CS 360: Programming Languages
Lecture 5: Environment Model of Evaluation

Geoffrey Mainland

Drexel University

Section 1

Administrivia

Administrivia

- Lab 2 due Monday, January 28 at 11:59pm.
- Office hours **3–5pm** on Monday (faculty candidate).

Homework 1

- The bodies of my solutions were each 3 lines long except for choose, which took 5 lines.
- No helpers were necessary.
- Style comments (see this style guide)
  - Parentheses should not appear on a line by themselves.
  - Use an `if` expression instead of `cond` when there are only two clauses.
  - If there are more than two clauses, use `cond`.
- Consider: `(append (list z) ...)` vs. `(cons z ...)`
- *Please* format reported times so we can automatically parse them :)

```
Problem 1: 15min

My comments about Problem 1.
```

Section 2

A New Model of Evaluation

The old model for compound procedure application...

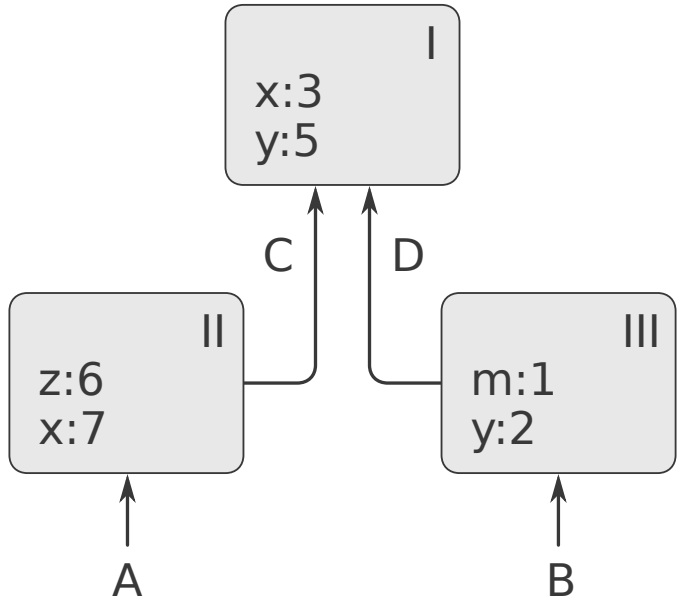
To apply a procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

- With assignment, this no longer works.
- In particular, a variable is no longer a value but must designate a "location" in which a value is stored.
- In our new evaluation model, these locations will be maintained in a structure called an **environment**.

(Abstract) Environment

- An environment is a sequence of **frames**.
- Each frame is a (possibly empty) table of **bindings**, which associate variable names with their values.
- Each frame also points to its **enclosing environment** unless it happens to be the global environment.
- The **value of a variable** with respect to an environment is the value given to the binding in the first frame in the environment that contains a binding for that variable.
- If no frame in the environment contains a binding for a variable, the variable is **unbound**.
- Note that a variable only has a value with respect to an environment.

Example environment



- What are the frames? What are the pointers to environments?
- Where are x, y, and z bound?
- Name an unbound variable.
- What is the value of x in the environment D? How about in A?
- With respect to environment A, the binding of x to 7 in frame II **shadows** the binding of x to 3 in frame I.

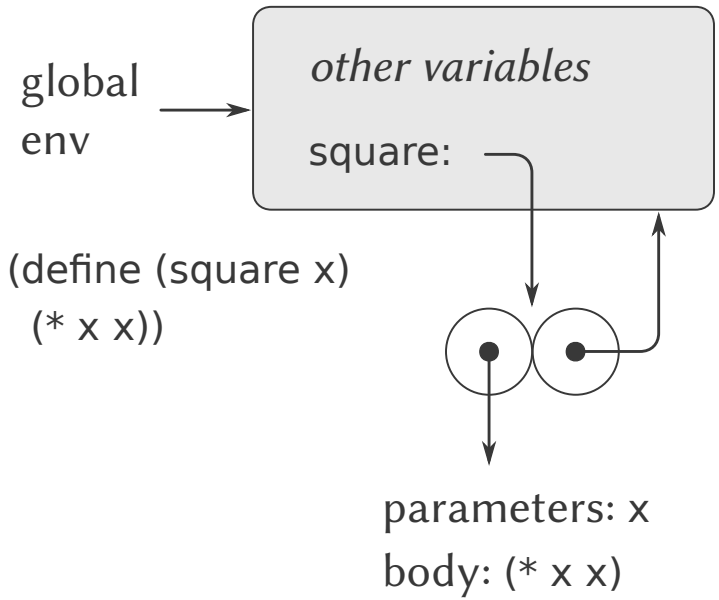
The environment model of evaluation for compound expressions

