6.4.2 The Top-Level Scope

Matters become more complex when we contemplate top-level definitions in many languages. For instance, some versions of Scheme (which is a paragon of lexical scoping) allow you to write this:

```
(define y 1)
(define (f x) (+ x y))
```

which seems to pretty clearly suggest where the `y` in the body of `f` will come from, except:

```
(define y 1)
(define (f x) (+ x y))
(define y 2)
```

is legal and `(f 10)` produces `12`. Wait, you might think, always take the last one! But:

```
(define y 1)
(define f (let ((z y)) (lambda (x) (+ x y z))))
(define y 2)
```

Here, `z` is bound to the first value of `y` whereas the inner `y` is bound to the second value. There is actually a valid explanation of this behavior in terms of lexical scope, but it can become convoluted, and perhaps a more sensible option is to prevent such redefinition. Racket does precisely this, thereby offering the convenience of a top-level without its pain.

Most "scripting" languages exhibit similar problems. As a result, on the Web you will find enormous confusion about whether a certain language is statically- or dynamically-scoped, when in fact readers are comparing behavior inside functions (often static) against the top-level (usually dynamic). Beware!

6.5 Exposing the Environment

If we were building the implementation for others to use, it would be wise and a courtesy for the exported interpreter to take only an expression and list of function definitions, and invoke our defined `interp` with the empty environment. This both spares users an implementation detail, and avoids the use of an interpreter with an incorrect environment. In some contexts, however, it can be useful to expose the environment parameter. For instance, the environment can represent a set of pre-defined bindings: e.g., if the language wishes to provide `pi` automatically bound to `3.2` (in Indiana).

7 Functions Anywhere

The introduction to the Scheme programming language definition establishes this design principle:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. [REF]

As design principles go, this one is hard to argue with. (Some restrictions, of course, have good reason to exist, but this principle forces us to argue for them, not admit them by default.) Let's now apply this to functions.

One of the things we stayed coy about when introducing functions (section 5) is exactly where functions go. We may have suggested we're following the model of an idealized DrRacket, with definitions and their uses kept separate. But, inspired by the Scheme design principle, let's examine how *necessary* that is.

Why can't functions definitions be expressions? In our current arithmetic-centric language we face the uncomfortable question "What value does a function *definition* represent?", to which we don't really have a good answer. But a real programming language obviously computes more than numbers, so we no longer need to confront the question in this form; indeed, the answer to the above can just as well be, "A function value". Let's see how that might work out.

What can we do with functions as values? Clearly, functions are a distinct kind of value than a number, so we cannot, for instance, add them. But there is one evident thing we can do: apply them to arguments! Thus, we can allow function values to appear in the function position of an application. The behavior would, naturally, be to apply the function. Thus, we're proposing a language where the following would be a valid program (where I've used brackets so we can easily identify the function)

```
(+ 2 ([define (f x) (* x 3)] 4))
```

and would evaluate to `(+ 2 (* 4 3))`, or `14`. (Did you see that I just used substitution?)

7.1 Functions as Expressions and Values

Let's first define the core language to include function definitions:

<***expr-type***> **::=**

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  <app-type>
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  <fun-type>)
```

For now, we'll simply copy function definitions into the expression language. We're free to change this if necessary as we go along, but for now it at least allows us to reuse our existing test cases.

<***fun-type-take-1***> **::=**

```
[fdC (name : symbol) (arg : symbol) (body : ExprC)]
```

We also need to determine what an application looks like. What goes in the function position of an application? We want to allow an entire function definition, not just

its name. Because we've lumped function definitions in with all other expressions, let's allow an arbitrary expression here, but with the understanding that we want only function definition expressions:

<***app-type***> **::=**

```
[appC (fun : ExprC) (arg : ExprC)]
```

We might consider more refined datatypes that split function definitions apart from other kinds of expressions. This amounts to trying to classify different kinds of expressions, which we will return to when we study types. [REF]

With this definition of application, we no longer have to look up functions by name, so the interpreter can get rid of the list of function definitions. If we need it we can restore it later, but for now let's just explore what happens with function definitions are written at the point of application: so-called *immediate* functions.

Now let's tackle `interp`. We need to add a case to the interpreter for function definitions, and this is a good candidate:

```
[fdC (n a b) expr]
```

Do Now!

What happens when you add this?

Immediately, we see that we have a problem: the interpreter no longer always returns numbers, so we have a type error.

We've alluded periodically to the answers computed by the interpreter, but never bothered gracing these with their own type. It's time to do so now.

<***answer-type-take-1***> **::=**

```
(define-type Value
  [numV (n : number)]
  [funV (name : symbol) (arg : symbol) (body : ExprC)])
```

We're using the suffix of `V` to stand for *values*, i.e., the result of evaluation. The pieces of a `funV` will be precisely those of a `fdC`: the latter is input, the former is output. By keeping them distinct we allow each one to evolve independently as needed.

Now we must rewrite the interpreter. Let's start with its type:

<***interp-hof***> **::=**

```
(define (interp [expr : ExprC] [env : Env]) : Value
  (type-case ExprC expr
    <interp-body-hof>))
```

This change naturally forces corresponding type changes to the `Binding` datatype and to `lookup`.

Exercise

Modify `Binding` and `lookup`, appropriately.

<***interp-body-hof***> **::=**

```
[numC (n) (numV n)]
[idC (n) (lookup n env)]
```
<app-case>
<plus/mult-case>
<fun-case>

Clearly, numeric answers need to be wrapped in the appropriate numeric answer constructor. Identifier lookup is unchanged. We have to slightly modify addition and multiplication to deal with the fact that the interpreter returns `Values`, not numbers:

<***plus/mult-case***> **::=**

```
[plusC (l r) (num+ (interp l env) (interp r env))]
[multC (l r) (num* (interp l env) (interp r env))]
```

It's worth examining the definition of one of these helper functions:

```
(define (num+ [l : Value] [r : Value]) : Value
  (cond
    [(and (numV? l) (numV? r))
     (numV (+ (numV-n l) (numV-n r)))]
    [else
     (error 'num+ "one argument was not a number")]))
```

Observe that it checks that both arguments are numbers before performing the addition. This is an instance of a *safe* run-time system. We'll discuss this topic more when we get to types. [REF]

There are two more cases to cover. One is function definitions. We've already agreed these will be their own kind of value:

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

```
[fdC (n a b) (funV n a b)]
```

That leaves one case, application. Though we no longer need to look up the function definition, we'll leave the code structured as similarly as possible:

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

```
[appC (f a) (local ([define fd f])
              (interp (fdC-body fd)
                      (extend-env (bind (fdC-arg fd)
                                        (interp a env))
                                  mt-env)))]
```

In place of the lookup, we reference `f` which is the function definition, sitting right there. Note that, because any expression can be in the function definition position, we really ought to harden the code to check that it is indeed a function.

Do Now!

What does *is* mean? That is, do we want to check that the function definition position is syntactically a function definition (`fdC`), or only that it evaluates to one (`funV`)? Is there a difference, i.e., can you write a program that satisfies one condition but not the other?

We have two choices:

1. We can check that it syntactically is an `fdC` and, if it isn't reject it as an error.
2. We can evaluate it, and check that the resulting *value* is a function (and signal an error otherwise).

We will take the latter approach, because this gives us a much more flexible language. In particular, even if we can't immediately imagine cases where we, as humans, might need this, it might come in handy when a program needs to generate code. And we're writing precisely such a program, namely the desugarer! (See section 7.5.) As a result, we'll modify the application case to evaluate the function position:

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

```
[appC (f a) (local ([define fd (interp f env)])
              (interp (funV-body fd)
                      (extend-env (bind (funV-arg fd)
                                        (interp a env))
                                  mt-env)))]
```

Exercise

Modify the code to perform both versions of this check.

And with that, we're done. We have a complete interpreter! Here, for instance, are some of our old tests again:

```
(test (interp (plusC (numC 10) (appC (fdC 'const5 '_ (numC 5)) (numC 10)))
              mt-env)
      (numV 15))

