Seems fine, right? Rewrite in terms of `lambda`:

```
((lambda (loop-forever)
   (loop-forever 10))
 (lambda (x) (loop-forever x)))
```

Clearly, the `loop-forever` on the last line isn't bound!

This is another feature we get "for free" from the top-level. To eliminate this magical force, we need to understand recursion explicitly, which we will do soon [REF].

8 Mutation: Structures and Variables

It's time for another

Which of these is the same?

- `f = 3`
- `o.f = 3`
- `f = 3`

Assuming all three are in Java, the first and third could behave exactly like each other or exactly like the second: it all depends on whether `f` is a local identifier (such as a parameter) or a field of the object (i.e., the code is really `this.f = 3`).

In either case, we are asking the evaluator to permanently change the value bound to `f`. This has important implications for other observers. Until now, for a given set of inputs, a computation always returned the same value. Now, the answer depends on *when* it was invoked: above, it depends on whether it was invoked before or after the value of `f` was changed. The introduction of time has profound effects on reasoning about programs.

However, there are really two quite different notions of change buried in the uniform syntax above. Changing the value of a field (`o.f = 3` or `this.f = 3`) is extremely different from changing that of an identifier (`f = 3` where `f` is bound inside the method, not by the object). We will explore these in turn. We'll tackle fields below, and return to identifiers in section 8.2.

8.1 Mutable Structures

8.1.1 A Simple Model of Mutable Structures

Objects are a generalization of structures, as we will soon see [REF]. Therefore, fields in objects are a generalization of fields in structures and to understand mutation, it is mostly (but not entirely! [REF]) sufficient to understand mutable objects. To be even more reductionist, we don't need a structure to have many fields: a single one will suffice. We call this a *box*. In Racket, boxes support just three operations:

```
box : ('a -> (boxof 'a))
unbox : ((boxof 'a) -> 'a)
set-box! : ((boxof 'a) 'a -> void)
```

Thus, `box` takes a value and wraps it in a mutable container. `unbox` extracts the current value inside the container. Finally, `set-box!` changes the value in the container, and in a typed language, the new value is expected to be type-consistent with what was there before. You can thus think of a box as equivalent to a Java container class with parameterized type, which has a single member field with a getter and setter: `box` is the constructor, `unbox` is the getter, and `set-box!` is the setter. (Because there is only one field, its name is irrelevant.)

```
class Box<T> {
    private T the_value;
    Box(T v) {
        this.the_value = v;
    }
    T get() {
        return the_value;
    }
    void set(T v) {
        the_value = v;
    }
}
```

Because we must sometimes mutate in groups (e.g., removing money from one bank account and depositing it in another), it is useful to be able to sequence a group of mutable operations. In Racket, `begin` lets you write a sequence of operations; it evaluates them in order and returns the value of the last one.

Exercise

Define `begin` by desugaring into `let` (and hence into `lambda`).

Even though it is possible to eliminate `begin` as syntactic sugar, it will prove extremely useful for understanding how mutation works. Therefore, we will add a simple, two-term version of sequencing to the core.

This is an excellent illustration of the non-canonical nature of desguaring. We've chosen to add to the core a construct that is certainly not necessary. If our goal was to shrink the size of the interpreter—perhaps at some cost to the size of the input program—we would not make this choice. But our goal in this book is to study pedagogic interpreters, so we choose a larger language because it is more instructive.

8.1.2 Scaffolding

First, let's extend our core language datatype:

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [appC (fun : ExprC) (arg : ExprC)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [lamC (arg : symbol) (body : ExprC)]
  [boxC (arg : ExprC)]
  [unboxC (arg : ExprC)]
  [setboxC (b : ExprC) (v : ExprC)]
  [seqC (b1 : ExprC) (b2 : ExprC)])
```

Observe that in a setboxC expression, both the box position and its new value are expressions. The latter is unsurprising, but the former might be. It means we can write programs such as this in corresponding Racket:

```
(let ([b0 (box 0)]
      [b1 (box 1)])
  (let ([l (list b0 b1)])
    (begin
      (set-box! (first l) 1)
      (set-box! (second l) 2)
      l)))
```

This evaluates to a list of boxes, the first containing 1 and the second 2. Observe that the first argument to the first set-box! instruction was (first l), i.e., an expression that evaluated to a box, rather than just a literal box or an identifier. This is precisely analogous to languages like Java, where one can (taking some type liberties) write

Your output may look like '(#&1 #&2). The #& notation is Racket's abbreviated syntactic prefix for "box".

