CS450 - Structure of Higher Level Languages

The Environment Model

October 14, 2020

The Environment Model

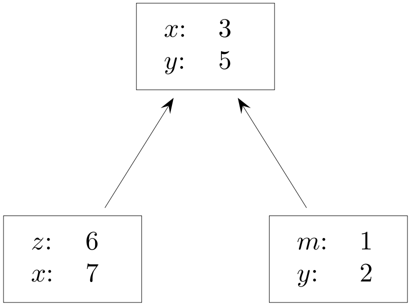
- So far we used the substitution model to evaluate compound procedures:
- To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.
- The substitution model we used so far is no longer adequate after introducing assignments (set!).
- A variable must somehow designate a "place" in which values can be stored.
- This is where the environment model comes in.

Environments

- An *environment* is a tree of *environment frames*.
- A frame is:
  - a (possibly empty) table of variables and their associated values (i.e., bindings), together with
  - a pointer to its parent in the tree of environment frames. This parent is called the *enclosing environment*. Of course there is one exception: the root frame, which is also called the *global environment* has no parent, and so has no such pointer.
- At each point during the execution of a program, we have a *current environment*. This is one of the frames in the tree.

Evaluating a variable

- The value of a variable is found by starting with the current environment and walking up the tree until the variable is found.
- Here is how it works: *x* may have the value 7 or 3, depending on the current environment; similarly for *y*



Evaluating a lambda expression

To evaluate a lambda expression, create a *procedure object* consisting of

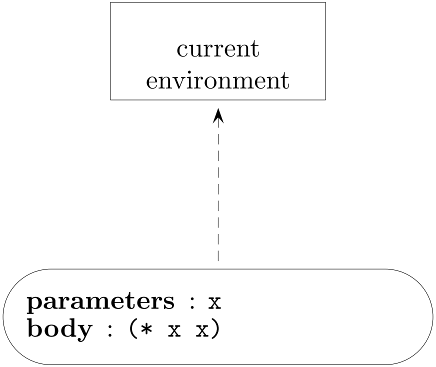
- the text of the procedure. This in turn consists of
    - the formal parameters of the procedure, and
    - the body of the procedure.

  It is important to remember that this is just copied textually—nothing in the text of the procedure is evaluated at this point.
- a pointer (or more accurately, a reference) to the environment in which the lambda expression was evaluated.

This procedure object is what the lambda expression evaluates to.

Evaluating a lambda expression

Evaluation of the lambda expression `(lambda(x) (* x x))`:



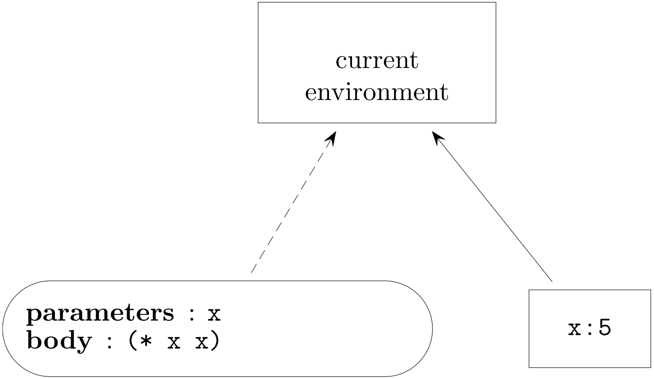
Evaluating a (user-defined) procedure call

To evaluate a procedure call (where the procedure is user-defined, and hence evaluates to a procedure object),

1. Evaluate the first expression in the list. This is the procedure itself, and so it evaluates to a procedure object, as above.
2. Evaluate the rest of the expressions of the list—these are the actual arguments to the procedure—in the current environment.
3. Construct a new frame containing the bindings of the formal parameters of the procedure to the corresponding values just produced in step 2.
   The enclosing environment of this frame is the environment part of the procedure object produced in step 1.
4. Evaluate the body of the procedure in this new environment.

