Extending REC with References

CS496

A New Language

- We are going to extend REC with references
- As a consequence we obtain a functional language that supports variables as in imperative programming
- Note that all previous programming features shall still be available, including
  - let expressions
  - procedures
  - recursion

Implicit vs Explicit References

There are essentially two ways of doing this:

1. Treat every variable as a mutable reference
   - The resulting language is called IMPLICIT-REFS
   - References are implicit
2. Add mutable references to the non-mutable ones
   - The resulting language is called EXPLICIT-REFS
   - References are explicit

We are going to study both

Implicit vs Explicit References

IMPLICIT-REFS

```
let g = let count = 0
        in proc(d)
            begin set count = -(count,-1);
                  count
            end
in -((g 11), (g 22))
```

EXPLICIT-REFS

```
let g = let counter = newref(0)
        in proc (d)
            begin
               setref(counter, -(deref(counter),-1));
               deref(counter)
            end
in -((g 11),(g 22))
```

Concrete Syntax

Two new productions that are added to those of REC

$$
\begin{array}{lcl}
\langle \textit{Expression} \rangle & ::= & \text{set } \langle \textit{Identifier} \rangle = \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{begin } \langle \textit{Expression} \rangle^{+(;)} \text{ end}
\end{array}
$$

- The `set` expression is assignment
- A `begin ... end` expression evaluates its subexpressions in order and returns the value of the last one.

Concrete Syntax – An Example

```
let g = let count = 0
        in proc(d)
            begin set count = -(count,-1);
                  count
            end
in -((g 11), (g 22))
```

Another Example

```
let x = 0
in letrec even(dummy)
            = if zero?(x)
              then 1
              else begin
                    set x = -(x,1);
                    (odd 888)
                   end
          odd(dummy)
            = if zero?(x)
              then 0
              else begin
                    set x = -(x,1);
                    (even 888)
                   end
   in begin set x = 13;
            (odd -888)
      end
```

Example (cont.)

- This program uses global state (the variable x) to communicate between even and odd
  - This is not recommended practice, it only serves the purpose of illustrating the use of references
- This program uses multideclaration letrec (exercise 3.32) and a begin expression (exercise 4.4).
- The dummy argument is necessary only because all our procedures have exactly one argument, but exercise 3.21 can fix that.

IMPLICIT-REFS: Abstract Syntax

```
(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var symbol?))
  (let-exp
    (var symbol?)
    (exp1 expression?)
    (body expression?))
```

The new variants for references

```
(proc-exp
  (var symbol?)
  (body expression?))
(call-exp
  (rator expression?)
  (rand expression?))
(letrec-exp
  (p-name symbol?)
  (b-var symbol?)
  (p-body expression?)
  (letrec-body expression?))
(begin-exp
  (exp (listof expression?)))
(assign-exp
  (id symbol?)
  (exp expression?)))
```

The new variants for references

```
(proc-exp
  (var symbol?)
  (body expression?))
(call-exp
  (rator expression?)
  (rand expression?))
(letrec-exp
  (p-name symbol?)
  (b-var symbol?)
  (p-body expression?)
  (letrec-body expression?))
(begin-exp
  (exp (listof expression?)))
(assign-exp
  (id symbol?)
  (exp expression?)))
```

Example - Abstract Syntax

```
let g = let count = 0
        in proc(d)
            begin set count = -(count,-1);
                  count
            end
in -((g 11), (g 22))
```

```
(a-program
 (let-exp
  'g
  (let-exp
   'count
   (const-exp 0)
   (proc-exp 'd (begin-exp (assign-exp 'count
    (diff-exp (var-exp 'count) (const-exp -1))) (list
    (var-exp 'count)))))
  (diff-exp (call-exp (var-exp 'g) (const-exp 11))
   (call-exp (var-exp 'g) (const-exp 22))))))
```

Motivating the Store

`set!` x (+ x 1)

