CSG111 February 20–21, 2007
Dr. Wand Readings: EOPL3, Chapter 4

Lecture 5: The Store

Key Concepts:

- Effects
- The store
  - locations, values, references
  - L-values, R-values
  - Assignment vs. binding
  - the state of the store
- Communication through the store
- Hidden state
- Explicit-Reference Design
  - references as values
- Store-Passing Specification
- Implementation using a global store variable
- Mutable Variables (variable assignment)
- Implicit-Reference Design
  - call by value
  - assignment to a variable
- Mutable Pairs
- Languages with Statements

5.1 Computational Effects

So far, we have only considered the *value* produced by a computation. But a computation may have *effects* as well: it may read, print, or alter the state of memory or a file system. In the real world, we are *always* interested in effects: if a computation doesn't print out its answer, it doesn't do us any good!

What's the difference between producing a value and producing an effect? An effect is *global*: it is seen by the entire computation. We could say that it affects the entire computation (pun intended).

We will be concerned primarily with a single effect: assignment to a location in memory. How does assignment differ from binding? As we have seen, binding is local, but variable assignment is potentially global. It is about the *sharing* of values between otherwise unrelated portions of the computation. Two procedures can share information if they both know about the same location in memory. A single procedure can share information with a future invocation of itself by leaving the information in a known location.

We model memory as a finite map from *locations* to a set of values called the *storable values*. For historical reasons, we call this the *store*. The storable values in a language are typically, but not always, the same as the expressed values of the language. This choice is part of the design of a language.

A data structure that represents a location is called a *reference*. A location is a place in memory where a value can be stored, and a reference is a data structure that refers to that place. The distinction between locations and references may be seen by analogy: a location is like a file and a reference is like a URL. The URL refers to the file, and the file contains some data. Similarly, a reference denotes a location, and the location contains some data.

References are sometimes called *L-values*. This name reflects the association of such data structures with variables appearing on the left-hand side of assignment statements. Analogously, expressed values, such as the values of the right-hand side expressions of assignment statements, are known as *R-values*.

We consider two designs for a language with a store. We call these designs *explicit references* and *implicit references*.

5.2 EXPLICIT-REFS: A language with explicit references

Add references as a new kind of expressed value.

$$\begin{aligned} \mathit{ExpVal} &= \mathit{Int} + \mathit{Bool} + \mathit{Proc} + \mathit{Ref}(\mathit{ExpVal}) \\ \mathit{DenVal} &= \mathit{ExpVal} \end{aligned}$$

Ref(*ExpVal*) means the set of references to locations that contain expressed values.

3 new operations:

- `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.

5.2.1 Examples

Recall: Binding is about the association of names with values; assignment is about the *sharing* of values between different procedures.

When a binding to a location is shared by multiple procedures, a change to the location by one is seen by all.

Here are two procedures, `even` and `odd`, that both refer to a shared location, which is bound to `x`.

They communicate not by passing data explicitly, but by changing the contents of the variable they share.

```
let x = newref(0)
in letrec even(dummy1)          % dummy arg because we don't have
                                % 0-argument procedures
            = if zero?(deref(x))
               then 1
               else begin2      % begin2 evaluates its subexps in
                                % order and returns value of the last one.
                     setref(x, -(deref(x),1));
                     (odd 888)
                    end
          odd(dummy1)
             = if zero?(deref(x))
                then 0
                else begin2
                      setref(x, -(deref(x),1));
                      (even 888)
                     end
   in begin2 setref(x,13); (odd -888) end
```

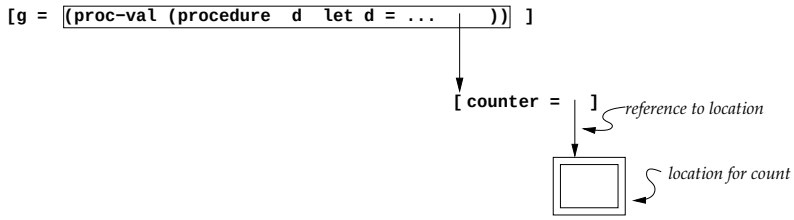
This style of communication is convenient when two procedures might share many quantities; one needs to assign only to the few quantities that change from one call to the next. Similarly, one procedure might call another procedure not directly but through a long chain of procedure calls. They could communicate data directly through a shared variable, without the intermediate procedures needing to know about it. Thus communication through a shared variable can be a kind of information hiding.

