```
(define (get-fundef [n : symbol] [fds : (listof FunDefC)]) : FunDefC
  (cond
    [(empty? fds) (error 'get-fundef "reference to undefined function")]
    [(cons? fds) (cond
                   [(equal? n (fdC-name (first fds))) (first fds)]
                   [else (get-fundef n (rest fds))])]))
```

5.5 Oh Wait, There's More!

Earlier, we gave the following type to `subst`:

```
; subst : ExprC * symbol * ExprC -> ExprC
```

Sticking to surface syntax for brevity, suppose we apply `double` to `(+ 1 2)`. This would substitute `(+ 1 2)` for each `x`, resulting in the following expression—`(+ (+ 1 2) (+ 1 2))`—for interpretation. Is this necessarily what we want?

When you learned algebra in school, you may have been taught to do this differently: first reduce the argument to an answer (in this case, `3`), then substitute the answer for the parameter. This notion of substitution might have the following type instead:

```
; subst : number * symbol * ExprC -> ExprC
```

Careful now: we can't put raw numbers inside expressions, so we'd have to constantly wrap the number in an invocation of `numC`. Thus, it would make sense for `subst` to have a helper that it invokes after wrapping the first parameter. (In fact, our existing `subst` would be a perfectly good candidate: because it accepts any `ExprC` in the first parameter, it will certainly work just fine with a `numC`.)

Exercise

Modify your interpreter to substitute names with answers, not expressions.

We've actually stumbled on a profound distinction in programming languages. The act of evaluating arguments before substituting them in functions is called *eager* application, while that of deferring evaluation is called *lazy*—and has some variations. For now, we will actually prefer the eager semantics, because this is what most mainstream languages adopt. Later [REF], we will return to talking about the lazy application semantics and its implications.

In fact, we don't even have substitution quite right! The version of substitution we have doesn't scale past this language due to a subtle problem known as "name capture". Fixing substitution is complex, subtle, and an exciting intellectual endeavor, but it's not the direction I want to go in here. We'll instead sidestep this problem in this book. If you're interested, however, read about the *lambda calculus*, which provides the tools for defining substitution correctly.

6 From Substitution to Environments

Though we have a working definition of functions, you may feel a slight unease about it. When the interpreter sees an identifier, you might have had a sense that it needs to "look it up". Not only did it not look up anything, we defined its behavior to be an error! While absolutely correct, this is also a little surprising. More importantly, we write interpreters to *understand* and *explain* languages, and this implementation might strike you as not doing that, because it doesn't match our intuition.

There's another difficulty with using substitution, which is the number of times we traverse the source program. It would be nice to have to traverse only those parts of the program that are actually evaluated, and then, only when necessary. But substitution traverses everything—unvisited branches of conditionals, for instance—and forces the program to be traversed once for substitution and once again for interpretation.

Exercise

Does substitution have implications for the time complexity of evaluation?

There's yet another problem with substitution, which is that it is defined in terms of representations of the program source. Obviously, our interpreter has and needs access to the source, to interpret it. However, other implementations—such as compilers—have no need to store it for that purpose. It would be nice to employ a mechanism that is more portable across implementation strategies.

Compilers might store versions of or information about the source for other reasons, such as reporting runtime errors, and JITs may need it to re-compile on demand.

6.1 Introducing the Environment

The intuition that addresses the first concern is to have the interpreter "look up" an identifier in some sort of directory. The intuition that addresses the second concern is to *defer* the substitution. Fortunately, these converge nicely in a way that also addresses the third. The directory records the *intent to substitute*, without actually rewriting the program source; by recording the intent, rather than substituting immediately, we can defer substitution; and the resulting data structure, which is called an *environment*, avoids the need for source-to-source rewriting and maps nicely to low-level machine representations. Each name association in the environment is called a *binding*.

Observe carefully that what we are changing is the *implementation strategy* for the programming language, *not the language itself*. Therefore, none of our datatypes for representing programs should change, nor even should the answers that the interpreter provides. As a result, we should think of the previous interpreter as a "reference implementation" that the one we're about to write should match. Indeed, we should create a generator that creates lots of tests, runs them through both interpreters, and makes sure their answers are the same. Ideally, we should *prove* that the two interpreters behave the same, which is a good topic for advanced study.