(test/exn (interp (appC (fdC 'f1 'x (appC (fdC 'f2 'y (plusC (idC 'x) (idC 'y)))
                                          (numC 4)))
                        (numC 3))
                  mt-env)
          "name not found")
```

7.2 Nested What?

The body of a function definition is an arbitrary expression. A function definition is itself an expression. That means a function definition can contain a...function definition. For instance:

<***nested-fdC***> **::=**

```
(fdC 'f1 'x
     (fdC 'f2 'x
          (plusC (idC 'x) (idC 'x))))
```

Evaluating this isn't very interesting:

```
(funV 'f1 'x (fdC 'f2 'x (plusC (idC 'x) (idC 'x))))
```

But suppose we apply the above function to something:

<***applied-nested-fdC***> **::=**

```
(appC <nested-fdC>
      (numC 4))
```

Now the answer becomes more interesting:

```
(funV 'f2 'x (plusC (idC 'x) (idC 'x)))
```

It's almost as if applying the outer function had no impact on the inner function at all. Well, why should it? The outer function introduces an identifier which is promptly masked by the inner function introducing one of the *same name*, thereby *masking* the outer definition if we obey static scope (as we should!). But that suggests a different program:

```
(appC (fdC 'f1 'x
           (fdC 'f2 'y
                (plusC (idC 'x) (idC 'y))))
      (numC 4))
```

This evaluates to:

```
(funV 'f2 'y (plusC (idC 'x) (idC 'y)))
```

Hmm, that's interesting.

Do Now!

What's interesting?

To see what's interesting, let's apply this once more:

```
(appC (appC (fdC 'f1 'x
                 (fdC 'f2 'y
                      (plusC (idC 'x) (idC 'y))))
            (numC 4))
      (numC 5))
```

This produces an error indicating that the identifier representing `x` isn't bound!

But it's bound by the function named `f1`, isn't it? For clarity, let's switch to representing it in our hypothetical Racket syntax:

```
((define (f1 x)
   ((define (f2 y)
      (+ x y))
    4))
 5)
```

On applying the outer function, we would expect `x` to be substituted with `5`, resulting in

```
((define (f2 y)
   (+ 5 y))
 4)
```

which on further application and substitution yields `(+ 5 4)` or `9`, not an error.

In other words, we're again failing to faithfully capture what substitution would have done. A function value needs to *remember the substitutions* that have already been applied to it. Because we're representing substitutions using an environment, a function value therefore needs to be bundled with an environment. This resulting data structure is called a *closure*.

On the other hand, observe that with substitution, as we've defined it, we would be replacing `x` with `(numV 4)`, resulting in a function body of `(plusC (numV 5) (idC 'y))`, which does not type. That is, substitution is predicated on the assumption that the type of answers is a form of syntax. It is actually possible to carry through a study of even very advanced programming constructs under this assumption, but we won't take that path here.

While we're at it, observe that the `appC` case above uses `funV-arg` and `funV-body`, but not `funV-name`. Come to think of it, why did a function need a name? so that we could find it. But if we're using the interpreter to find the function for us, then there's nothing to find and fetch. Thus the name is merely descriptive, and might as well be a comment. In other words, a function no more needs a name than any other immediate constant: we don't name every use of `3`, for instance, so why should we name every use of a function? A function is *inherently* anonymous, and we should separate its definition from its naming.

(But, you might say, this argument only makes sense if functions are always written in-place. What if we want to put them somewhere else? Won't they need names then? They will, and we'll return to this (section 7.5).)

7.3 Implementing Closures

We need to change our representation of values to record closures rather than raw function text:

<*answer-type*> **::=**

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

While we're at it, we might as well alter our syntax for defining functions to drop the useless name. This construct is historically called a *lambda*:

<*fun-type*> **::=**

```
[lamC (arg : symbol) (body : ExprC)]
```

When encountering a function definition, the interpreter must now remember to save the substitutions that have been applied so far:

"Save the environment! Create a closure today!" —Cormac Flanagan

<*fun-case*> **::=**

```
[lamC (a b) (closV a b env)]
```

This saved set, not the empty environment, must be used when applying a function:

<*app-case*> **::=**