```
public static void main (String[] args) {
    Box<Integer> b0 = new Box<Integer>(0);
    Box<Integer> b1 = new Box<Integer>(1);

    ArrayList<Box<Integer>> l = new ArrayList<Box<Integer>>();
    l.add(b0);
    l.add(b1);

    l.get(0).set(1);
    l.get(1).set(2);
}
```

Observe that l.get(0) is a compound expression being used to find the appropriate box, and evaluates to the box object on which set is invoked.

For convenience, we will assume that we have implemented desguaring to provide us with (a) let and (b) if necessary, more than two terms in a sequence (which can be desugared into nested sequences). We will also sometimes write expressions in the original Racket syntax, both for brevity (because the core language terms can grow quite large and unwieldy) and so that you can run these same terms in Racket and observe what answers they produce. As this implies, we are taking the behavior in Racket—which is similar to the behavior in just about every mainstream language with mutable objects and structures—as the reference behavior.

8.1.3 Interaction with Closures

Consider a simple counter:

```
(define new-loc
  (let ([n (box 0)])
    (lambda ()
        (begin
          (set-box! n (add1 (unbox n)))
          (unbox n)))))
```

Every time it is invoked, it produces the next integer:

```
> (new-loc)
- number
1
> (new-loc)
- number
2
```

Why does this work? It's because the box is created only once, and bound to `n`, and then closed over. All subsequent mutations affect *the same box*. In contrast, swapping two lines makes a big difference:

```
  (define new-loc-broken
    (lambda ()
      (let ([n (box 0)])
        (begin
          (set-box! n (add1 (unbox n)))
          (unbox n)))))
```

Observe:

```
> (new-loc-broken)
- number
1
> (new-loc-broken)
- number
1
```

In this case, a new box is allocated on every invocation of the function, so the answer each time is the same (despite the mutation inside the procedure). Our implementation of boxes should be certain to preserve this distinction.

The examples above hint at an implementation necessity. Clearly, whatever the environment closes over in `new-loc` must refer to the same box each time. Yet something also needs to make sure that the value in that box is different each time! Look at it more carefully: it must be *lexically* the same, but *dynamically* different. This distinction will be at the heart of our implementation.

8.1.4 Understanding the Interpretation of Boxes

Let's begin by reproducing our current interpreter:

<***interp-take-1***> **::=**

```
  (define (interp [expr : ExprC] [env : Env]) : Value
    (type-case ExprC expr
      [numC (n) (numV n)]
      [idC (n) (lookup n env)]
    [appC (f a) (local ([define f-value (interp f env)])
                  (interp (closV-body f-value)
                          (extend-env (bind (closV-arg f-value)
                                            (interp a env))
                                      (closV-env f-value))))]
    [plusC (l r) (num+ (interp l env) (interp r env))]
    [multC (l r) (num* (interp l env) (interp r env))]
    [lamC (a b) (closV a b env)]
    <boxC-case>
    <unboxC-case>
    <setboxC-case>
    <seqC-case>))
```

Because we've introduced a new kind of value, the box, we have to update the set of values:

<***value-take-1***> **::=**

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)]
  [boxV (v : Value)])
```

Two of these cases should be easy. When we're given a `box` expression, we simply evaluate it and return it wrapped in a `boxV`:

<***boxC-case-take-1***> **::=**

```
[boxC (a) (boxV (interp a env))]
```

Similarly, extracting a value from a box is easy:

<***unboxC-case-take-1***> **::=**

```
[unboxC (a) (boxV-v (interp a env))]
```

By now, you should be constructing a healthy set of test cases to make sure these behave as you'd expect.

Of course, we haven't done any hard work yet. All the interesting behavior is, presumably, hidden in the treatment of `setboxC`. It may therefore surprise you that we're going to look at `seqC` first instead (and you'll see why we included it in the core).

Let's take the most natural implementation of a sequence of two instructions:

<***seqC-case-take-1***> **::=**

```
[seqC (b1 b2) (let ([v (interp b1 env)])
                (interp b2 env))]
```

That is, we evaluate the first term, then the second, and return the result of the second.

You should immediately spot something troubling. We bound the result of evaluating the first term, but didn't subsequently do anything with it. That's okay: presumably the first term contained a mutation expression of some sort, and its value is uninteresting (indeed, note that `set-box!` returns a void value). Thus, another implementation might be this:

<***seqC-case-take-2***> **::=**

```
[seqC (b1 b2) (begin
                (interp b1 env)
                (interp b2 env))]
```

Not only is this slightly dissatisfying in that it just uses the analogous Racket sequencing construct, it still can't possibly be right! This can only work *only if the result of the mutation is being stored somewhere*. But because our interpreter only computes values, and does not perform any mutation itself, any mutations in `(interp b1 env)` are completely lost. This is obviously not what we want.

8.1.5 Can the Environment Help?

Here is another example that can help:

```
(let ([b (box 0)])
  (begin (begin (set-box! b (+ 1 (unbox b)))
                (set-box! b (+ 1 (unbox b))))
         (unbox b)))
```

In Racket, this evaluates to `2`.

Exercise

Represent this expression in `ExprC`.

Let's consider the evaluation of the inner sequence. In both cases, the expression (the representation of `(set-box! ...)`) is exactly identical. Yet something is changing underneath, because these cause the value of the box to go from `0` to `2`! We can "see" this even more clearly if instead we evaluate

```
(let ([b (box 0)])
  (+ (begin (set-box! b (+ 1 (unbox b)))
            (unbox b))
     (begin (set-box! b (+ 1 (unbox b)))
            (unbox b))))
```

which evaluates to `3`. Here, the two calls to `interp` in the rule for addition are sending exactly the same textual expression in both cases. Yet somehow the effects from the left branch of the addition are being felt in the right branch, and we must rule out spukhafte Fernwirkung.