- As we know, variables now have two readings:
  - An address (the blue occurrence of `x`)
  - A value (the red occurence of `x`)
- Environments therefore have to change their type:
  - Before: $\mathit{Vars} \longrightarrow \mathit{ExpVal}$
  - Now: $\mathit{Vars} \longrightarrow \mathit{Refs}$
- *Refs* is a set of references or locations
- References point to expressed values
  - Hence we typically write $\mathit{Refs}(\mathit{ExpVal})$ (rather than just *Refs*)

Motivating the Store

$$\texttt{set!}\ \mathsf{x}\ (+\ \mathsf{x}\ 1)$$

- Environments therefore have to change their type:
  - Before: $\mathit{Vars} \longrightarrow \mathit{ExpVal}$
  - Now: $\mathit{Vars} \longrightarrow \mathit{Refs}(\mathit{ExpVal})$
- Revisiting the two readings of a variable
  - Blue `x`: we just look it up in the environment
  - Red `x`: we look it up in the environment and then use that location to get its value in a store or memory
- Our interpreter thus shall need a store

Environment and Store

Environment

0	1	2
`x`	`y`	`z`

Store

`(num-val 3)`	`(bool-val #t)`	`(num-val 7)`
0	1	2

Summing Up Our Analysis

$$\begin{array}{rcl} \mathsf{ExpVal} & = & \mathsf{Int} + \mathsf{Bool} + \mathsf{Proc} \\ \mathsf{DenVal} & = & \mathsf{Ref}(\mathsf{ExpVal}) \end{array}$$

- Recall: Denoted values = Values assigned to variables in the environment
- References exist only as the bindings of variables.
- Two options for the type of the interpreter
  1. Store is passed as argument
     ```
     value-of::{ expression, environment, store } -> expval
     ```
  2. Store is held in global variable (we choose this one)
     ```
     value-of::{ expression, environment } -> expval
     ```

Implementing the Store

- There are many ways to represent the store (eg. Lists, Vectors, User-defined datatypes, etc.)
  - We choose lists since it is the simplest
  - The references will just be numbers (indices of the list)
  - An example

`(num-val 3)`	`(bool-val #t)`	`(num-val 7)`	`(num-val 28)`
0	1	2	3

- Sample operations we should support
  - Create a store
  - Lookup the value associated to a reference
  - Ask the store for a fresh reference and assign it a value (allocation)

Implementing the Store

- As mentioned, the store will be held in a global variable
- We name that variable `the-store`

```scheme
;; the-store: a Scheme variable containing the
;;  current state of the store.
(define the-store 'uninitialized)
```

- Initially set to a dummy value (`'uninitialized`)
- This definition and the code that follows is in the file `store.scm`

Implementing the Store

```scheme
;; empty-store : () -> Sto
;; Page: 111
(define empty-store
  (lambda () '()))

;; initialize-store! : () -> Sto
;; usage: (initialize-store!) sets the-store to the
    empty-store
;; Page 111
(define initialize-store!
  (lambda ()
    (set! the-store (empty-store))))
```

Implementing the Store

- Allocation of a new association (page 111)

```
;; newref : ExpVal -> Ref
(define newref
  (lambda (val)
    (let ((next-ref (length the-store)))
      (set! the-store
            (append the-store (list val)))
      (when (instrument-newref)
          (eopl:printf "newref: allocating location
    ~s with initial contents ~s~%" next-ref val))
      next-ref)))
```

Implementing the Store

(num-val 3)	(bool-val #t)	(num-val 7)
0	1	2

After `newref (num-val 28)`:

(num-val 3)	(bool-val #t)	(num-val 7)	(num-val 28)
0	1	2	3

Implementing the Store

- Accessing the store (page 111)

```
;; deref :: Ref -> ExpVal
(define deref
  (lambda (ref)
    (list-ref the-store ref)))
```

- `(list-ref aList anIndex)` simply accesses the `anIndex`-th element in the list `aList`