To evaluate a compound expression:

1. Evaluate the subexpressions.
2. Apply the value of the operator to the values of the operand subexpressions.

Question: Does order of evaluation matter? Did it in the substitution model?
But how do we apply a compound procedure? We can't use substitution...
In the environment model of evaluation, a compound procedure is always a *pair of code and an environment*.

Compound procedure representation in the environment model of evaluation



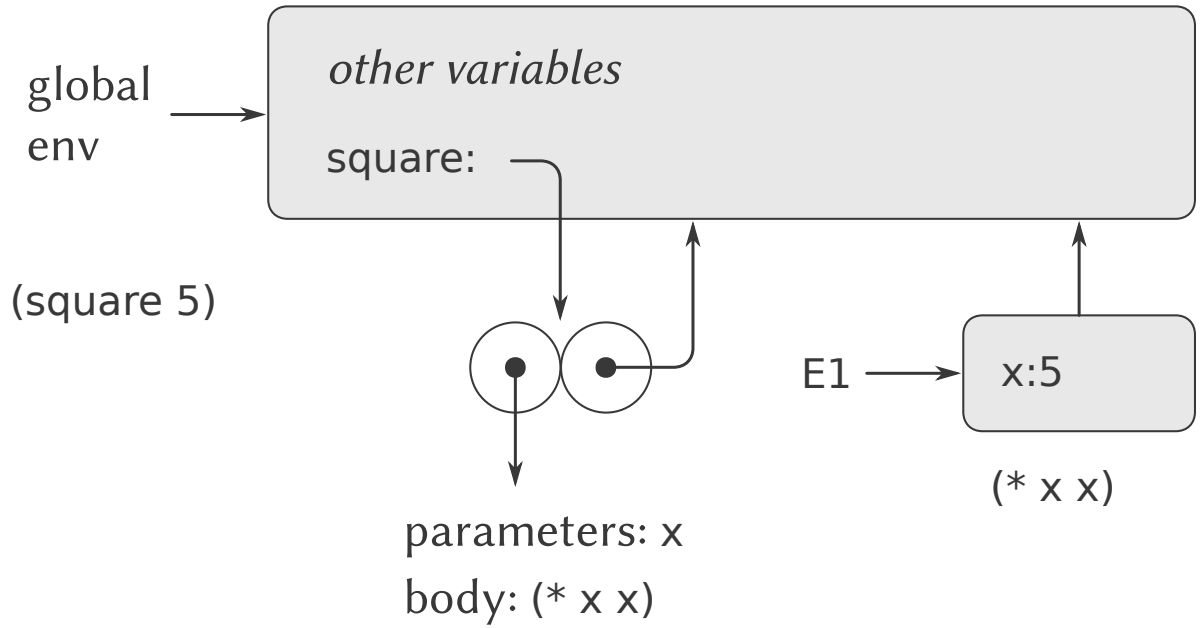
- The compound procedure square is a pair whose code specifies that the procedure has one formal parameter, x, and a body (* x x).
- The environment part of the pair points to the global environment because that is the environment in which square was defined.
- Note that square has itself been added to the global environment.

The environment model of evaluation for compound procedure application

To apply a compound procedure to arguments:

1. Create a new environment whose initial frame binds the procedure's parameters to the values of the arguments and whose enclosing environment is the environment specified by the procedure.
2. Evaluate the body of the procedure in this new environment.