If the interpreter is being given precisely the same expression, how can it possibly avoid producing precisely the same answer? The most obvious way is if the interpreter's other parameter, the environment were somehow different. As of now the exact same environment is sent to both both branches of the sequence and both arms of the addition, so our interpreter—which produces the same output every time on a given input—cannot possibly produce the answers we want.

Here is what we know so far:

1. We must somehow make sure the interpreter is fed different arguments on calls that are expected to potentially produce different results.
2. We must return from the interpreter some record of the mutations made when evaluating its argument expression.

Because the expression is what it is, the first point suggests that we might try to use the environment to reflect the differences between invocations. In turn, the second point suggests that each invocation of the interpreter should also *return* the environment, so it can be passed to the next invocation. Roughly, then, the type of the interpreter might become:

```
; interp : ExprC * Env -> Value * Env
```

That is, the interpreter consumes an expression and environment; it evaluates in that environment, updating it as it proceeds; when the expression is done evaluating, the interpreter returns the answer (as it did before), *along with* an updated environment, which in turn is sent to the next invocation of the interpreter. And the treatment of `setboxC` would somehow impact the environment to reflect the mutation.

Before we dive into the implementation, however, we should consider the consequences of such a change. The environment already serves an important purpose: it holds deferred substitutions. In that respect, it already has a precise semantics—given by substitution—and we must be careful to not alter that. One consequence of its tie to substitution is that it is also the *repository of lexical scope information*. If we were to allow the extended environment escape from one branch of addition and be used in the other, for instance, consider the impact on the equivalent of the following program:

```
(+ (let ([b (box 0)])
     1)
   b)
```

It should be evident that this program has an error: `b` in the right branch of the addition is unbound (the scope of the `b` in the left branch ends with the closing of the `let`—if this is not evident, desugar the above expression to use functions). But the extended environment at the end of interpreting the `let` clearly has `b` bound in it.

Exercise

Work out the above problem in detail and make sure you understand it.

You could try various other related proposals, but they are likely to all have similar failings. For instance, you may decide that, because the problem has to do with additional bindings in the environment, you will instead remove all added bindings in the returned environment. Sounds attractive? Did you remember we have closures?

Exercise

Consider the representation of the following program:

```
(let ([a (box 1)])
  (let ([f (lambda (x) (+ x (unbox a)))])
    (begin
      (set-box! a 2)
      (f 10))))
```

What problems does this example cause?

Rather, we should note that while the *constraints* described above are all valid, the *solution* we proposed is not the only one. What we require are the two conditions enumerated above; observe that neither one actually requires the environment to be the responsible agent. Indeed, it is quite evident that the environment *cannot* be the principal agent.

8.1.6 Introducing the Store

The preceding discussion tells us that we need *two* repositories to accompany the expression, not one. One of them, the environment, continues to be responsible for maintaining lexical scope. But the environment cannot directly map identifiers to their value, because the value might change. Instead, something else needs to be responsible for maintaining the dynamic state of mutated boxes. This latter data structure is called the *store*.

Like the environment, the store is a partial map. Its domain could be any abstract set of names, but it is natural to think of these as numbers, meant to stand for memory locations. This is because the store in the semantics maps directly onto (abstracted) physical memory in the machine, which is traditionally addressed by numbers. Thus the environment maps names to locations, and the store maps locations to values:

```
(define-type-alias Location number)

(define-type Binding
  [bind (name : symbol) (val : Location)])

(define-type-alias Env (listof Binding))
(define mt-env empty)
(define extend-env cons)

(define-type Storage
  [cell (location : Location) (val : Value)])

(define-type-alias Store (listof Storage))
(define mt-store empty)
(define override-store cons)
```

We'll also equip ourselves with a function to look up values in the store, just as we already have one for the environment (which now returns locations instead):

```
(define (lookup [for : symbol] [env : Env]) : Location
  ...)
(define (fetch [loc : Location] [sto : Store]) : Value
  ...)
```

With this, we can refine our notion of values to the correct one:

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)]
  [boxV (l : Location)])
```

Exercise

Fill in the bodies of `lookup` and `fetch`.

8.1.7 Interpreting Boxes

Now we have something that the environment can return, updated, reflecting mutations during the evaluation of the expression, without having to change the environment in any way. Because a function can return only one value, let's define a data structure to hold the new result from the interpreter:

```
(define-type Result
  [v*s (v : Value) (s : Store)])
```

Thus the interpreter's type becomes:
<***interp-mut-struct***> **::=**

```
(define (interp [expr : ExprC] [env : Env] [sto : Store]) : Result
  <ms-numC-case>
  <ms-idC-case>
  <ms-appC-case>
  <ms-plusC/multC-case>
  <ms-lamC-case>
  <ms-boxC-case>
  <ms-unboxC-case>
  <ms-setboxC-case>
  <ms-seqC-case>)
```

The easiest one to dispatch is numbers. Remember that we have to return the store reflecting all mutations that happened while evaluating the given expression. Because a number is a constant, no mutations could have happened, so the returned store is the same as the one passed in:
<***ms-numC-case***> **::=**

```
[numC (n) (v*s (numV n) sto)]
```

A similar argument applies to closure creation; observe that we are speaking of the *creation*, not *use*, of closures:

<***ms-lamC-case***> **::=**