Implementing the Store

- Updating value of existing reference (page 112)

```scheme
;; setref! :: {Ref, ExpVal} -> Unit
(define setref!
 (lambda (ref val)
  (set! the-store
    (letrec
      ((setref-inner
          ;; returns a list like store1, except that position ref1 contains val.
          (lambda (store1 ref1)
            (cond
              ((null? store1) (report-invalid-reference ref the-store))
              ((zero? ref1) (cons val (cdr store1)))
              (else
                (cons (car store1)
                      (setref-inner
                         (cdr store1) (- ref1 1))))))))
      (setref-inner the-store ref)))))
```

- The function `report-invalid-reference` is used for error

Implementing the Store

(num-val 3)	(bool-val #t)	(num-val 7)	(num-val 28)
0	1	2	3

After `setref! 1 (num-val 42)`:

(num-val 3)	(num-val 42)	(num-val 7)	(num-val 28)
0	1	2	3

Assessment

- This representation is extremely inefficient
- Ordinary memory operations require approximately constant time, but in our representation these operations require time proportional to the size of the store.
- No real implementation would ever do this, of course, but it suffices for our purposes.

Specification of Behavior of the Interpreter

- We now specify how the interpreter `value-of` behaves
- Input:
  - expression
  - environment
  - store
- Output:
  - expressed value
  - updated store

```
value-of::{ expression, environment, store } -> expval * store
```

- Note: As mentioned, in our implementation the store won't be passed as a parameter, it will be held in a global variable

Specification of Behavior of Interpreter on Constants

- First we revisit the interpreter's behavior for the simplest cases.

$$\texttt{(value-of (const-exp n)}\ \rho\ \sigma\texttt{) = (num-val n, }\sigma\texttt{)}$$

- (`var-exp` *var*) is the expression to evaluate
- ρ is an environment
- σ is the store
- The specification of `value-of` denotes a pair (result, updatedStore)
- In this case the store σ is returned unaltered

Specification of Behavior of Interpreter on Difference

```
(value-of (diff-exp exp1 exp2)  ρ σ0) =
    let (val1,σ1) = (value-of exp1 ρ σ0)
        (val2,σ2) = (value-of exp2 ρ σ1)
    in (num-val ((expval->num val1) - (expval->num
     val2)), σ2)
```

Specification of Behavior of Interpreter on Variables

1. look up the identifier in the environment to find the location to which it is bound
2. look up in the store to find the value at that location

$$\texttt{(value-of (var-exp } \mathit{var}\texttt{)}\ \rho\ \sigma\texttt{)} = \texttt{(}\sigma(\rho(\mathit{var}))\texttt{,}\ \sigma\texttt{)}$$

Specification of Behavior of Interpreter w.r.t. Assignment

```
(value-of (assign-exp var exp1) ρ σ0) =
  let (val1,σ1) = (value-of exp1 ρ σ0)
  in (num-val 27, [ρ(var) = val1]σ1)
```

- 27 is a dummy value; an assignment is evaluated to cause an effect (the value is irrelevant)
- $[\rho(var) = val1]\sigma_1$ stands for "update the reference $\rho(var)$ in the store σ_1 with the new value `val1`

Implementation

```
(value-of (var-exp var) ρ σ) = (σ(ρ(var)),σ)
```

```
(define value-of
  (lambda (exp env)
    (cases expression exp
           (var-exp (var) (deref (apply-env env var)))
           ;; more cases to follow
)))
```

Implementation

```
(value-of (assign-exp var exp1) ρ σ0) =
  let (val1,σ1) = (value-of exp1 ρ σ0)
  in (num-val 27, [ρ(var) = val1]σ1)
```

```
(define value-of
  (lambda (exp env)
    (cases expression exp
        (var-exp (var) ...)
        (assign-exp (var exp1)
          (begin
            (setref!
              (apply-env env var)
              (value-of exp1 env))
            (num-val 27)))
)))
```

Updating the Implementation of Extant REC Features

- The implementation of features that were already present in REC also have to be updated
- We've already seen the cases for constants and difference
- Here is the specification of the behavior of the interpreter for `let`

```
(value-of (let-exp var exp1 body) ρ σ0) =
  let (val1,σ1) = (value-of exp1 ρ σ0)
  in (value-of body [var = l]ρ [l = val1]σ1)
```