Example: evaluate (square 5)

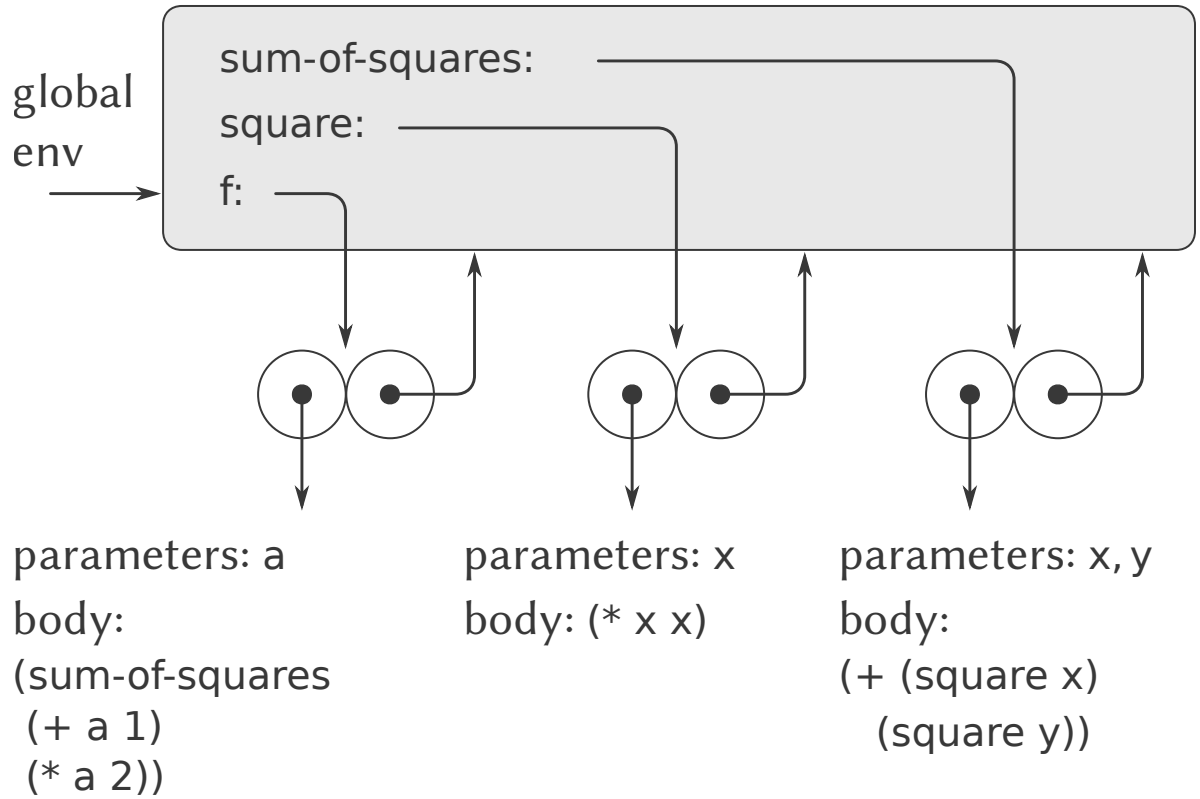


To apply a compound procedure to arguments:

1. Create a new environment whose initial frame binds the parameters to the values of the arguments and whose enclosing environment is the environment specified by the procedure.
2. Evaluate the body of the procedure in this new environment.