```
[lamC (a b) (v*s (closV a b env) sto)]
```

Identifiers are almost as straightforward, though if you are simplistic, you'll get a type error that will alert you that to obtain a value, you must now look up both in the environment `and in the store`:

<***ms-idC-case***> **::=**

```
[idC (n) (v*s (fetch (lookup n env) sto) sto)]
```

Notice how `lookup` and `fetch` compose to produce the same result that `lookup` alone produced before.

Now things get interesting.

Let's take sequencing. Clearly, we need to interpret the two terms:

```
(interp b1 env sto)
(interp b2 env sto)
```

Oh, but wait. The whole point was to evaluate the second term *in the store returned by the first one*—otherwise there would have been no point to all these changes. Therefore, instead we must evaluate the first term, capture the resulting store, and use it to evaluate the second. (Evaluating the first term also yields its value, but sequencing ignores this value and assumes the first time was run purely for its potential mutations.) We will write this in a stylized manner:

<***ms-seqC-case***> **::=**

```
[seqC (b1 b2) (type-case Result (interp b1 env sto)
                [v*s (v-b1 s-b1)
                     (interp b2 env s-b1)])]
```

This says to `(interp b1 env sto)`; name the resulting value and store `v-b1` and `s-b1`, respectively; and evaluate the second term in the store from the first: `(interp b2 env s-b1)`. The result will be the value and store returned by the second term, which is what we expect. The fact that the first term's effect is only on the store can be read from the code because, though we bind `v-b1`, we never subsequently use it.

Do Now!

> Spend a moment contemplating the code above. You'll soon need to adjust your eyes to read this pattern fluently.

Now let's move on to the binary arithmetic primitives. These are similar to sequencing in that they have two sub-terms, but in this case we really do care about the value from each branch. As usual, we'll look at only `plusC` since `multC` is virtually identical.

<***ms-plusC/multC-case***> **::=**

```
[plusC (l r) (type-case Result (interp l env sto)
               [v*s (v-l s-l)
                    (type-case Result (interp r env s-l)
                      [v*s (v-r s-r)
                           (v*s (num+ v-l v-r) s-r)])])]
```

Observe that we've unfolded the sequencing pattern out another level, so we can hold on to both results and supply them to `num+`.

Here's an important distinction. When we evaluate a term, we usually use the same environment for all its sub-terms in accordance with the scoping rules of the language. The environment thus flows in a recursive-descent pattern. In contrast, the store is *threaded*: rather than using the same store in all branches, we take the store from one branch and pass it on to the next, and take the result and send it back out. This pattern is called *store-passing style*.

Now the penny drops. We see that store-passing style is our secret ingredient: it enables the environment to preserve lexical scope while still giving a binding structure that can reflect changes. Our intution told us that the environment had to somehow participate in obtaining different results for the same expression, and we can now see how it does: not directly, by itself changing, but indirectly, by referring to the store, which updates. Now we only need to see how the store itself "changes".

Let's begin with boxing. To store a value in a box, we have to first allocate a new place in the store where its value will reside. The value corresponding to a box will then remember this location, for use in box mutation.

***<ms-boxC-case>* ::=**

```
[boxC (a) (type-case Result (interp a env sto)
            [v*s (v-a s-a)
                 (let ([where (new-loc)])
                   (v*s (boxV where)
                        (override-store (cell where v-a)
                                        s-a)))])]
```

Do Now!

Observe that we have relied above on `new-loc`, which is itself implemented in terms of boxes! This is outright cheating. How would you modify the interpreter so that we no longer need an mutating implementation of `new-loc`?

To eliminate this style of `new-loc`, the simplest option would be to add yet another parameter to and return value from the interpreter, which represents the largest address used so far. Every operation that allocates in the store would return an incremented address, while all others would return it unchanged. In other words, this is precisely another application of the store-passing pattern. Writing the interpreter this way would make it extremely unwieldy and might obscure the more important use of store-passing for the store itself, which is why we have not done so. However, it is important to make sure that we can: that's what tells us that we are not reliant on boxes to add boxes to the language.

Now that boxes are recording the location in memory, getting the value corresponding to them is easy.

***<ms-unboxC-case>* ::=**

```
[unboxC (a) (type-case Result (interp a env sto)
              [v*s (v-a s-a)
                   (v*s (fetch (boxV-l v-a) s-a) s-a)])]
```

It's the same pattern we saw before, where we have to use `fetch` to obtain the actual value residing at that location. Note that we are relying on Racket to halt with an error if the underlying value isn't actually a `boxV`; otherwise it would be dangerous to not check, since this would be tantamount to dereferencing arbitrary memory (as C programs can, sometimes with disastrous consequences).

Let's now see how to update the value held in a box. First we have to evaluate the box expression to obtain a box, and the value expression to obtain the new value to store in it. The box's value is going to be a `boxV` holding a location.

In principle, we want to "change", or override, the value at that location in the store. We can do this in two ways.

1. One is to traverse the store, find the old binding for that location, and replace it with the new one, copying all the other store bindings unchanged.

2. The other, lazier, option is to simply extend the store with a new binding for that location, which works provided we always obtain the most recent binding for a location (which is how `lookup` works in the environment, so `fetch` presumably also does in the store).

The code below is written to be independent of these options:

***<ms-setboxC-case>* ::=**