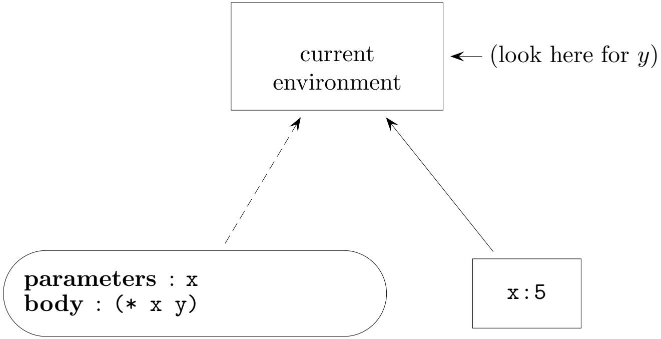
Evaluating a (user-defined) procedure call

Applying a lambda expression: `((lambda(x) (* x x)) 5)`:



Evaluating a (user-defined) procedure call

Applying a lambda expression: `((lambda(x) (* x y)) 5)`:

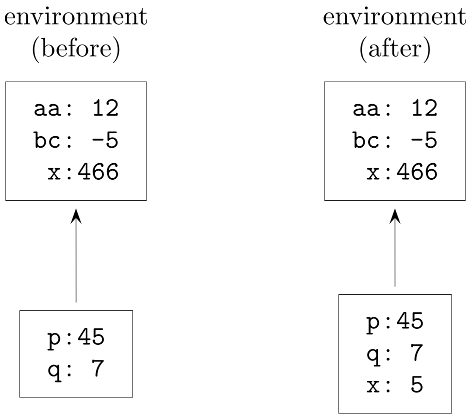


Conventions Used in the Pictures

- **Frames are represented by rectangles.** Frames are the *only* nodes in the environment tree.
- **The parent-child relation** between frames is represented by a solid arrow.
- **Procedure objects are represented by ovals.** This is to emphasize that procedure objects are *not* nodes in the environment tree. Each procedure object does point to a node (i.e., a frame) in the environment tree. This frame is called the "environment of the procedure object". It is *not*, however, the "parent" of the procedure object, since the procedure object is not a node in the tree.
- **The arrow from a procedure object to its environment frame is dotted.** This is to reinforce the fact that the arrow does not represent a parent-child relation.

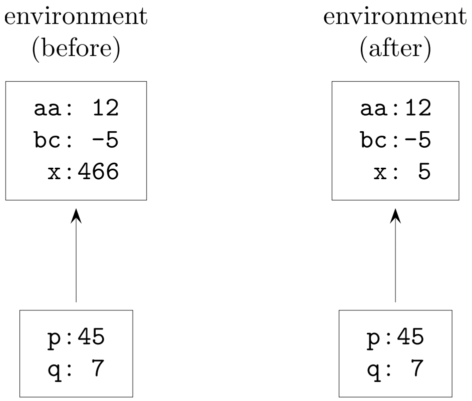
Evaluating `define` and `set!`

To evaluate `(define x <exp>)`, evaluate `<exp>`, and add a binding for `x` to the value of `<exp>` to the current frame. Thus, `(define x 5)` adds a binding to the current frame.



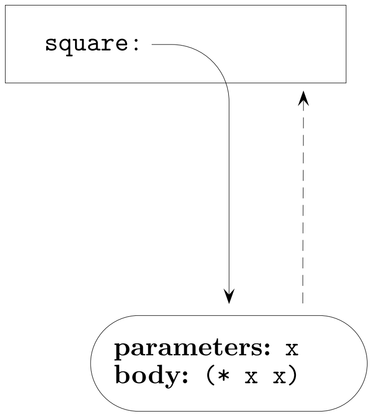
Evaluating `define` and `set!`

To evaluate `(set! x <exp>)`, evaluate `<exp>`, search up in the environment for `x`, and change the value bound to `x`. Thus, `(set! x 5)` changes the binding of `x` in the first frame in which `x` is found.

