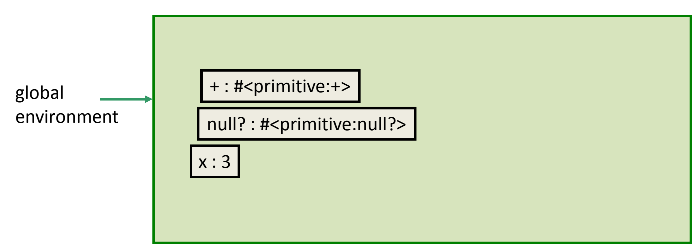


How are these places different?

Location, Location, Location

- Places live in **frames**
- An **environment** is a frame and a pointer to a parent environment
- All environments except the global environment have exactly one parent environment, global environment has no parent
- Application creates a new environment

Environments



The global environment points to the outermost frame. It starts with all Scheme built-ins defined.

> (define x 3)

Stateful Definition Evaluation Rule

A definition creates a new place with the definition's name in the frame associated with the evaluation environment. The value in the place is value of the definition's expression.

If there is already a place with the name in the current frame, the definition replaces the old place with a new place and value.

Stateful Name Evaluation Rule

To evaluate a name expression, search the evaluation environment's frame for a place with a name that matches the name in the expression.

If such a place exists, the value of the name expression is the value in that place.

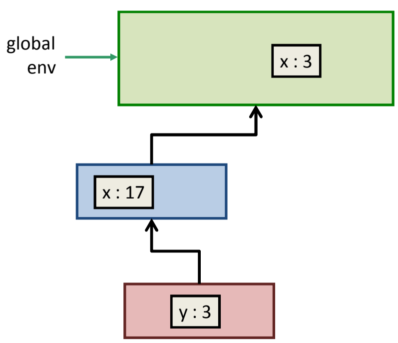
Otherwise, the value of the name expression is the result of **evaluating the name expression in the parent environment.** If the evaluation environment has no parent, the name is not defined and the name expression evaluates to an error.

Evaluating Names

To evaluate a name expression, search the evaluation environment's frame for a place with a name that matches the name in the expression.

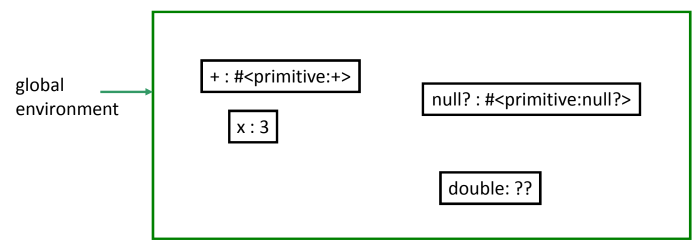
If such a place exists, the value of the name expression is the value in that place.

Otherwise, the value of the name expression is the result of **evaluating the name expression in the parent environment.** If the evaluation environment has no parent, the name is not defined and the name expression evaluates to an error.



How are environments like this created?

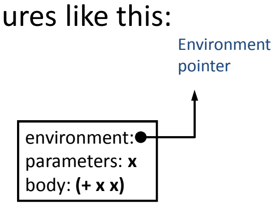
Procedures



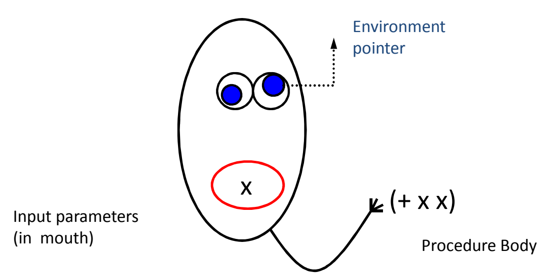
> (define double (lambda (x) (+ x x)))

How to Draw a Procedure

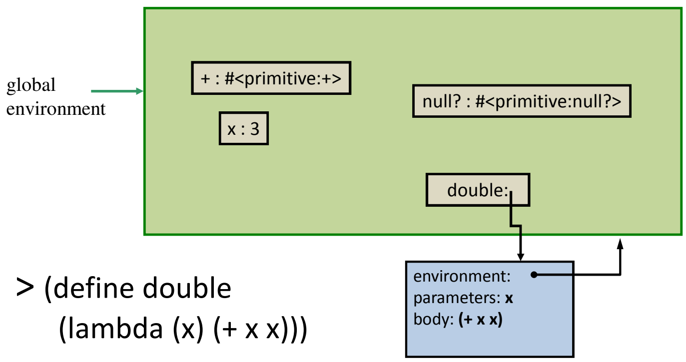
- A procedure needs **both code and an environment**
  - We'll see why soon
- We draw procedures like this:



How to Draw a Procedure (for artists only)



Procedures



> (define double
  (lambda (x) (+ x x)))

Application

- Old rule: (Substitution model)

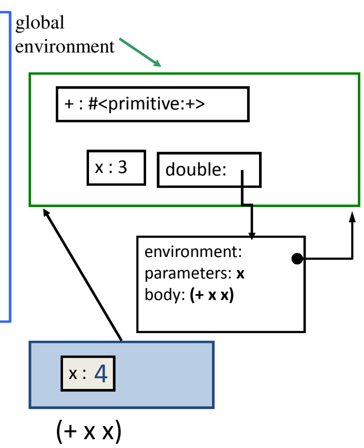
  Apply Rule 2: Constructed Procedures. To apply a constructed procedure, **evaluate** the body of the procedure with each formal parameter replaced by the corresponding actual argument expression value.