- l denotes a fresh store location

Updating the Implementation of Extant REC Features

- The implementation for `let`

```
(define value-of
  (lambda (exp env)
    (cases expression exp
        (var-exp (var) ...)
        (assign-exp (var exp1) ...)
        (let-exp (var exp1 body)
            (let ((val1 (value-of exp1 env)))
              (value-of body
                (extend-env var (newref val1) env))))
)))
```

Specification of Behavior of Interpreter w.r.t. Procedures

```
(value-of (proc-exp var body) ρ σ) =
    ((proc-val (closure var body ρ)), σ)
```

- No changes w.r.t. REC here

Specification of Behavior of Interpreter w.r.t. Procedure Calls

```
(value-of (call-exp rator rand) ρ σ0)
  = let (proc,σ1) =  (value-of rator ρ σ0)
        (arg,σ2)  =  (value-of rand ρ σ1)))
     ((apply-procedure (expval->proc proc) arg), σ2)
```

- No changes w.r.t. REC here but must update `apply-procedure`

Specifying the Behavior of Procedure Application

```
(apply-procedure (procedure var body ρ) val σ) =
   (value-of body [var = l]ρ [l = val]σ)
```

- Here l denotes a fresh store location

Implementing Procedure Application

```
;;  Proc * ExpVal --> ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref val) saved-env))))))
```

Environment Lookup

- Last we have to update environment lookup
- This operation was called `apply-env`
- Only the case for `letrec` has to be updated
- Before doing so we recall its definition

apply-env as Implemented in REC

```
(define apply-env
  (lambda (env search-var)
    (cases environment env
      (empty-env ()
        (report-no-binding-found search-var))
      (extend-env (saved-var saved-val saved-env)
        (if (eqv? saved-var search-var)
          saved-val
          (apply-env saved-env search-var)))
      (extend-env-rec (p-name
                       b-var
                       p-body
                       saved-env)
        (if (eqv? search-var p-name)
          (proc-val (procedure b-var p-body env))
          (apply-env saved-env search-var))))))
```

apply-env as Implemented in REC

```
(define apply-env
  (lambda (env search-var)
    (cases environment env
      (empty-env ()
        (report-no-binding-found search-var))
      (extend-env (saved-var saved-val saved-env)
        (if (eqv? saved-var search-var)
          saved-val
          (apply-env saved-env search-var)))
      (extend-env-rec (p-name
                       b-var
                       p-body
                       saved-env)
        (if (eqv? search-var p-name)
          (proc-val (procedure b-var p-body env))
          (apply-env saved-env search-var))))))
```

Updating `apply-env`

- Before:
  - `apply-env` would return a closure
- Now:
  - `apply-env` should return a reference to a location in the store containing the appropriate closure
- The reason: how the interpreter manages variable lookup

```
(value-of (var-exp var) ρ σ) = (σ(ρ(var)),σ)
```

$\rho(\mathit{var})$ is environment lookup of var in ρ

Updating `apply-env`

```
(extend-env-rec (p-names b-vars p-bodies saved-env)
  (let ((n (location search-var p-names)))
    (if n
      (newref
        (proc-val
          (procedure
            (list-ref b-vars n)
            (list-ref p-bodies n)
            env)))
      (apply-env saved-env search-var))))
```

- Manages multideclaration letrec.
- `location` takes a variable and a list of variables and returns either the position of the variable in the list, or #f if it is not present.
- This completes the implementation of IMPLICIT-REFS.