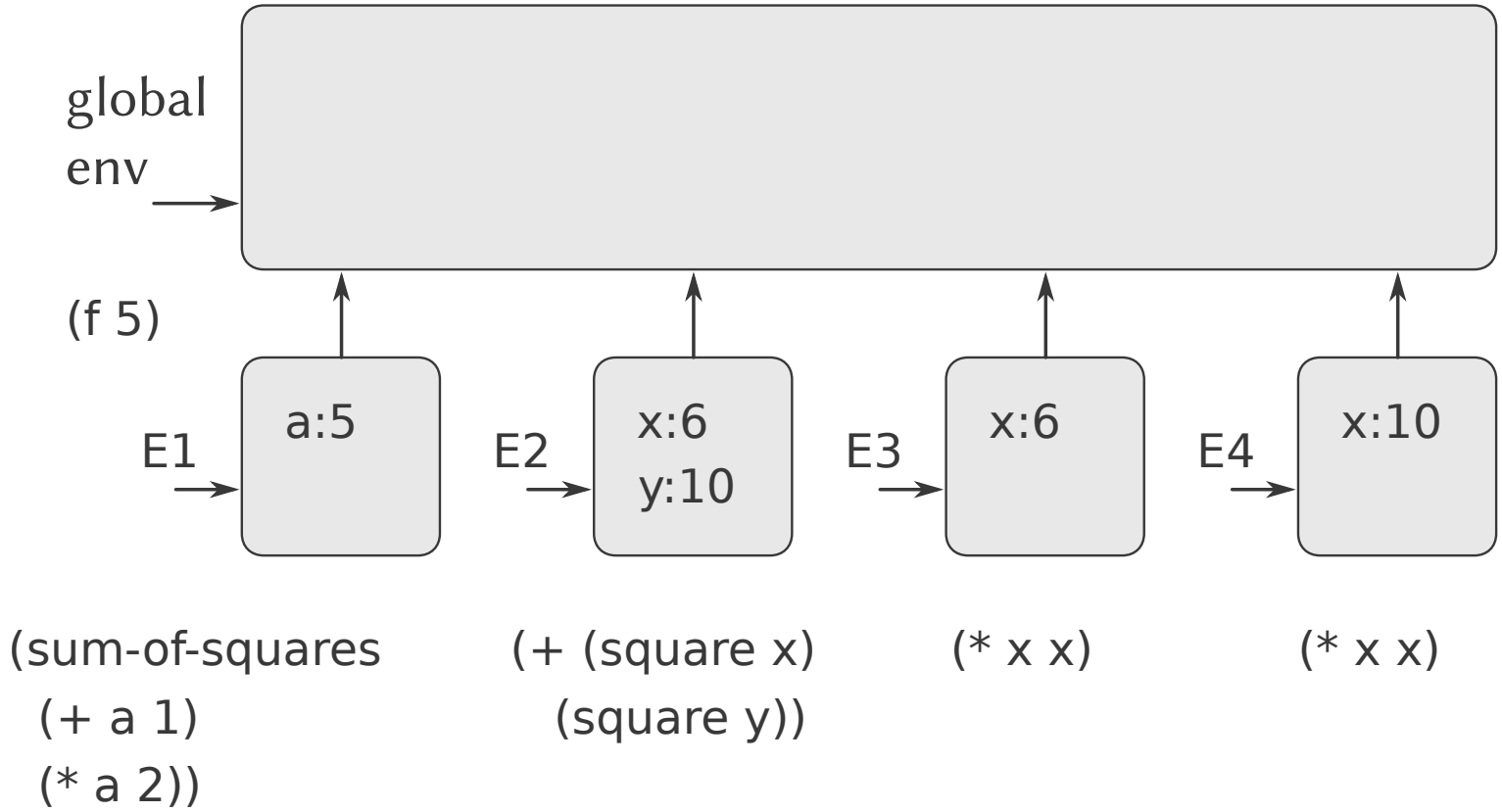
What are the similarities/differences with respect to the substitution model?