One subtlety is in defining precisely what "the same" means, especially with regards to failure.

Let's first define our environment data structure. An environment is a list of pairs of names associated with...what?

Do Now!

A natural question to ask here might be what the environment maps names to. But a better, more fundamental, question is: How to determine the answer to the "natural" question?

Remember that our environment was created to defer substitutions. Therefore, the answer lies in substitution. We discussed earlier (section 5.5) that we want substitution to map names to answers, corresponding to an eager function application strategy. Therefore, the environment should map names to answers.

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

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

6.2 Interpreting with Environments

Now we can tackle the interpreter. One case is easy, but we should revisit all the others:

<*> **::=**

```
(define (interp [expr : ExprC] [env : Env] [fds : (listof FunDefC)]) : number
  (type-case ExprC expr
    [numC (n) n]
    <idC-case>
    <appC-case>
    <plusC/multC-case>))
```

The arithmetic operations are easiest. Recall that before, the interpreter recurred without performing any new substitutions. As a result, there are no new deferred substitutions to perform either, which means the environment does not change:

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

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

Now let's handle identifiers. Clearly, encountering an identifier is no longer an error: this was the very motivation for this change. Instead, we must look up its value in the directory:

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

```
[idC (n) (lookup n env)]
```

Do Now!

Implement `lookup`.

Finally, application. Observe that in the substitution interpreter, the only case that caused new substitutions to occur was application. Therefore, this should be the case that constructs bindings. Let's first extract the function definition, just as before:

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

```
[appC (f a) (local ([define fd (get-fundef f fds)])
                <appC-interp>)]
```

Previously, we substituted, then interpreted. Because we have no substitution step, we can proceed with interpretation, so long as we record the deferral of substitution.

<***appC-interp***> **::=**

```
(interp (fdC-body fd)
        <appC-interp-bind-in-env>
        fds)
```

That is, the set of function definitions remains unchanged; we're interpreting the body of the function, as before; but we have to do it in an environment that binds the formal parameter. Let's now define that binding process:

<appC-interp-bind-in-env-take-1> **::=**

```
(extend-env (bind (fdC-arg fd)
                  (interp a env fds))
            env)
```

the name being bound is the formal parameter (the same name that was substituted for, before). It is bound to the result of interpreting the argument (because we've decided on an eager application semantics). And finally, this extends the environment we already have. Type-checking this helps to make sure we got all the little pieces right.

Once we have a definition for `lookup`, we'd have a full interpreter. So here's one:

```
(define (lookup [for : symbol] [env : Env]) : number
  (cond
    [(empty? env) (error 'lookup "name not found")]
    [else (cond
            [(symbol=? for (bind-name (first env)))
             (bind-val (first env))]
            [else (lookup for (rest env))])]))
```

Observe that looking up a free identifier still produces an error, but it has moved from the interpreter—which is by itself unable to determine whether or not an identifier is free—to `lookup`, which determines this based on the content of the environment.

Now we have a full interpreter. You should of course test it make sure it works as you'd expect. For instance, these tests pass:

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

(test (interp (plusC (numC 10) (appC 'double (plusC (numC 1) (numC 2))))
              mt-env
              (list (fdC 'double 'x (plusC (idC 'x) (idC 'x)))))
      16)

(test (interp (plusC (numC 10) (appC 'quadruple (plusC (numC 1) (numC 2))))
              mt-env
              (list (fdC 'quadruple 'x (appC 'double (appC 'double (idC 'x))))
                    (fdC 'double 'x (plusC (idC 'x) (idC 'x)))))
      22)
```

So we're done, right?

Do Now!

Spot the bug.

6.3 Deferring Correctly

Here's another test:

```
(interp (appC 'f1 (numC 3))
                  mt-env
                  (list (fdC 'f1 'x (appC 'f2 (numC 4)))
                        (fdC 'f2 'y (plusC (idC 'x) (idC 'y)))))
```

In our interpreter, this evaluates to `7`. Should it?

Translated into Racket, this test corresponds to the following two definitions and expression:

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

(f1 3)
```

What should this produce? `(f1 3)` substitutes `x` with `3` in the body of `f1`, which then invokes `(f2 4)`. But notably, in `f2`, the identifier `x` is *not bound*! Sure enough, Racket will produce an error.