The Interpreter for IMPLICIT-REFS

- Code available from `http://www.eopl3.com`
- Directory `chapter4/implicit-refs`
- Open `top.scm` in Racket
- There are a number of tests in `tests.scm`
- You can run them with `run-one`. Eg.

```
(run-one 'gensym-test)
(num-val -1)
```

Implicit References

Concrete and Abstract Syntax

The Store

The Interpreter

Explicit References

Concrete and Abstract Syntax

EXPLICIT-REFS: A language with explicit references

1. We now define a new language EXPLICIT-REFS, which adds references as expressed values to our language.
2. Concrete and Abstract Syntax
3. Specification
4. Implementation

Implicit vs Explicit References

IMPLICIT-REFS

```
let g = let count = 0
        in proc(d)
            begin set count = -(count,-1);
                  count
            end
in -((g 11), (g 22))
```

EXPLICIT-REFS

```
let g = let counter = newref(0)
        in proc (d)
            begin
               setref(counter, -(deref(counter),-1));
               deref(counter)
            end
in -((g 11),(g 22))
```

Implicit vs Explicit Store

- In the implicit store design, every variable is mutable.
  - Allocation, dereferencing and mutation are built into the language.
- The explicit reference design gives a clear account of allocation, dereferencing, and mutation
  - All these operations are explicit in the programmer's code.

Expressed and Denoted Values

Before (IMPLICIT-REFS)

$$\begin{array}{lcl} \text{ExpVal} & = & \text{Int} + \text{Bool} + \text{Proc} \\ \text{DenVal} & = & \text{Ref(ExpVal)} \end{array}$$

Now (EXPLICIT-REFS):

$$\begin{array}{lcl} \text{ExpVal} & = & \text{Int} + \text{Bool} + \text{Proc} + \text{Ref(ExpVal)} \\ \text{DenVal} & = & \text{ExpVal} \end{array}$$

- Before: programs could only produce numbers, booleans or closures as a result
- Now: programs can also produce references as a result (or even store them)

Expressed Values for EXPLICIT-REFS

- Expressed values before

```
(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?))
  (proc-val
    (proc proc?)))
```

- Now

```
(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?))
  (proc-val
    (proc proc?))
  (ref-val
    (ref integer?)))
```

Environment and Store

Environment

(num-val 3)	(num-val 3)	(ref-val 0)
x	y	z

Store

(num-val 7)	(ref-val 9)	(bool-val #t)
0	1	2

Yet Another Example – References to References

```
let x = newref(newref(0))
in begin
     setref(deref(x), 11);
     deref(deref(x))
    end
```

- Allocates a new reference containing 0.
- Then binds x to a reference containing the above reference.
- The value of `deref(x)` is a reference to the first reference.
- So when it evaluates the setref, it is the first reference that is modified, and the entire program returns 11.

EXPLICIT-REFS: Concrete Syntax

$$
\begin{array}{lcl}
\langle \mathit{Expression} \rangle & ::= & \langle \mathit{Number} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \text{-}(\langle \mathit{Expression} \rangle \text{ , } \langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \text{zero? } (\langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \text{if } \langle \mathit{Expression} \rangle \\
 & & \text{then } \langle \mathit{Expression} \rangle \text{ else } \langle \mathit{Expression} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \langle \mathit{Identifier} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \text{let } \langle \mathit{Identifier} \rangle = \langle \mathit{Expression} \rangle \text{ in } \langle \mathit{Expression} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \text{proc } (\langle \mathit{Identifier} \rangle) \ \langle \mathit{Expression} \rangle \\
\langle \mathit{Expression} \rangle & ::= & (\langle \mathit{Expression} \rangle \ \langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \text{letrec } \langle \mathit{Identifier} \rangle(\langle \mathit{Identifier} \rangle) = \langle \mathit{Expression} \rangle \\
 & & \text{in } \langle \mathit{Expression} \rangle
\end{array}
$$

EXPLICIT-REFS: Concrete Syntax

$$\begin{array}{lcl}
\langle \mathit{Expression} \rangle & ::= & \texttt{newref}\ (\langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \texttt{deref}\ (\langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \texttt{setref}\ (\langle \mathit{Expression} \rangle)
\end{array}$$

We add three new operations to create and use references.

- `newref`: allocates a new location and returns a reference to it.
- `deref`: dereferences a reference: that is, it returns the contents of the location that the reference represents.
- `setref`: changes the contents of the location that the reference represents.