Another example




```
(define (square x)
  (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

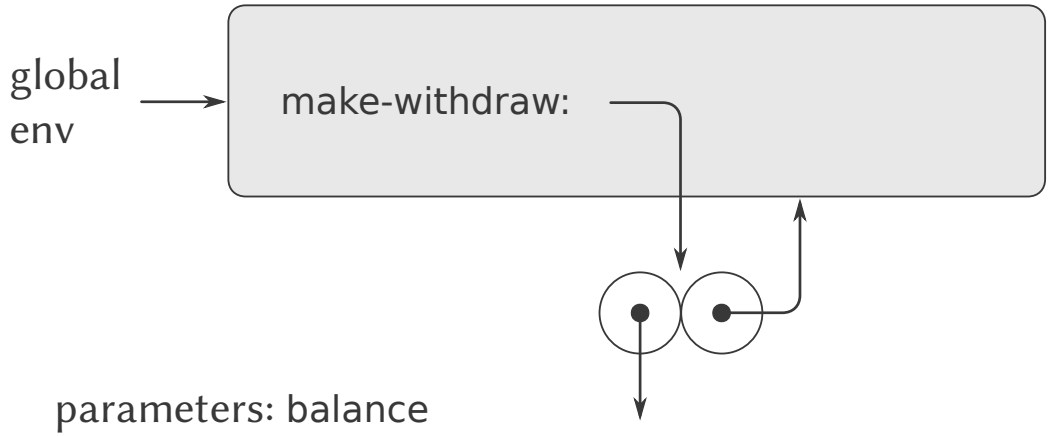
Evaluating (f 5)



What about `set!`?

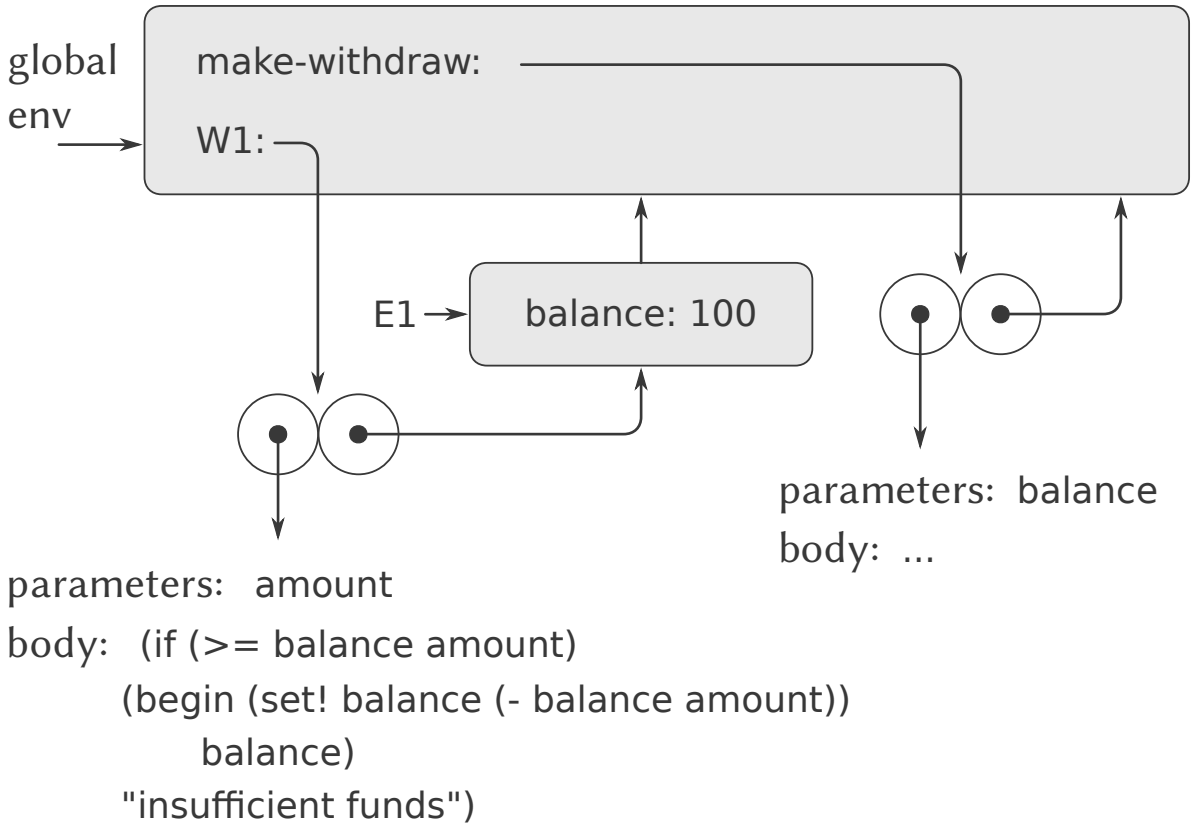
1. `set!` is a special form...it gets a special evaluation rule.