Another use of assignment is to create hidden state through the use of private variables.

```
let g = let counter = newref(0)
         in proc (dummy)
              begin2
               setref(counter, -(deref(counter),-1));
               deref(counter)
              end
 in -((g 11),(g 11))
```

Here the procedure g keeps a private variable that stores the number of times g has been called. In our language, the operands in a difference expression are evaluated left-to-right. Hence the first call to g returns 1, the second call to g returns 2, and the entire program has the value -1.

Here is a picture of the environment in which g is bound.



We can think of this as the different invocations of g sharing information with each other. In Scheme, this technique is used by the procedure `gensym` to create unique symbols.

Exercise: What would have happened had the program been instead

```
let g = proc (dummy)
          let counter = newref(0)
          in begin2
               setref(counter, -(deref(counter),-1));
               deref(counter)
             end
in -((g 11),(g 11))
```

5.2.2 Store-Passing Specifications

In our language, any expression may have an effect. To specify these effects, we need to describe how each evaluation can modify the store, and what store should be used for each evaluation.

In our specifications, we use σ to range over stores. We write $[l = v]\sigma$ to mean a store just like σ, except that location l is mapped to v. When we refer to a particular value of σ, we sometimes call it the *state* of the store.

We use *store-passing specfications*. In a store-passing specification, the store is passed as an explicit argument to `value-of` and is returned as an explicit result from `value-of`.

So instead of saying

$$(\texttt{value-of}\ \ exp\ \ \rho) = v$$

we write

$$(\texttt{value-of}\ \ exp\ \ \rho\ \ \sigma_0) = (v_1, \sigma_1)$$

This asserts that expression exp, evaluated in environment ρ and with the store in state σ_0, returns the value v_1 and leaves the store in a possibly-different state σ_1.

Thus we can specify an effect-free operation like `const-exp` by writing

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

showing that the store is unchanged by evaluation of this expression.

$$\frac{\begin{array}{c}(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (v_1, \sigma_1) \\ (\texttt{value-of}\ exp_2\ \rho\ \sigma_1) = (v_2, \sigma_2)\end{array}}{(\texttt{value-of}\ (\texttt{diff-exp}\ exp_1\ exp_2)\ \rho\ \sigma_0) = (v_1\texttt{-}v_2, \sigma_2)}$$

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (v_1, \sigma_1)}{\begin{array}{l}(\texttt{value-of}\ (\texttt{if-exp}\ exp_1\ exp_2\ exp_3)\ \rho\ \sigma_0) \\ \quad = \begin{cases}(\texttt{value-of}\ exp_2\ \rho\ \sigma_1) & \text{if } (\texttt{expval->bool}\ v_1) = \texttt{\#t} \\ (\texttt{value-of}\ exp_3\ \rho\ \sigma_1) & \text{if } (\texttt{expval->bool}\ v_1) = \texttt{\#f}\end{cases}\end{array}}$$

5.2.3 Specifying Operations on Explicit References

$$\frac{(\texttt{value-of}\ exp\ \rho\ \sigma_0) = (v, \sigma_1) \qquad l \notin \mathrm{dom}(\sigma_1)}{(\texttt{value-of}\ (\texttt{newref-exp}\ exp)\ \rho\ \sigma_0) = (l, [l{=}v]\sigma_1)}$$

$$\frac{(\texttt{value-of}\ exp\ \rho\ \sigma_0) = (l, \sigma_1)}{(\texttt{value-of}\ (\texttt{deref-exp}\ exp)\ \rho\ \sigma_0) = (\sigma_1(l), \sigma_1)}$$

$$\frac{\begin{array}{c}(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (l, \sigma_1) \\ (\texttt{value-of}\ exp_2\ \rho\ \sigma_1) = (v, \sigma_2)\end{array}}{(\texttt{value-of}\ (\texttt{setref-exp}\ exp_1\ exp_2)\ \rho\ \sigma_0) = (23, [l{=}v]\sigma_2)}$$

This rule says that a `setref-exp` evaluates its operands from left to right. The value of the first operand must be a reference to a location l. The `setref-exp` then updates the resulting store by putting the value v of the second argument in location l. What should a `setref-exp` return? It could return anything. To emphasize the arbitrary nature of this choice, we have specified that it returns 23.