In fact, so will our substitution-based interpreter!

Why does the substitution process result in an error? It's because, when we replace the representation of `x` with the representation of `3` in the representation of `f1`, we do so in `f1` *only*. (Obviously: `x` is `f1`'s parameter; even if another function had a parameter named `x`, that's a *different* `x`.) Thus, when we get to evaluating the body of `f2`, its `x` hasn't been substituted, resulting in the error.

This "the representation of" is getting a little annoying, isn't it? Therefore, I'll stop saying that, but do make sure you understand why I had to say it. It's an important bit of pedantry.

What went wrong when we switched to environments? Watch carefully: this is subtle. We can focus on applications, because only they affect the environment. When we substituted the formal for the value of the actual, we did so by *extending the current environment*. In terms of our example, we asked the interpreter to substitute not only `f2`'s substitution in `f2`'s body, but also the current ones (those for the caller, `f1`), and indeed all past ones as well. That is, the environment only grows; it never shrinks.

Because we agreed that environments are only an alternate implementation strategy for substitution—and in particular, that the language's meaning should not change—we have to alter the interpreter. Concretely, we should not ask it to carry around all past deferred substitution requests, but instead make it start afresh for every new function, just as the substitution-based interpreter does. This is an easy change:

⟨***appC-interp-bind-in-env***⟩ **::=**

```
(extend-env (bind (fdC-arg fd)
                  (interp a env fds))
            mt-env)
```

Now we have truly reproduced the behavior of the substitution interpreter.

In case you're wondering how to write a test case that catches errors, look up `test/exn`.

6.4 Scope

The broken environment interpreter above implements what is known as *dynamic scope*. This means the environment accumulates bindings as the program executes. As a result, whether an identifier is even bound depends on the history of program execution. We should regard this unambiguously as a flaw of programming language design. It adversely affects all tools that read and process programs: compilers, IDEs, and humans.

In contrast, substitution—and environments, done correctly—give us *lexical scope* or *static scope*. "Lexical" in this context means "as determined from the source program", while "static" in computer science means "without running the program", so these are appealing to the same intuition. When we examine an identifier, we want to know two things: (1) Is it bound? (2) If so, where? By "where" we mean: if there are multiple bindings for the same name, which one governs this identifier? Put differently, which one's substitution will give a value to this identifier? In general, these questions cannot be answered statically in a dynamically-scoped language: so your IDE, for instance, cannot overlay arrows to show you this information (as DrRacket does). Thus, even though the rules of scope become more complex as the space of names becomes richer (e.g., objects, threads, etc.), we should always strive to preserve the spirit of static scoping.

A different way to think about it is that in a dynamically-scoped language, the answer to these questions is the same for *all* identifiers, and it simply refers to the dynamic environment. In other words, it provides no useful information.

6.4.1 How Bad Is It?

You might look at our running example and wonder whether we're creating a tempest in a teapot. In return, you should consider two situations:

1. To understand the binding structure of your program, you may need to look at *the whole program*. No matter how much you've decomposed your program into small, understandable fragments, it doesn't matter if you have a free identifier anywhere.

2. Understanding the binding structure is not only a function of the *size* of the program but also of the complexity of its control flow. Imagine an interactive program with numerous callbacks; you'd have to track through every one of them, too, to know which binding governs an identifier.

Need a little more of a nudge? Let's replace the expression of our example program with this one:

```
(if (moon-visible?)
  (f1 10)
  (f2 10))
```

Suppose `moon-visible?` is a function that presumably evaluates to false on new-moon nights, and true at other times. Then, this program will evaluate to an answer except on new-moon nights, when it will fail with an unbound identifier error.

Exercise

What happens on cloudy nights?

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]