```
[setboxC (b v) (type-case Result (interp b env sto)
                 [v*s (v-b s-b)
                      (type-case Result (interp v env s-b)
                        [v*s (v-v s-v)
                             (v*s v-v
                                  (override-store (cell (boxV-l v-
b)
                                                                 v-v)
                                                          s-v))])])]
```

However, because we've implemented `override-store` as `cons` above, we've actually taken the lazier (and slightly riskier, because of its dependence on the implementation of `fetch`) option.

Exercise

> Implement the other version of store alteration, whereby we update an existing binding and thereby avoid multiple bindings for a location in the store.

Exercise

When we look for a location to override the value stored at it, can the location fail to be present? If so, write a program that demonstrates this. If not, explain what invariant of the interpreter prevents this from happening.

Alright, we're now done with everything other than application! Most of application should already be familiar: evaluate the function position, evaluate the argument position, interpret the closure body in an extension of the closure's environment...but how do stores interact with this?

<***ms-appC-case***> **::=**

```
[appC (f a)
      (type-case Result (interp f env sto)
        [v*s (v-f s-f)
             (type-case Result (interp a env s-f)
               [v*s (v-a s-a)
                    <ms-appC-case-main>])])]
```

Let's start by thinking about extending the closure environment. The name we're extending it with is obviously the name of the function's formal parameter. But what location do we bind it to? To avoid any confusion with already-used locations (a confusion we will explicitly introduce later! [REF]), let's just allocate a new location. This location is used in the environment, and the value of the argument resides at this location in the store:

<***ms-appC-case-main***> **::=**