5.2.4 Implementation: `lecture05/explicit-refs`

The specification language we have used so far makes it easy to describe the desired behavior of effectful computations, but it does not embody a key fact about the store: a reference ultimately refers to a real location in a memory that exists in the real world. Since we have only one real world, our program can only keep track of one state σ of the store.

In our implementations, we take advantage of this fact by modelling the store using Scheme's own store. Thus we model an effect as a Scheme effect.

We represent the state of the store as a Scheme value, but we do not explicitly pass and return it, as the specification suggests. Instead, we keep the state in a single Scheme global variable, to which all the procedures of the implementation have access. This is much like `even`/`odd` example, where we used a shared location instead of passing an explicit argument. By using a single Scheme global variable, we also use as little as possible of our understanding of Scheme effects.

We still have to choose how to model the store as a Scheme value. We choose the simplest possible model: we represent the store as a list of expressed values, and a reference is a number that denotes a position in the list. A new reference is allocated by appending a new value to the list; and updating the store is modelled by copying over as much of the list as necessary.

This representation is extremely inefficient. Ordinary memory operations require approximately constant time to execute, 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.

```
(module store (lib "eopl.ss" "eopl")

  (require "drscheme-init.scm")

  (provide initialize-store! reference? newref deref setref!)

  ;;;;;;;;;;;;;;;; references and the store ;;;;;;;;;;;;;;;;

  ;;; world's dumbest model of the store:  the store is a list and a
  ;;; reference is number which denotes a position in the list.

  empty-store : () -> store
  (define empty-store
    (lambda () '()))

  usage: A Scheme variable containing the current state of the store.
     Initially set to a dummy value.  NOT exported.
  (define the-store 'uninitialized)

  initialize-store! : () -> unspecified
  usage: (initialize-store) sets the-store to the empty store
  (define initialize-store!
    (lambda ()
      (set! the-store (empty-store))))

  reference? : scheme-value -> boolean
  (define reference? integer?)
```

newref : expval -> reference

```
(define newref
  (lambda (val)
    (let ((next-ref (length the-store)))
      (set! the-store
            (append the-store (list val)))
      next-ref)))

(define deref
  (lambda (ref) (list-ref the-store ref)))
```

usage: sets the-store to a state like the original, but with position ref0 containing val.

```
(define setref!
  (lambda (ref0 val)
    (set! the-store
          (letrec
            ((setref-inner
               usage: returns a list like store1, except that
                  position ref contains val.
                  (lambda (store1 ref)
                    (cond
                      ((null? store1)
                       (eopl:error 'setref
                         "illegal reference ~s in store ~s"
                         ref0 the-store))
                      ((zero? ref)
                       (cons val (cdr store1)))
                      (else
                        (cons
                          (car store1)
                          (setref-inner
                            (cdr store1) (- ref 1))))))))
            (setref-inner the-store ref0)))))

)
```

We add a new variant, `ref-val`, to the datatype for expressed values, and we modify `value-of-program` to initialize the store before each evaluation.

```
;;; in interp.scm
  (define value-of-program
    (lambda (pgm)
      (initialize-store!)
      (cases program pgm
        (a-program (e)
          (value-of e (init-env))))))
```

Now we can write clauses in `value-of` for `newref`, `deref`, and `setref`.

```
;; in value-of

        (newref-exp (e1)
          (let ((v1 (value-of e1 env)))
            (ref-val (newref v1))))

        (deref-exp (e1)
          (let ((v1 (value-of e1 env)))
            (let ((ref1 (expval->ref v1)))
              (deref ref1))))

        (setref-exp (e1 e2)
          (let ((ref (expval->ref (value-of e1 env))))
            (let ((v2 (value-of e2 env)))
              (begin
                (setref! ref v2)
                (num-val 23)))))
```

5.2.5 Instrumentation

```
;; in store.scm:

  (provide get-store)

  get-store : () -> store
  usage: exports the store as a list for instrumentation clients.
  (define get-store
    (lambda () the-store))

  (provide instrument-newref)

  (define instrument-newref (make-parameter #f))

;; say (instrument-newref #t) or (instrument-newref #f) to turn
;; instrumentation on or off

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

```
;; in interp.scm:

  (provide instrument-let instrument-newref)

;; say (instrument-let #t) or (instrument-let #f) to turn
;; instrumentation on or off