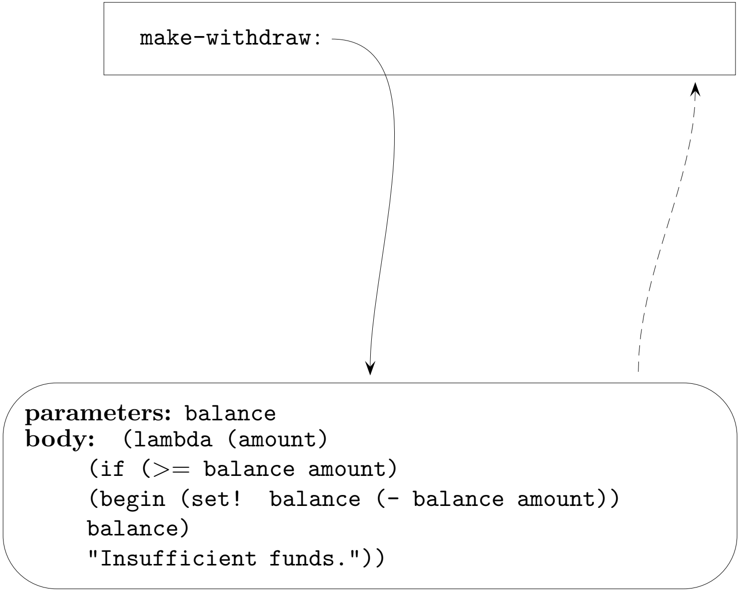


Another View of Functions

`(define (square x) (* x x))`, which is the same as `(define square (lambda (x) (* x x)))`



Some larger examples



parameters: balance
body:

```
(lambda (amount)
    (if (>= balance amount)
    (begin (set!  balance (- balance amount))
    balance)
    "Insufficient funds."))
```

Make Withdraw Example

The figure above shows:

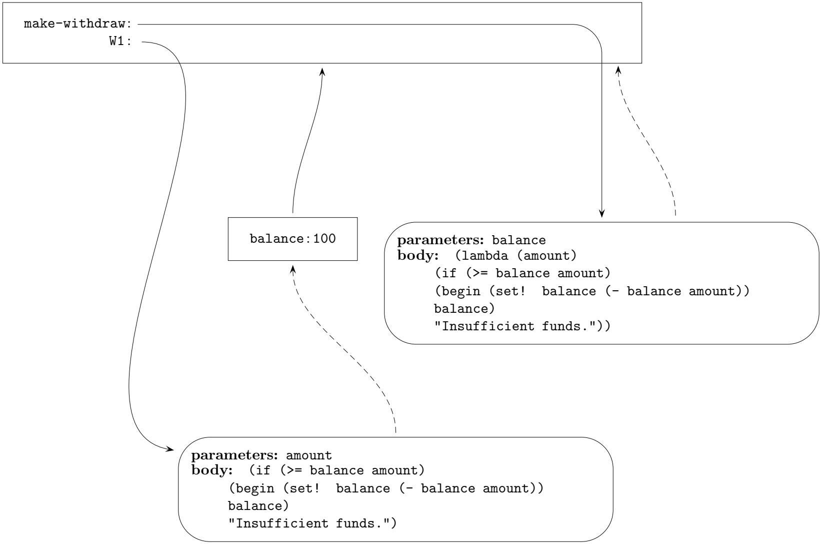
```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds.")))
```

This is equivalent to

```
(define make-withdraw
  (lambda (balance)
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds."))))
```

Make Withdraw Example

This is what `(define W1 (make-withdraw 100))` looks like:



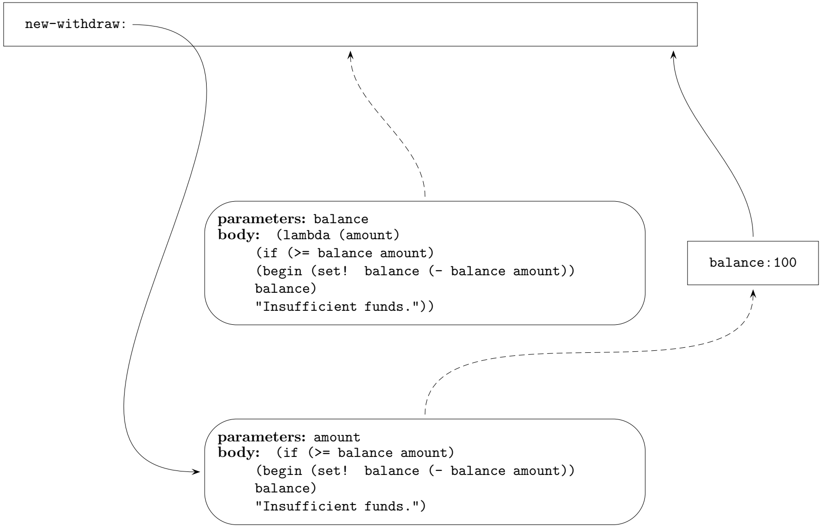
New Withdraw

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds."))))
```

This is immediately turned internally into

```
(define new-withdraw
  ((lambda (balance)
     (lambda (amount)
       (if (>= balance amount)
           (begin (set! balance (- balance amount))
                  balance)
           "Insufficient funds.")))
   100))
```

New Withdraw



parameters: balance
body:
```
(lambda (amount)
    (if (>= balance amount)
    (begin (set!  balance (- balance amount))
    balance)
    "Insufficient funds."))
```

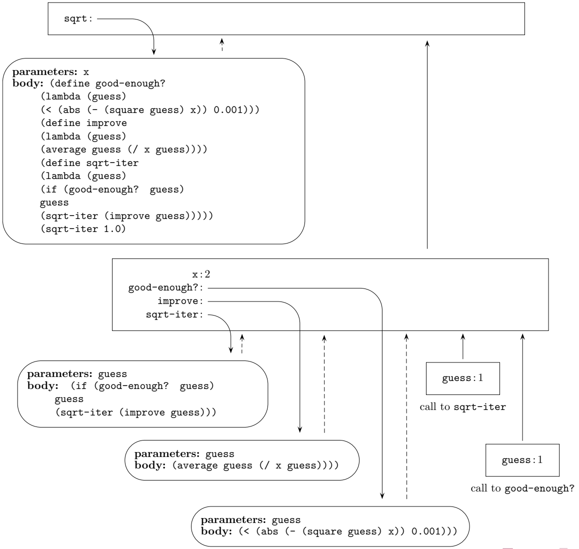
parameters: amount
body:
```
(if (>= balance amount)
    (begin (set!  balance (- balance amount))
    balance)
    "Insufficient funds.")
```

Evaluation of sqrt

```
(define sqrt
  (lambda (x)
    (define (good-enough? guess)
      (< (abs (- (square guess) x)) 0.001))
    (define improve
      (lambda (guess)
        (average guess (/ x guess))))
    (define sqrt-iter
      (lambda (guess)
        (if (good-enough? guess)
            guess
            (sqrt-iter (improve guess)))))
    (sqrt-iter 1.0)))
```

Evaluation of (sqrt 2)

Only the first few steps of the computation are shown



sqrt:
parameters: x
body: (define good-enough?
(lambda (guess)
(< (abs (- (square guess) x)) 0.001)))
(define improve
(lambda (guess)
(average guess (/ x guess))))
(define sqrt-iter
(lambda (guess)
(if (good-enough? guess)
guess
(sqrt-iter (improve guess)))))
(sqrt-iter 1.0)
x: 2
good-enough?:
improve:
sqrt-iter:
parameters: guess
body: (if (good-enough? guess)
guess
(sqrt-iter (improve guess)))
parameters: guess
body: (average guess (/ x guess))))
parameters: guess
body: (< (abs (- (square guess) x)) 0.001)))
guess: 1
call to sqrt-iter
guess: 1
call to good-enough?