```
[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))))]
```

There's actually another possibility: we could use the environment present at the point of application:

```
[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))
                                  env)))]
```

Exercise

What happens if we extend the dynamic environment instead?

In retrospect, it becomes even more clear why we interpreted the body of a function in the empty environment. When a function is defined at the top-level, it is not "closed over" any identifiers. Therefore, our previous function applications have been special cases of this form of application.

7.4 Substitution, Again

We have seen that substitution is instructive in thinking through how to implement lambda functions. However, we have to be careful with substitution itself! Suppose we have the following expression (to give lambda functions their proper Racket syntax):

```
(lambda (f)
  (lambda (x)
    (f 10)))
```

Now suppose we substitute for `f` the following expression: `(lambda (y) (+ x y))`. Observe that it has a free identifier (`x`), so if it is ever evaluated, we would expect to get an unbound identifier error. Substitution would appear to give:

```
(lambda (x)
  ((lambda (y) (+ x y)) 10))
```

But observe that this latter program has no free identifiers!

That's because we have too naive a version of substitution. To prevent unexpected behavior like this (which is a form of dynamic binding), we need to define *capture-free substitution*. It works roughly as follows: we first *consistently rename* all bound identifiers to entirely previously unused (known as *fresh*) names. Imagine that we give each identifier a numeric suffix to attain freshness. Then the original expression becomes

```
(lambda (f1)
  (lambda (x1)
    (f1 10)))
```

(Observe that we renamed `f` to `f1` in both the binding and bound locations.) Now let's do the same with the expression we're substituting:

```
(lambda (y1) (+ x y1))
```

Now let's substitute for `f1`:

```
(lambda (x1)
  ((lambda (y1) (+ x y1)) 10))
```

Why didn't we rename `x`? Because `x` may be a reference to a top-level binding, which should then also be renamed. This is simply another application of the consistent renaming principle. In the current setting, the distinction is irrelevant.

...and `x` is still free! *This* is a good form of substitution.

But one moment. What happens if we try the same example in our environment-based interpreter?

Do Now!

Try it out.

Observe that it works correctly: it reports an unbound identifier error. Environments automatically implement capture-free substitution!

Exercise

In what way does using an environment avoid the capture problem of substitution?

7.5 Sugaring Over Anonymity

Now let's get back to the idea of naming functions, which has evident value for program understanding. Observe that we *do* have a way of naming things: by passing them to functions, where they acquire a local name (that of the formal parameter). Anywhere within that function's body, we can refer to that entity using the formal parameter name.

Therefore, we can take a collection of function definitions and name them using other...functions. For instance, the Racket code

```
(define (double x) (+ x x))
(double 10)
```

could first be rewritten as the equivalent

```
(define double (lambda (x) (+ x x)))
(double 10)
```

We can of course just inline the definition of `double`, but to preserve the name, we could write this as:

```
((lambda (double)
   (double 10))
 (lambda (x) (+ x x)))
```

Indeed, this pattern—which we will pronounce as "left-left-lambda"—is a local naming mechanism. It is so useful that in Racket, it has its own special syntax:

```
(let ([double (lambda (x) (+ x x))])
  (double 10))
```

where `let` can be defined by desugaring as shown above.

Here's a more complex example:

```
(define (double x) (+ x x))
(define (quadruple x) (double (double x)))
(quadruple 10)
```

This could be rewritten as

```
(let ([double (lambda (x) (+ x x))])
  (let ([quadruple (lambda (x) (double (double x)))])
    (quadruple 10)))
```

which works just as we'd expect; but if we change the order, it no longer works—

```
(let ([quadruple (lambda (x) (double (double x)))])
  (let ([double (lambda (x) (+ x x))])
    (quadruple 10)))
```

—because `quadruple` can't "see" `double`. so we see that top-level binding is different from local binding: essentially, the top-level has an "infinite scope". This is the source of both its power and problems.

There is another, subtler, problem: it has to do with recursion. Consider the simplest infinite loop:

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

Let's convert it to `let`:

```
(let ([loop-forever (lambda (x) (loop-forever x))])
  (loop-forever 10))
```

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)
```