        (let-exp (id rhs body)
          (if (instrument-let)
            (eopl:printf "entering let ~s~%" id))
          (let ((val (value-of rhs env)))
            (let ((new-env (extend-env id val env)))
              (if (instrument-let)
                (begin
                  (eopl:printf "entering body of let ~s with env =~%" id)
                  (pretty-print (env->list new-env))
                  (eopl:printf "store =~%")
                  (pretty-print (store->list (get-store-as-list)))
                  (eopl:printf "~%")
                  ))
              (value-of body new-env))))

  (define apply-procedure
    (lambda (proc1 arg)
      (cases proc proc1
        (procedure (bvar body saved-env)
          (let ((new-env (extend-env bvar arg saved-env)))
            (if (instrument-let)
              (begin
                (eopl:printf
                  "entering body of proc ~s with env =~%"
                  bvar)
                (pretty-print (env->list new-env))
                (eopl:printf "store =~%")
                (pretty-print (store->list (get-store-as-list)))
                (eopl:printf "~%")))
            (value-of body new-env)))))))
```

5.2.6 Example

```
> (run "
let x = newref(22)
in let f = proc (z) let zz = newref(-(z,deref(x)))
                    in deref(zz)
   in -((f 66), (f 55))")

entering let x
newref: allocating location 0
entering body of let x with env =
((x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))

entering let f
entering body of let f with env =
((f
  (procedure
    z
    ...
    ((x #(struct:ref-val 0))
     (i #(struct:num-val 1))
     (v #(struct:num-val 5))
     (x #(struct:num-val 10)))))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))
```

```
entering body of proc z with env =
((z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))

entering let zz
newref: allocating location 1
entering body of let zz with env =
((zz #(struct:ref-val 1))
 (z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))

entering body of proc z with env =
((z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))
```

```
entering let zz
newref: allocating location 2
entering body of let zz with env =
((zz #(struct:ref-val 2))
 (z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22))
 (1 #(struct:num-val 44))
 (2 #(struct:num-val 33)))

#(struct:num-val 11)
```

This is `show-allocation-1` in `lecture05/explicit-refs/tests.scm`.

5.3 IMPLICIT-REFS: a language with implicit references

The explicit reference design gives a clear account of allocation, dereferencing, and mutation because all these operations are explicit in the programmer's code.

Most programming languages take common patterns of allocation, dereferencing, and mutation, and package them up as part of the language. Then the programmer need not worry about when to perform these operations, because they are built into the language.

In this design, every identifier denotes a reference. Denoted values are references to locations that contain expressed values. References are no longer expressed values. They exist only as the bindings of variables.

$$\begin{aligned} \mathit{ExpVal} &= \mathit{Int} + \mathit{Bool} + \mathit{Proc} \\ \mathit{DenVal} &= \mathit{Ref}(\mathit{ExpVal}) \end{aligned}$$

Locations are created with each binding operation: at each procedure call, `let`, or `letrec`.

When an identifier appears in an expression, we first look up the identifier in the environment to find the location to which it is bound, and then we look up in the store to find the value at that location. Hence we have a "two-level" system for `var-exp`.

Locations are changed by a `set` expression. We use the syntax

Expression ::= `set` *Identifier* `=` *Expression*

```
set-exp (var rhs)
```

Here the *Identifier* is not part of an expression, so it does not get dereferenced. In this design, we say that variables are *mutable*, meaning changeable.

This design for doing this is called *call-by-value*, or *implicit references*. Most programming languages, including Scheme, use some variation on this design.

Let's look at our two sample programs in this design:

```
let x = 0
in letrec even(dummy1)
            = if zero?(x)
               then 1
               else begin2
                     set x = -(x,1);
                     (odd 888)
                    end
          odd(dummy1)
            = if zero?(x)
               then 0
               else begin2
                     set x = -(x,1);
                     (even 888)
                    end
   in begin2 set x = 13; (odd -888) end