2. Evaluating (`set!` <name> <new-value>) in an environment locates the binding of the variable in the environment and changes it to associate the name with the new value.
3. If the variable is unbound, an error is signaled.

Frames with set!




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

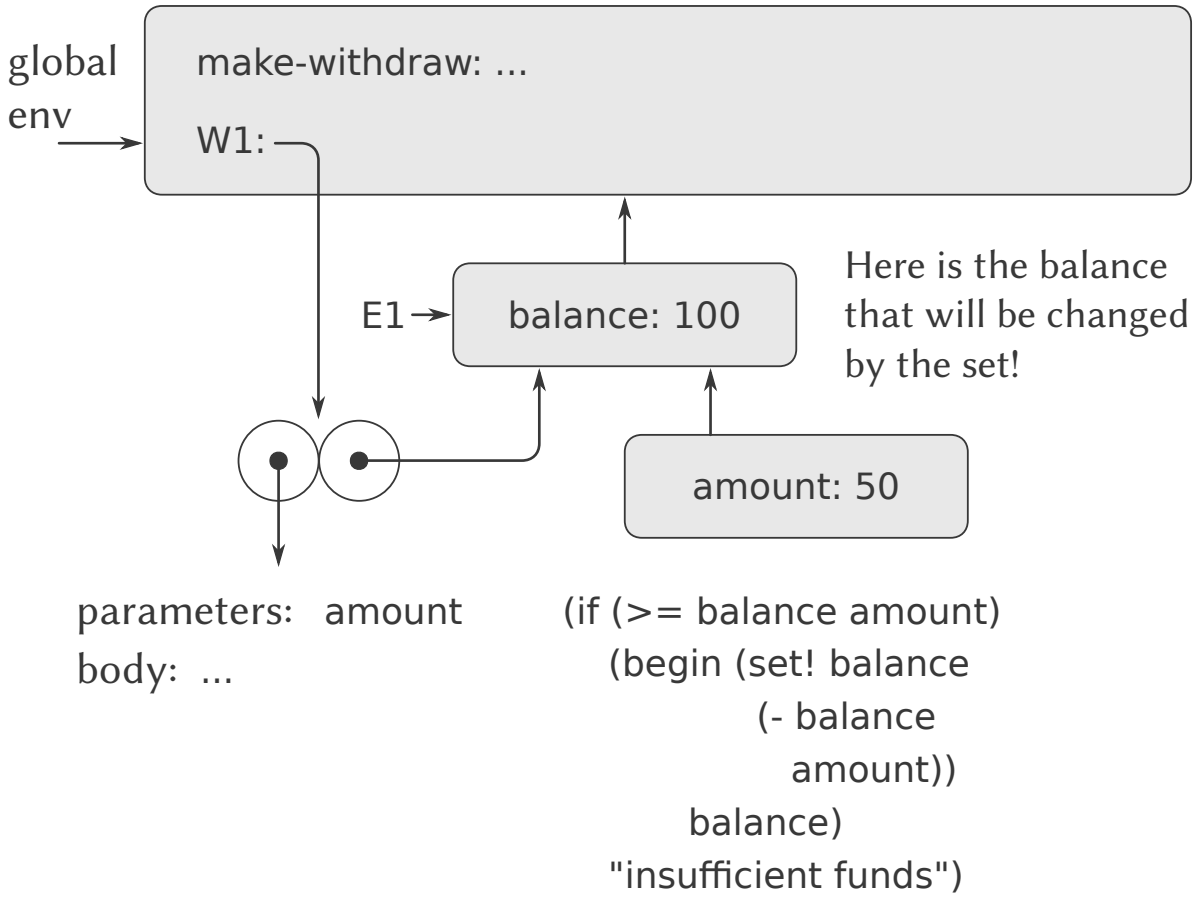
Frames with set!



Evaluate:

```
(define W1 (make-withdraw 100))
```

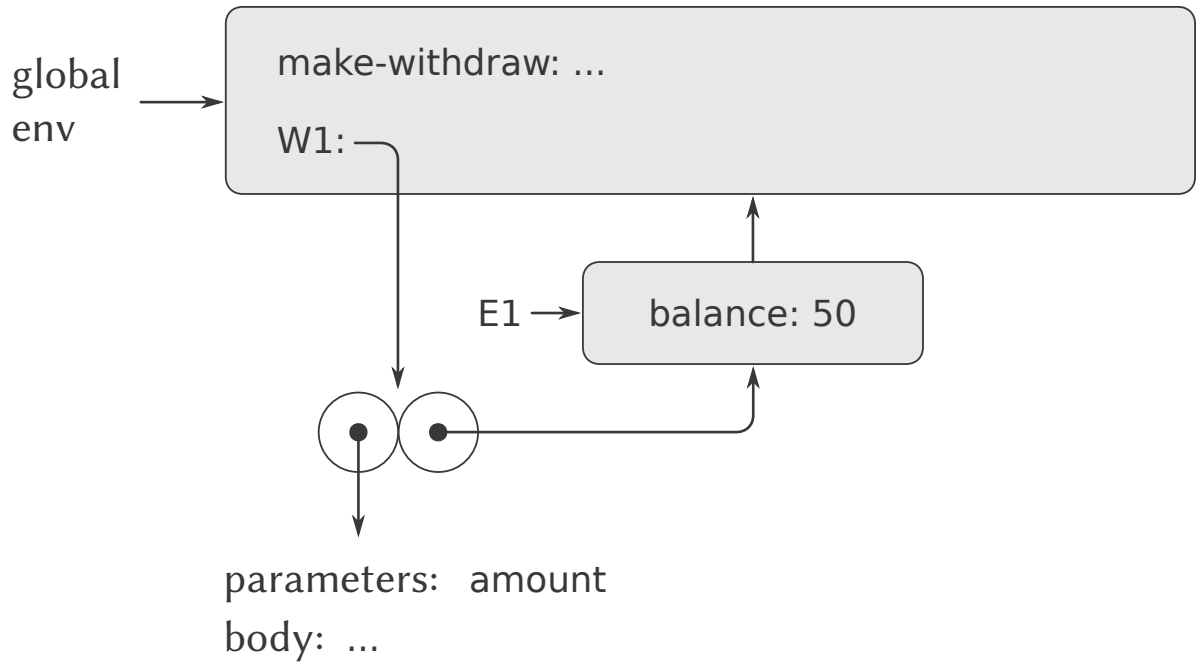
Frames with set!



Evaluate:

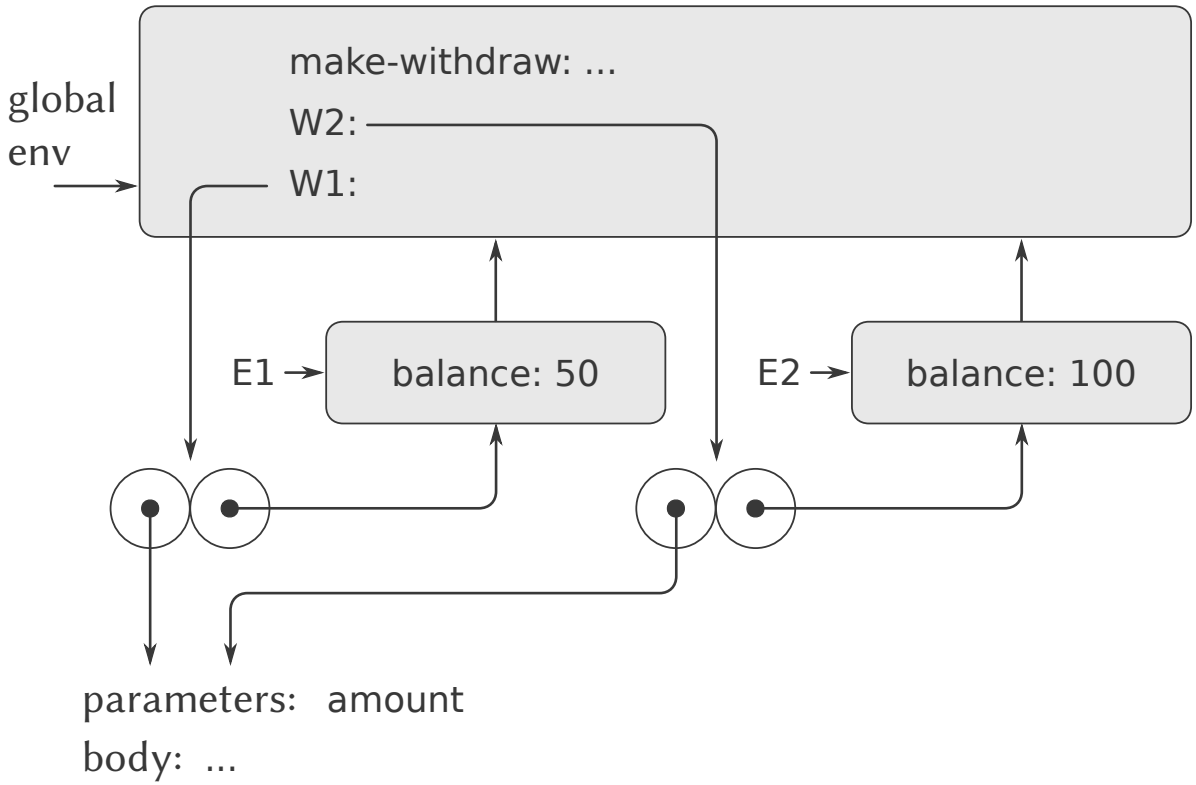
```
(W1 50)
```

Frames with set!



After the call to `W1`.

Frames with set!



Using `(define W2 (make-withdraw 100))` to create a second object.

Section 3

Implementing Streams

Streams

- Streams are sequences that are constructed partially.
- If a stream consumer tries to access part of the stream that is not yet constructed, the stream will automatically construct just enough of itself to produce the required part.
- This should remind you of normal order evaluation—only what is needed is evaluated.
- We will now see how to *implement* streams.

Implementing streams with `force` and `delay`

- To implement streams, we need a way to prevent the stream's tail from being evaluated.
- Idea: new syntax.

```
(delay <exp>)
(force <exp>)
```

- `delay` does not evaluate its argument, but returns a *delayed object*, which is a "promise" to evaluate <exp> at some point in the future.
- `force` takes a delayed object and evaluates it—it forces the delay to fulfill its promise.
- Question: does `delay` need to be a special form?

Implementing streams

- Question: how can we implement `stream-cons`?
- Would this work?

```
(define (stream-cons x s) (cons x (delay s)))
```

- `(stream-cons <a> <b>)` $\equiv$ `(cons <a> (delay <b>))`
- The idea is to **rewrite** `(stream-cons <a> <b>)` into an expression we *already* know how to evaluate.
- What about `stream-first` and `stream-rest`?

```
(define (stream-first s) (car s))
(define (stream-rest s) (force (cdr s)))
```

Streams: enumerating a large range

```
(stream-first
 (stream-rest
  (stream-enumerate 10000 1000000))))
```

- `(stream-enumerate 10000 1000000)` $\Rightarrow$
  `(cons 10000 (delay (stream-enumerate 10001 1000000)))`
- `stream-rest` forces
  `(delay (stream-enumerate 10001 1000000))` $\Rightarrow$
  `(cons 10001 (delay (stream-enumerate 10002 1000000)))`
- `stream-first` extracts 10001.
- The computation `(stream-enumerate 10002 1000000)` is never performed since it is not needed.

Implementing `force` and `delay`

- We already decided that `delay` needs to be a special form. How can we represent `(delay <exp>)` in terms of Scheme constructs we've already seen?
- `(delay <exp>)` is syntactic sugar for `(lambda () <exp>)`.
- Once again, the solution is to **rewrite** `(delay <exp>)` into an expression we already know how to evaluate.
- `force` can simply call the procedure produced by `delay` (it doesn't need to be a special form):

```
(define (force delayed-object)
  (delayed-object))
```

- Do you see an efficiency issue with `delay`?
- How could we use state to solve this problem?

Implementing delay efficiently

```
(define (memo-proc proc)
  (let ((already-run? false)
        (result null))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

Now we can define (delay <exp>) as syntactic sugar for

```
(memo-proc (lambda () <exp>))
```

Summary: streams

- Streams are delayed lists.
- The stream abstraction is built on lower-level abstractions `delay` and `force`.
- These abstractions allow us to only perform as much computation as is necessary to get our desired result.
- Streams recover efficiency while allowing us to compose abstractions like `map`, `filter`, and `reduce`.
- Question: which of our implementations of `delay` implements call-by-name? Call-by-need? Hint: see Lecture 2.
- You will implement (call-by-need) `force` and `delay` as well as streams for **Homework** 2.