EXPLICIT-REFS: Abstract Syntax

```
(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var identifier?))
  (let-exp
    (var identifier?)
    (exp1 expression?)
    (body expression?))
```

The new variants for references

```
(proc-exp
  (var identifier?)
  (body expression?))
(call-exp
  (rator expression?)
  (rand expression?))
(letrec-exp
  (p-name identifier?)
  (b-var identifier?)
  (p-body expression?)
  (letrec-body expression?))
(newref-exp
  (exp1 expression?))
(deref-exp
  (exp1 expression?))
(setref-exp
  (ref  expression?)
  (exp  expression?))
```

The new variants for references

```
(proc-exp
  (var identifier?)
  (body expression?))
(call-exp
  (rator expression?)
  (rand expression?))
(letrec-exp
  (p-name identifier?)
  (b-var identifier?)
  (p-body expression?)
  (letrec-body expression?))
(newref-exp
  (exp1 expression?))
(deref-exp
  (exp1 expression?))
(setref-exp
  (ref  expression?)
  (exp  expression?))
```

Example

```
let x = newref(newref(0))
in begin
     setref(deref(x), 11);
     deref(deref(x))
    end
```

```
(a-program
 (let-exp
  'x
  (newref-exp (newref-exp (const-exp 0)))
  (begin-exp
     (setref-exp (deref-exp (var-exp 'x)) (const-exp
    11))
     (list (deref-exp (deref-exp (var-exp 'x))))))))
```

Specification – `newref`

```
(value-of (newref-exp exp) ρ σ0) =
   let (val,σ1) = (value-of exp ρ σ0)
   in (l, [l=val]σ1)
```

- l is fresh, that is $l \notin dom(\sigma_1)$
- `ref-val` is the tag that indicates that the result is a reference

Implementation – `newref`

```
(value-of (newref-exp exp) ρ σ0) =
   let  (val,σ1) = (value-of exp ρ σ0)
   in (l, [l=val]σ1)
```

```
(define value-of
  (lambda (exp env)
    (cases expression exp
          (var-exp (var) ...)
          (newref-exp (exp1)
            (let ((v1 (value-of exp1 env)))
              (ref-val (newref v1))))
)))
```

Specification – deref and setref

```
(value-of (deref-exp exp) ρ σ0) =
    let  (l,σ1) = (value-of exp ρ σ0)
    in (σ1(l),σ1)
```

```
(value-of (setref-exp exp1 exp2) ρ σ0) =
    let  (l,σ1)   = (value-of exp1 ρ σ0)
         (val,σ2) = (value-of exp2 ρ σ1)
    in (23,[l=val]σ2)
```

Because in our language every expression returns a value, setref returns a dummy value: 23.

Implementation – deref

```
(value-of (deref-exp exp) ρ σ0) =
    let  (l,σ1) = (value-of exp ρ σ0)
    in (σ1(l), σ1)
```

```
(define value-of
  (lambda (exp env)
    (cases expression exp
          (var-exp (var) ...)
          (newref-exp (exp1) ...)
          (deref-exp (exp1)
            (let ((v1 (value-of exp1 env)))
              (let ((ref1 (expval->ref v1)))
                (deref ref1))))
)))
```

Implementation – `setref`

```
(value-of (setref-exp exp1 exp2) ρ σ0) =
    let  (l,σ1)   = (value-of exp1 ρ σ0)
         (val,σ2) = (value-of exp2 ρ σ1)
    in (23,[l=val]σ2)
```

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) ...)
      (newref-exp (exp1) ...)
      (deref-exp (exp1) ...)
      (setref-exp (exp1 exp2)
        (let ((ref (expval->ref (value-of exp1 env))))
          (let ((val2 (value-of exp2 env)))
            (begin
              (setref! ref val2)
              (num-val 23)))))
)))
```

The Interpreter for EXPLICIT-REFS

- Code available from `http://www.eopl3.com`
- Directory `chapter4/explicit-refs`
- Open `top.scm` in Racket
- There are a number of tests in `tests.scm`
- You can run them with `run-one`. Eg.

```
(run-one 'gensym-test-1)
(num-val -1)
```