let g = let count = 0
        in proc (dummy)
            begin2
             set count = -(count,-1);
             count
           end
in -((g 11), (g 11))
```

5.3.1 Specification

We can write the rules for dereference and `set` easily. The environment now always binds variables to locations, so when a variable appears as an expression, we need to dereference it:

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

Assignment works as one might expect: we look up the left-hand side in the environment, getting a location, we evaluate the right-hand side in the environment, and we modify the desired location. As with `setref`, the value returned by a `set` expression is arbitrary. We choose to have it return 27. In the rule, we omit the `num-val` constructor since it should be clear from the context.

$$\frac{\texttt{(value-of } exp \ \rho \ \sigma_0\texttt{)} = (v, \sigma_1)}{\texttt{(value-of (set-exp } var \ exp\texttt{)} \ \rho \ \sigma_0\texttt{)} = (27, [\rho(var) = v]\sigma_1)}$$

We also need to rewrite the rules for procedure call and `let` to show the modified store. For procedure call, the rule becomes

$$\begin{array}{l}\texttt{(apply-procedure (procedure } var \ exp \ \rho\texttt{)} \ v \ \sigma\texttt{)} = \\ \quad \texttt{(value-of } exp \ [var = l]\rho \ [l = v]\sigma\texttt{)}\end{array}$$

where l is a location not in the domain of σ.

5.3.2 Implementation: `lecture05/implicit-refs`

We modify the interpreter for LETREC.

```
;; in value-of:

      (var-exp (var) (deref (apply-env env var)))


;; add:

      (set-exp (var rhs)
        (begin
          (setref!
            (apply-env env var)
            (value-of rhs env))
          (num-val 27)))
```

What about creating references? New locations are supposed to be allocated at every new binding. There are exactly three places in the language where new bindings are created: in a `let`, in a procedure call, and in a `letrec`.

```
;; in interp.scm:

;; in value-of:

          (let-exp (id rhs body)
            (let ((val (value-of rhs env)))
              (value-of body
                (extend-env id (newref val) env))))

;; in apply-procedure:

  (define apply-procedure
    (lambda (proc1 arg)
      (cases proc proc1
        (procedure (bvar body saved-env)
          (value-of body
            (extend-env bvar (newref arg) saved-env))))))
```

Last, to handle `letrec`, we replace the `extend-env-recursively` clause in `apply-env` to return a reference to a location containing the appropriate closure:

```
;; in apply-env

          (extend-env-recursively (p x e saved-env)
            (if (eqv? search-sym p)
              (newref
                (proc-val
                  (procedure p x e env)))
              (apply-env saved-env search-sym)))
```

```
> (run  "
let f = proc (x) proc (y)
                  begin
                   set x = -(x,-1);
                   -(x,y)
                  end
in ((f 44) 33)")
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))

newref: allocating location 4
entering body of proc x with env =
((x 4) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44)))
```

```
newref: allocating location 5
entering body of proc y with env =
((y 5) (x 4) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44))
 (5 #(struct:num-val 33)))

#(struct:num-val 12)
>
```

This is `example-for-book-1` in `interps/lecture05/implicit-refs/tests.scm`.

5.4 MUTABLE-PAIRS: a language with mutable pairs

Want to add mutable aggregates (like arrays, vectors, etc.).

For simplicity, we'll just do mutable pairs.

We'll extend IMPLICIT-REFS.

Pairs will be expressed values, and will have the following operations:

newpair	: `expval * expval -> mutpair`
left	: `mutpair -> expval`
right	: `mutpair -> expval`
setleft	: `mutpair * expval -> unspecified`
setright	: `mutpair * expval -> unspecified`

A pair consists of two locations, each of which is independently assignable. This gives us the domain equations:

$$\begin{aligned} \mathit{ExpVal} &= \mathit{Int} + \mathit{Bool} + \mathit{Proc} + \mathit{MutPair} \\ \mathit{DenVal} &= \mathit{Ref}(\mathit{ExpVal}) \\ \mathit{MutPair} &= \mathit{Ref}(\mathit{ExpVal}) \times \mathit{Ref}(\mathit{ExpVal}) \end{aligned}$$

We can implement this literally using the `reference` data type from our preceding examples.

```
(module pairval1 (lib "eopl.ss" "eopl")

  (require "store.scm")

  (provide (all-defined))

  (define-datatype mutpair mutpair?
    (a-pair
      (left-loc reference?)
      (right-loc reference?)))

  make-pair : expval * expval -> mutpair
  (define make-pair
    (lambda (val1 val2)
      (a-pair
        (newref val1)
        (newref val2))))

  left : mutpair -> expval
  (define left
    (lambda (p)
      (cases mutpair p
        (a-pair (left-loc right-loc)
          (deref left-loc)))))

  setleft : mutpair * expval -> unspecified
  (define setleft
    (lambda (p val)
      (cases mutpair p
        (a-pair (left-loc right-loc)
          (setref! left-loc val)))))

  ... and similarly for right and setright

  )
```

Once we've done this, it is straightforward to add these to the language. We add a `mutpair-val` variant to our datatype of expressed values, and five new lines to `value-of`. We arbitrarily choose to make `setleft` return 82 and `setright` return 83.

```
;; add to value-of:

        (newpair-exp (e1 e2)
          (let ((v1 (value-of e1 env))
                (v2 (value-of e2 env)))
            (mutpair-val (make-pair v1 v2))))

        (left-exp (e1)
          (let ((v1 (value-of e1 env)))
            (let ((p1 (expval->mutpair v1)))
              (left p1))))

        (setleft-exp (e1 e2)
          (let ((v1 (value-of e1 env))
                (v2 (value-of e2 env)))
            (let ((p (expval->mutpair v1)))
              (begin
                (setleft p v2)
                (num-val 82)))))

        ... similarly for right and setright
```

5.4.1 Worked Example

This is `example-for-mutable-pairs-section` in `interps/lecture05/mutable--pairs/tests.scm`.

```
> (run "let glo = pair(11,22)
in let f = proc (loc)
            let d1 = setright(loc, left(loc))
            in let d2 = setleft(glo, 99)
            in -(left(loc),right(loc))
in (f glo)")
;; allocating cells for init-env
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let glo
;; allocating cells for the pair
newref: allocating location 3
newref: allocating location 4
;; allocating cell for glo
newref: allocating location 5
entering body of let glo with env =
((glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4))))
```

```
entering let f
;; allocating cell for f
newref: allocating location 6
entering body of let f with env =
((f 6) (glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
 (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2)))))

;; allocating cell for loc
newref: allocating location 7
entering body of proc loc with env =
((loc 7) (glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
 (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2))))
 (7 #(struct:mutpair-val #(struct:a-pair 3 4))))

#(struct:num-val 88)
>
```

5.4.2 Another representation: `pairval2.scm`

The representation of a mutable pair as two references does not take advantage of all we know about *MutPair* . The two locations in a pair are independently assignable, but they are not independently allocated. We know that they will be allocated together: if the left part of a pair is one location, then the right part is in the next location. So we can instead represent the pair by a reference to its left.

```
(module pairval2 (lib "eopl.ss" "eopl")

  ...

  (define mutpair? reference?)          ; inaccurate, but good enough

  (define make-pair
    (lambda (val1 val2)
      (let ((ref1 (newref val1)))
        (let ((ref2 (newref val2)))     ; guaranteed to be ref1 + 1
          ref1))))

  (define left
    (lambda (p)
      (deref p)))

  (define right
    (lambda (p)
      (deref (+ 1 p))))

  (define setleft
    (lambda (p val)
      (setref! p val)))

  (define setright
    (lambda (p val)
      (setref! (+ 1 p) val)))

  )
```

Similarly, one could implement aggregate objects as a contiguous range of locations, and then represent that object in the language by a pointer to its first location.

This used to be the standard practice:

- In Lisp or Scheme, a `cons` pair is represented by a single pointer l, representing the two locations l and $l+1$ (or $l+4$ or something else if you're counting in bytes).
- This is also how arrays are represented in C, where a reference to `a[i]` can be written as `*(a+i)` ([1]). This leads to *pointer arithmetic*: in C, pointer objects are normalized so that "if `pa` points to a particular element of an array, then by definition `pa+1` points to the next element[2].

However, a pointer does not by itself identify an area of memory unless it is supplemented by information about the length of the area. The lack of length information is a source of classic security errors, such as out-of-bounds array writes.

Therefore, modern safe languages such as Java or Scheme, include the size of each aggregate structure (such as a Java vector or a Scheme array) as part of the representation and always check array references to make sure they are in-bounds.

[1] Kernighan and Richie, `The C Programming Language, Second Edition`, Section 5.3
[2] *ibid.*

5.5 Languages with Statements

So far our languages have been expression-oriented: the primary syntactic category of interest has been expressions and we have primarily been interested in their values. Now extend our interpreter to model a simple statement-oriented language.

As in IMPLICIT-REFS, expressed values are integers, booleans, and procedures, and the denoted values are references to locations containing expressed values.

Syntax:

program ::= *statement*
`a-program (stmt)`

statement ::= *id* `=` *expression*
`assign-statement (id exp)`

::= `print (`*expression*`)`
`print-statement (exp)`

::= `{`{*statement*}$^{*(;)}$ `}`
`compound-statement (stmts)`

::= `if` *expression* *statement* *statement*
`if-statement (exp true-stmt false-stmt)`

::= `var` {*id*}$^{*(,)}$ `;` *statement*
`block-statement (ids body)`

The non-terminal *expression* refers to the language of expressions of IMPLICIT-REFS.

A program is a statement. A program does not return a value, but works by printing.

Assignment statements work in the usual way. A print statement evaluates its actual parameter and prints the result. The `if` statement works in the usual way. A block statement binds each of the declared identifiers to an uninitialized reference and then executes the body of the block. The scope of these bindings is the body.

5.5.1 Examples

```
--> var x,y; {x = 3; y = 4; print(+(x,y))}   % Example 1
7
--> var x,y,z; {x = 3; y = 4; z = 0;         % Example 2
                while x {z = +(z,y); x = sub1(x)};
                print(z)}
12
--> var x; {x = 3; print(x);                  % Example 3
            var x; {x = 4; print(x)};
            print(x)}
3
4
3
--> var f,x; {f = proc(x,y) *(x,y);          % Example 4
              x = 3;
              print ( (f 4 x) )}
12
```

Example 3 illustrates the scoping of the block statement.

Example 4 illustrates the interaction between statements and expressions. A procedure value is created and stored in the variable `f`. In the last line, this procedure is applied to the actual parameters 4 and `x`; since `x` is bound to a reference, it is dereferenced to obtain 3. Our syntax requires the two sets of parentheses here: the outer set are from the `print-statement` production and the inner ones are from the `app` production for expressions.

5.5.2 Specifying Statements

We can still use a store-passing specification, but for statements there is no return value, so a typical formula will look like

$$\texttt{(execute-statement s } \rho \ \sigma_0\texttt{)} = \sigma_1$$

meaning that if we execute statement s in environment ρ and store σ_0, then afterwards the store will be left in state σ_1.

For assignment statements:

$$\frac{\rho(var) = l \qquad \texttt{(value-of } exp \ \rho \ \sigma_0\texttt{)} = (v_1, \sigma_1)}{\texttt{(execute-statement (assign-statement } var \ exp\texttt{)} \ \rho \ \sigma_0\texttt{)} = [l{=}v_1]\sigma_1}$$

For compound statement, we'll show just the 2-statement version:

$$\frac{\begin{array}{l}\texttt{(execute-statement } s_1 \ \rho \ \sigma_0\texttt{)} = \sigma_1 \\ \texttt{(execute-statement } s_2 \ \rho \ \sigma_1\texttt{)} = \sigma_2\end{array}}{\texttt{(execute-statement (compound-statement } s_1 \ s_2\texttt{)} \ \rho \ \sigma_0\texttt{)} = \sigma_2}$$

[Exercise: what if we wanted to model output, as well? How would the state σ change?]

5.5.3 Implementation

We'll implement this using the same implicit-store strategy we've used for all our other interpreters in this lecture. So in place of a 3-argument `execute-statement` procedure, we'll have a procedure called `execute-statement!` that will take 2 arguments, side-effect the Scheme store, and return an unspecified value. (We say it is executed *for effect only*).

```
(define execute-program!
  (lambda (pgm)
    (cases program pgm
      (initialize-store!)
      (a-program (statement)
        (execute-statement! statement (init-env))))))

(define execute-statement!
  (lambda (stmt env)
    (cases statement stmt
      (assign-statement (id exp)
        (setref!
          (apply-env env id)
          (value-of exp env)))
      (compound-statement (statements)
        (for-each      ;; guaranteed to work from left to right
          (lambda (statement) (execute-statement! statement env))
          statements))
      (print-statement (exp)
        (begin (eopl:printf "~s" (value-of exp env)) (newline)))
      (if-statement (exp true-statement false-statement)
        (if (true-value? (value-of exp env))
          (execute-statement! true-statement env)
          (execute-statement! false-statement env)))
      (block-statement (ids statement)
        (execute-statement! statement
          (extend-env ids
            (map (lambda (d) '*uninitialized*) ids)
            env)))
      )))
```