Stateful Application Rule (Constructed Procedures)

To apply a constructed procedure:

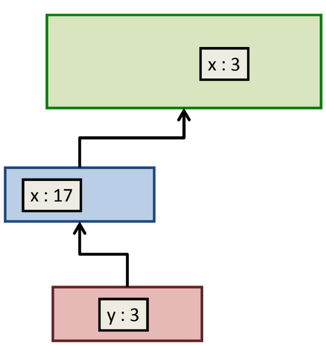
1. **Construct a new environment**, whose parent is the environment of the applied procedure.
2. **For each procedure parameter, create a place** in the frame of the new environment with the name of the parameter. Evaluate each operand expression in the environment or the application and initialize the value in each place to the value of the corresponding operand expression.
3. **Evaluate** the body of the procedure **in the newly created environment.** The resulting value is the value of the application.

1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment



> (double 4)
8

What would create this environment?



Think about this, we'll discuss in Monday's class...

Exam 1

- Overall: very good
  - Average: 93
  - Average for Q's 3,4,8 (defining procs): 8.9
  - Average for Q's 5,6,7,9 (analysis): 7.8
- Main complaints:
  - Hard to program without Scheme interpreter
  - Too much emphasis on runtime analysis

Question 10: count-unique

Define a procedure, *count-unique*, that takes as input a list of numbers. It produces as output a number that indicates the number of unique numbers in the input list. So,

(*count-unique (list 1 1 2 0))* should evaluate to 3.
(*count-unique (list 2 2 2))* should evaluate to 1.
(*count-unique (list 1 2 1 2 1))* should evaluate to 2.

For full credit, your procedure must work correctly for all possible inputs that are Lists of numbers.

count-unique: hard and slow way

From Chapter 5:

```
(define (list-filter test p)
  (if (null? p) null
    (if (test (car p))
      (cons (car p) (list-filter test (cdr p)))
      (list-filter test (cdr p)))))
```

```
(define (count-unique p)
  (if (null? p) 0
     (+ 1 (count-unique
            (list-filter
              (lambda (el) (not (= el (car p))))
              (cdr p))))))
```

Running time is in θ(N^2) where *N* is number of elements in *p*.
Worst case: no duplicates.
There are *N* recursive calls, each calls *list-filter* which has running time in θ(*N*).
Assumes = is constant time: only true if elements of *p* are bounded (always below some max value)

count-unique: easier, faster way

- Observe: if elements are sorted, don't need to search entire list to find duplicates

```
(define (count-unique p)
  (- (length p) (count-repeats (sort p <))))
```

Running time is in θ(*N* log *N*).
Body of *count-unique* applies sort (to a list of length N), count-repeats (to a list of length N), and length (to a list of length <= N):

θ(*N* log *N*) + θ(*N*) + θ(*N*) = θ(*N* log *N*)

Assumes: all values in p below some fixed constant *C* (needed for < to be constant time).

count-unique: "fastest" way

Assumes: all values in p below some fixed constant *C* (needed for < to be constant time).

```
(define C 100)

(define (count-unique p)
  (length
    (list-filter
      (lambda (n) (list-contains? p n))
      (intsto C))))
```

C executions of list-contains? which has running time in θ(*N*).
Running time is in θ(*N*).

Is this really the fastest?

Do you trust your classmates to follow the honor expectations in this class?

___ Yes, I trust them completely

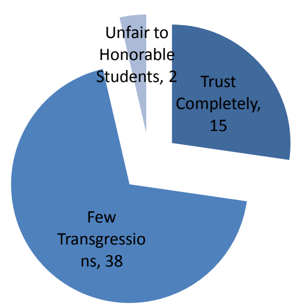
___ I worry that there may be a few transgressions, but I believe the vast majority of the class is honorable and it is fair and beneficial to rely on this.

___ I think this class places too high a burden on students' honor, and there are enough dishonorable students that it is unfair on the honorable students.

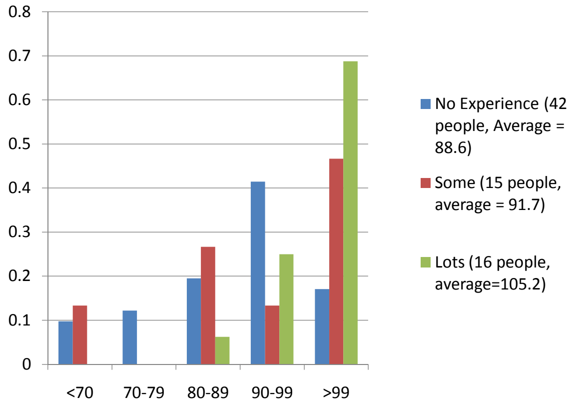
0___ I have direct knowledge of other students violating the honor policy on problem sets.

0___ I have direct knowledge of other students violating the honor policy on this exam.

Honor Expectations



Exam 1 Distribution



Charge

- Return Exam1 and PS4 now
- Read the Exam1 Comments
  - If there are things that don't make sense after reading them, come to office hours or send me email
- You know everything you need for PS5 now
- Next week: programming with mutation