```
(let ([where (new-loc)])
  (interp (closV-body v-f)
          (extend-env (bind (closV-arg v-f)
                            where)
                      (closV-env v-f))
          (override-store (cell where v-a) s-a)))
```

Because we have not said the function parameter is mutable, there is no real need to have implemented procedure calls this way. We could instead have followed the same strategy as before. Indeed, observe that the mutability of this location will never be used: only `setboxC` changes what's in an existing store location (the `override-store` above is technically a store *initialization*), and then only when they are referred to by `boxV`s, but no box is being allocated above. However, we have chosen to implement application this way for uniformity, and to reduce the number of cases we'd have to handle.

You could call this the useless app store.

Exercise

It's a useful exercise to try to limit the use of store locations *only* to boxes. How many changes would you need to make?

8.1.8 The Bigger Picture

Even though we've finished the implementation, there are still many subtleties and insights to discuss.

1. Implicit in our implementation is a subtle and important decision: the *order of evaluation*. For instance, why did we not implement addition thus?

   ```
   [plusC (l r) (type-case Result (interp r env sto)
                  [v*s (v-r s-r)
                       (type-case Result (interp l env s-r)
                         [v*s (v-l s-l)
                              (v*s (num+ v-l v-r) s-l)])])]
   ```

   It would have been perfectly consistent to do so. Similarly, embodied in the pattern of store-passing is the decision to evaluate the function position before the argument. Observe that:

   (a) Previously, we delegated such decisions to the underlying language implementation. Now, store-passing has forced us to *sequentialize* the computation, and hence make this decision ourselves (whether we realized it or not).

   (b) Even more importantly, *this decision is now a semantic one*. Before there were mutations, one branch of an addition, for instance, could not affect the value produced by the other branch. Because each branch can have mutations that impact the value of the other, we *must* choose some order so that programmers can predict what their program is going to do! Being forced to write a store-passing interpreter has made this clear.

   The only effect they could have was halting with an error or failing to terminate—which, to be sure, are certainly observable effects, but at a much more gross level. A program would not terminate with two different answers depending on the order of evaluation.

2. Observe that in the application rule, we are passing along the *dynamic* store, i.e., the one resulting from evaluating both function and argument. This is precisely the opposite of what we said to do with the environment. This distinction is critical. The store is, in effect, "dynamically scoped", in that it reflects the history of the computation, not its lexical shape. Because we are already using the term "scope" to refer to the bindings of identifiers, however, it would be confusing to say "dynamically scoped" to refer to the store. Instead, we simply say that it is *persistent*.

   Languages sometimes dangerously conflate these two. In C, for instance, values bound to local identifiers are allocated (by default) on the stack. However, the stack matches the environment, and hence disappears upon completion of the call. If the call, however, returned references to any of these values, these references are now pointing to unused or even overridden memory: a genuine source of serious errors in C programs. The problem is that the values themselves persist; it is only the identifiers that refer to them that have lexical scope.

3. We have already discussed how there are two strategies for overriding the store: to simply extend it (and rely on `fetch` to extract the newest one) or to "search-and-replace". The latter strategy has the virtue of not holding on to useless store bindings that will can never be obtained again.

   However, this does not cover all the wasted memory. Over time, we cease to be able to access some boxes entirely: e.g., if they are bound to only one identifier, and that identifier is no longer in scope. These locations are called *garbage*. Thinking more conceptually, garbage locations are those whose elimination does not have any impact on the value produced by a program. There are many strategies for identifying and reclaiming garbage locations, usually called *garbage collection* [REF].

4. It's very important to evaluate every expression position and thread the store that results from it. Consider, for instance, this implementation of `unboxC`:

   ```
   [unboxC (a) (type-case Result (interp a env sto)
                 [v*s (v-a s-a)
                      (v*s (fetch (boxV-l v-a) sto) s-a)])]
   ```

   Did you notice? We `fetch`ed the location from `sto`, not `s-a`. But `sto` reflects mutations up to but before the evaluation of the `unboxC` expression, not any *within* it. Could there possibly be any? Mais oui!

   ```
   (let ([b (box 0)])
     (unbox (begin (set-box! b 1)
                   b)))
   ```

   With the incorrect code above, this would evaluate to `0` rather than `1`.

5. Here's another, similar, error:

   ```
   [unboxC (a) (type-case Result (interp a env sto)
                 [v*s (v-a s-a)
                      (v*s (fetch (boxV-l v-a) s-a) sto)])]
   ```

   How do we break this? Well, we're returning the old store, the one before any mutations in the `unboxC` happened. Thus, we just need the outside context to depend on one of them.

   ```
   (let ([b (box 0)])
     (+ (unbox (begin (set-box! b 1)
                      b))
        (unbox b)))
   ```

   This should evaluate to `2`, but because the store being returned is one where `b`'s location is bound to the representation of `0`, the result is `1`.

   If we combined both bugs above—i.e., using `sto` twice in the last line instead of `s-a` twice—this expression would evaluate to `0` rather than `2`.

   Exercise

Go through the interpreter; replace every reference to an updated store with a reference to one before update; make sure your test cases catch all the introduced errors!

6. Observe that these uses of "old" stores enable us to perform a kind of *time travel*: because mutation introduces a notion of time, these enable us to go back in time to when the mutation had not yet occurred. This sounds both interesting and perverse; does it have any use?

   It does! Imagine that instead of directly mutating the store, we introduce the idea of a journal of *intended* updates to the store. The journal flows in a threaded manner just like the real store itself. Some instruction creates a new journal; after that, all lookups first check the journal, and only if the journal cannot find a binding for a location is it looked for in the actual store. There are two other new instructions: one to *discard* the journal (i.e., perform time travel), and the other to *commit* it (i.e., all of its edits get applied to the real store).

   This is the essence of *software transactional memory*. Each thread maintains its own journal. Thus, one thread does not see the edits made by the other before committing (because each thread sees only its own journal and the global store, but not the journals of other threads). At the same time, each thread gets its own consistent view of the world (it sees edits it made, because these are recorded in the journal). If the transaction ends successfully, all threads atomically see the updated global store. If the transaction aborts, the discarded journal takes with it all changes and the state of the thread reverts (modulo global changes committed by other threads).

   Software transactional memory offers one of the most sensible approaches to tackling the difficulties of multi-threaded programming, if we insist on programming with shared mutable state. Because most computers have only one global store, however, maintaining the journals can be expensive, and much effort goes into optimizing them. As an alternative, some hardware architectures have begun to provide direct support for transactional memory by making the creation, maintenance, and commitment of journals as efficient as using the global store, removing one important barrier to the adoption of this idea.

   Exercise

   Augment the language with the journal features of software transactional memory journal.

Exercise

An alternate implementation strategy is to have the environment map names to *boxed* `Values`. We don't do it here because it: (a) would be cheating, (b) wouldn't tell us how to implement the same feature in a language without boxes, (c) doesn't necessarily carry over to other mutation operations, and (d) most of all, doesn't really give us *insight* into what is happening here.

It is nevertheless useful to understand, not least because you may find it a useful strategy to adopt when implementing your own language. Therefore, alter the implementation to obey this strategy. Do you still need store-passing style? Why or why not?

8.2 Variables

Now that we've got structure mutation worked out, let's consider the other case: variable mutation.

8.2.1 Terminology

First, our choice of terms. We've insisted on using the word "identifier" before because we wanted to reserve "variable" for what we're about to study. In Java, when we say (assuming `x` is locally bound, e.g., as a method parameter)

```
x = 1;
x = 3;
```

we're asking to *change* the value of `x`. After the first assignment, the value of `x` is `1`; after the second one, it's `3`. Thus, the value of `x` *varies* over the course of the execution of the method.

Now, we also use the term "variable" in mathematics to refer to function parameters. For instance, in $f(y) = y + 3$ we say that y is a "variable". That is called a variable because it varies *across invocations*; however, *within* each invocation, it has the same value in its scope. Our identifiers until now have corresponded to this notion of a variable. In contrast, programming variables can vary even *within* each invocation, like the Java `x` above.

If the identifier was bound to a box, then it remained bound to the same box value. It's the content of the box that changed, not which box the identifier was bound to.

Henceforth, we will use *variable* when we mean an identifier whose value can change within its scope, and *identifier* when this cannot happen. If in doubt, we might play it safe and use "variable"; if the difference doesn't really matter, we might use either one. It is less important to get caught up in these specific terms than to understand that they represent a distinction that matters [REF].

8.2.2 Syntax

Whereas other languages overload the mutation syntax (`=` or `:=`), in Racket they are kept distinct: `set!` is used to mutate variables. This forces Racket programmers to confront the distinction we introduced at the beginning of section 8. We will, of course, sidestep these syntactic issues in our core language by using different constructs for boxes and for variables.

The first thing to note about variable mutation is that, although it too has two subterms like box mutation (`setboxC`), its syntax is fundamentally different. To understand why, let's return to our Java fragment:

```
x = 3;
```

In this setting, we cannot write an arbitrary expression in place of `x`: we must literally write the name of the identifier itself. That is because, if it were an expression position, then we could evaluate it, yielding a value: for instance, if `x` were previously bound to `1`, this would be tantamout to writing the following statement:

```
1 = 3;
```

But this is, of course, nonsensical! We can't assign a new value to `1`, and indeed `1` is pretty much the definition of immutable. Thus, what we instead want is to find *where* `x` is in the store, and change the value held over there.

Here's another way to see this. Suppose the local variable `o` were bound to some `String` object; let's call this object s. Say we write

```
o = new String("a new string");
```

Are we trying to change s in any way? Certainly not: this statement intends to leave s alone. It only wants to change the value that `o` is referring to, so that subsequent references evaluate to this new `string` object instead.

8.2.3 Interpreting Variables

We'll start by reflecting this in our syntax:

```
(define-type ExprC
  [numC (n : number)]
  [varC (s : symbol)]
  [appC (fun : ExprC) (arg : ExprC)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [lamC (arg : symbol) (body : ExprC)]
  [setC (var : symbol) (arg : ExprC)]
  [seqC (b1 : ExprC) (b2 : ExprC)])
```

Observe that we've jettisoned the box operations, but kept sequencing because it's handy around mutation. Importantly, we've now added the `setC` case, and its first sub-term is not an expression but the literal name of a variable. We've also renamed `idC` to `varC`.

Because we've gotten rid of boxes, we can also get rid of the special box values. When the only kind of mutation you have is variables, you don't need new kinds of values.

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)])
```

As you might imagine, to support variables we need the same store-passing style that we've seen before (section 8.1.7), and for the same reasons. What differs is in precisely how we use it. Because sequencing is interpreted in just the same way (observe that the code for it does not depend on boxes versus variables), that leaves us just the variable mutation case to handle.

First, we might as well evaluate the value expression and obtain the updated store:
<***setC-case***> **::=**

```
[setC (var val) (type-case Result (interp val env sto)
                  [v*s (v-val s-val)
                       <rest-of-setC-case>])]
```

What now? Remember we just said that we don't want to fully evaluate the variable, because that would just give the value it is bound to. Instead, we want to know which memory location it corresponds to, and update what is stored at that memory location; this *latter* part is just the same thing we did when mutating boxes:
<***rest-of-setC-case***> **::=**

```
(let ([where (lookup var env)])
  (v*s v-val
       (override-store (cell where v-val)
                       s-val)))
```

The very interesting new pattern we have here is this. When we added boxes, in the `idC` case, we looked up an identifier in the environment, and immediately fetched the value at that location from the store; the composition yielded a value, just as it used to before we added stores. Now, however, we have a new pattern: looking up an identifier in the environment *without* subsequently fetching its value from the store. The result of invoking just `lookup` is traditionally called an *l-value*, for "left-hand-side (of an assignment) value". This is a fancy way of saying "memory address", and stands in contast to the actual values that the store yields: observe that it does not directly correspond to anything in the type `Value`.

And we're done! We did all the hard work when we implemented store-passing style (and also in that application allocated new locations for variables).

8.3 The Design of Stateful Language Operations

Though most programming languages include one or both kinds of state we have studied, their admission should not be regarded as a trivial or foregone matter. On the one hand, state brings some vital benefits:

- State provides a form of *modularity*. As our very interpreter demonstrates, without explicit stateful operations, to achieve the same effect:
  - We would need to add explicit parameters and return values that pass the equivalent of the store around.
  - These changes would have to be made to *all* procedures that may be involved in a communication path between producers and consumers of state.

Thus, a different way to think of state in a programming language is that it is an *implicit parameter already passed to and returned from all procedures*, without imposing that burden on the programmer. This enables procedures to communicate "at a distance" without all the intermediaries having to be aware of the communication.

- State makes it possible to construct dynamic, cyclic data structures, or at least to do so in a relatively straightforward manner (section 9).

- State gives procedures *memory*, such as `new-loc` above. If a procedure could not remember things for itself, the callers would need to perform the remembering on its behalf, employing the moral equivalent of store-passing. This is not only unwieldy, it creates the potential for a caller to interfere with the memory for its own nefarious purposes (e.g., a caller might purposely send back an old store, thereby obtaining a reference already granted to some other party, through which it might launch a correctness or security attack).

On the other hand, state imposes real costs on programmers as well as on programs that process programs (such as compilers). One is "aliasing", which we discuss later [REF]. Another is "referential transparency", which too I hope to return to [REF]. Finally, we have described above how state provides a form of modularity. However, this same description could be viewed as that of a back-channel of communication that the intermediaries did not know and could not monitor. In some (especially security and distributed system) settings, such back-channels can lead to collusion, and can hence be extremely dangerous and undesirable.

Because there is no optimal answer, it is probably wise to include mutation operators but to carefully delinate them. In Standard ML, for instance, there is no variable mutation, because it is considered unnecessary. Instead, the language has the equivalent of boxes (called `ref`s). One can easily simulate variables using boxes (e.g., see `new-loc` and consider how it would be written with variables instead), so no expressive power is lost, though it does create more potential for aliasing than variables alone would have ([REF aliasing]) if the boxes are not used carefully.

In return, however, developers obtain expressive *types*: every data structure is considered immutable unless it contains a `ref`, and the presence of a `ref` is a warning to both developers and programs (such as compilers) that the underlying value may keep changing. Thus, for instance, if `b` is a box, a developer should be aware that replacing all instances of `(unbox b)` with `v`, where `v` is bound to `(unbox b)`, is unwise: the former always fetches the *current* value in the box, while the latter may be referring to an older content. (Conversely, if the developer wants the value at a certain point in time, oblivious to future mutations to the box, they should be sure to retrieve and bind it rather than always use `unbox`.)

8.4 Parameter Passing

In our current implementation, on every function call, we allocate a fresh location in the store for the parameter. This means the following program

```
(let ([f (lambda (x) (set! x 3))])
  (let ([y 5])
    (begin
      (f y)
      y)))
```

evaluates to 5, not 3. That is because the value of the formal parameter x is held at a different location than that of the actual parameter y, so the mutation affects the location of x, leaving y unscathed.

Now suppose, instead, that application behaved as follows. When the actual parameter is a variable, and hence has a location in memory, instead of allocating a new location for the value, it simply passes along the existing one for the variable. Now the formal parameter is referring to the *same store location* as the actual: i.e., they are *variable aliases*. Thus any mutation on the formal will leak back out into the calling context; the above program would evaluate to 3 rather than 5. These is called a *call-by-reference* parameter-passing strategy.

Instead, our interpreter implements *call-by-value*, and this is the same strategy followed by languages like Java. This causes confusion because *when the value is itself mutable*, changes made to the value in the callee are observed by the caller. However, that is simply an artifact of mutable values, not of the calling strategy. Please avoid this confusion!

For some years, this power was considered a good idea. It was useful because programmers could write abstractions such as swap, which swaps the *value of two variables* in the caller. However, the disadvantages greatly outweigh the advantages:

- A careless programmer can alias a variable in the caller and modify it without realizing they have done so, and the caller may not even realize this has happened until some obscure condition triggers it.
- Some people thought this was necessary for efficiency: they assumed the alternative was to *copy* large data structures. However, call-by-value is compatible with passing just the address of the data structure. You only need make a copy if (a) the data structure is mutable, (b) you do not want the caller to be able to mutate it, and (c) the language does not itself provide immutability annotations or other mechanisms.
- It can force non-uniform and hence non-modular reasoning. For instance, suppose we have the procedure:

  ```
  (define (f g)
    (let ([x 10])
      (begin
        (g x)
        ...)))
  ```

  If the language were to permit by-reference parameter passing, then the programmer cannot locally—i.e., just from the above code—determine what the value of x will be in the ellipses.

At the very least, then, if the language is going to permit by-reference parameters, it should let the *caller* determine whether to pass the reference—i.e., let the callee share the memory address of the caller's variable—or not. However, even this option is not quite as attractive as it may sound, because now the callee faces a symmetric problem, not knowing whether its parameters are aliased or not. In traditional, sequential programs this is less of a concern, but if the procedure is *reentrant*, the callee faces precisely the same predicaments.

At some point, therefore, we should consider whether any of this fuss is worthwhile. Instead, callers who want the callee to perform a mutation could simply send a boxed value to the callee. The box signals that the caller accepts—indeed, invites—the callee to perform a mutation, and the caller can extract the value when it's done. This does obviate the ability to write a simple swapper, but that's a small price to pay for genuine software engineering concerns.

9 Recursion and Cycles: Procedures and Data

Recursion is the act of self-reference. When we speak of recursion in programming languages, we may have one of (at least) two meanings in mind: recursion in data, and recursion in control (i.e., of program behavior—that is to say, of functions).

9.1 Recursive and Cyclic Data

Recursion in data can refer to one of two things. It can mean referring to something of the same *kind*, or referring to the same *thing* itself.

Recursion of the same kind leads to what we traditionally call *recursive data*. For instance, a tree is a recursive data structure: each vertex can have multiple children, each of which is itself a tree. But if we write a procedure to traverse the nodes of a tree, we expect it to terminate without having to keep track of which nodes it has already visited. They are finite data structures.

In contrast, a graph is often a *cyclic* datum: a node refers to another node, which may refer back to the original one. (Or, for that matter, a node may refer directly to itself.) When we traverse a graph, absent any explicit checks for what we have already visited, we should expect a computation to *diverge*, i.e., not terminate. Instead, graph algorithms need a memory of what they have visited to avoid repeating traversals.

Adding recursive data, such as lists and trees, to our language is quite straightforward. We mainly require two things:

1. The ability to create compound structures (such as nodes that have references to children).

2. The ability to bottom-out the recursion (such as leaves).

Exercise

Add lists and binary trees as built-in datatypes to the programming language.

Adding cyclic data is more subtle. Consider the simplest form of cyclic datum, a cell referring back to itself: