Programming Languages: Application and Interpretation

Version Second Edition

Shriram Krishnamurthi

April 14, 2017

Contents

1 Introduction

1.1 Our Philosophy

Please watch the video on YouTube. Someday there will be a textual description here instead.

1.2 The Structure of This Book

Unlike some other textbooks, this one does not follow a top-down narrative. Rather it has the flow of a conversation, with backtracking. We will often build up programs incrementally, just as a pair of programmers would. We will include mistakes, not because I don't know the answer, but because *this is the best way for you to learn*. Including mistakes makes it impossible for you to read passively: you must instead engage with the material, because you can never be sure of the veracity of what you're reading.

At the end, you'll always get to the right answer. However, this non-linear path is more frustrating in the short term (you will often be tempted to say, "Just tell me the answer, already!"), and it makes the book a poor reference guide (you can't open up to a random page and be sure what it says is correct). However, that feeling of frustration is the sensation of learning. I don't know of a way around it.

At various points you will encounter this:

Exercise

This is an exercise. Do try it.

This is a traditional textbook exercise. It's something you need to do on your own. If you're using this book as part of a course, this may very well have been assigned as homework. In contrast, you will also find exercise-like questions that look like this:

Do Now!

There's an activity here! Do you see it?

When you get to one of these, **stop**. Read, think, and formulate an answer before you proceed. You must do this because this is actually an *exercise*, but the answer is already in the book—most often in the text immediately following (i.e., in the part you're reading right now)—or is something you can determine for yourself by running a program. If you just read on, you'll see the answer without having thought about it (or not see it at all, if the instructions are to run a program), so you will get to neither (a) test your knowledge, nor (b) improve your intuitions. In other words, these are additional, explicit attempts to encourage active learning. Ultimately, however, I can only encourage it; it's up to you to practice it.

1.3 The Language of This Book

The main programming language used in this book is Racket. Like with all operating systems, however, Racket actually supports a host of programming languages, so you

must tell Racket *which* language you're programming in. You inform the Unix shell by writing a line like
`#!/bin/sh`
at the top of a script; you inform the browser by writing, say,
`<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" ...>`
Similarly, Racket asks that you declare which language you will be using. Racket languages can have the same parenthetical syntax as Racket but with a different semantics; the same semantics but a different syntax; or different syntax and semantics. Thus every Racket program begins with `#lang` followed by the name of some language: by default, it's Racket (written as `racket`). In this book we'll almost always use the language

```
plai-typed
```

. When we deviate we'll say so explicitly, so unless indicated otherwise, put
`#lang plai-typed`
at the top of every file (and assume I've done the same).

In DrRacket v. 5.3, go to Language, then Choose Language, and select "Use the language declared in the source".

The *Typed PLAI* language differs from traditional Racket most importantly by being statically typed. It also gives you some useful new constructs: `define-type`, `type-case`, and `test`. Here's an example of each in use. We can introduce new datatypes:

```
(define-type MisspelledAnimal
  [caml (humps : number)]
  [yacc (height : number)])
```

There are additional commands for controlling the output of testing, for instance. Be sure to read the documentation for the language In DrRacket v. 5.3, go to Help, then Help Desk, and in the Help Desk search bar, type "plai-typed".

You can roughly think of this as analogous to the following in Java: an abstract class `MisspelledAnimal` and two concrete sub-classes `caml` and `yacc`, each of which has one numeric constructor argument named `humps` and `height`, respectively.

In this language, we construct instances as follows:

```
(caml 2)
(yacc 1.9)
```

As the name suggests, `define-type` creates a type of the given name. We can use this when, for instance, binding the above instances to names:

```
(define ma1 : MisspelledAnimal (caml 2))
(define ma2 : MisspelledAnimal (yacc 1.9))
```

In fact you don't need these particular type declarations, because Typed PLAI will infer types for you here and in many other cases. Thus you could just as well have written

```
(define ma1 (caml 2))
(define ma2 (yacc 1.9))
```

but we prefer to write explicit type declarations as a matter of both discipline and comprehensibility when we return to programs later.

The type names can even be used recursively, as we will see repeatedly in this book (for instance, section 2.4).

The language provides a pattern-matcher for use when writing expressions, such as a function's body:

```
(define (good? [ma : MisspelledAnimal]) : boolean
  (type-case MisspelledAnimal ma
    [caml (humps) (>= humps 2)]
    [yacc (height) (> height 2.1)]))
```

In the expression `(>= humps 2)`, for instance, `humps` is the name given to whatever value was given as the argument to the constructor `caml`.

Finally, you should write test cases, ideally before you've defined your function, but also afterwards to protect against accidental changes:

```
(test (good? ma1) #t)
(test (good? ma2) #f)
```

When you run the above program, the language will give you verbose output telling you both tests passed. Read the documentation to learn how to suppress most of these messages.

Here's something important that is obscured above. We've used the same name, `humps` (and `height`), in *both* the datatype definition and in the fields of the pattern-match. This is absolutely unnecessary because the two are related by *position*, not name. Thus, we could have as well written the function as

```
(define (good? [ma : MisspelledAnimal]) : boolean
  (type-case MisspelledAnimal ma
    [caml (h) (>= h 2)]
    [yacc (h) (> h 2.1)]))
```

Because each `h` is only visible in the case branch in which it is introduced, the two `h`s do not in fact clash. You can therefore use convention and readability to dictate your choices. In general, it makes sense to provide a long and descriptive name when defining the datatype (because you probably won't use that name again), but shorter names in the `type-case` because you're likely to use use those names one or more times.

I did just say you're unlikely to use the field descriptors introduced in the datatype definition, but you can. The language provides selectors to extract fields without the need for pattern-matching: e.g., `caml-humps`. Sometimes, it's much easier to use the selector directly rather than go through the pattern-matcher. It often isn't, as when defining `good?` above, but just to be clear, let's write it without pattern-matching:

```
(define (good? [ma : MisspelledAnimal]) : boolean
  (cond
    [(caml? ma) (>= (caml-humps ma) 2)]
    [(yacc? ma) (> (yacc-height ma) 2.1)]))
```

Do Now!

What happens if you mis-apply functions to the wrong kinds of values? For instance, what if you give the `caml` constructor a string? What if you send a number into each version of `good?` above?

2 Everything (We Will Say) About Parsing

Parsing is the act of turning an input character stream into a more structured, internal representation. A common internal representation is as a tree, which programs can recursively process. For instance, given the stream

```
23 + 5 - 6
```

we might want a tree representing addition whose left node represents the number `23` and whose right node represents subtraction of `6` from `5`. A *parser* is responsible for performing this transformation.

Parsing is a large, complex problem that is far from solved due to the difficulties of ambiguity. For instance, an alternate parse tree for the above input expression might put subtraction at the top and addition below it. We might also want to consider whether this addition operation is commutative and hence whether the order of arguments can be switched. Everything only gets much, much worse when we get to full-fledged programming languages (to say nothing of natural languages).

2.1 A Lightweight, Built-In First Half of a Parser

These problems make parsing a worthy topic in its own right, and entire books, tools, and courses are devoted to it. However, from our perspective parsing is mostly a distraction, because we want to study the parts of programming languages that are *not* parsing. We will therefore exploit a handy feature of Racket to manage the transformation of input streams into trees: `read`. `read` is tied to the parenthetical form of the language, in that it parses fully (and hence unambiguously) parenthesized terms into a built-in tree form. For instance, running `(read)` on the parenthesized form of the above input—

```
(+ 23 (- 5 6))
```

—will produce a list, whose first element is the symbol `'+`, second element is the number `23`, and third element is a list; this list's first element is the symbol `'-`, second element is the number `5`, and third element is the number `6`.

2.2 A Convenient Shortcut

As you know you need to test your programs extensively, which is hard to do when you must manually type terms in over and over again. Fortunately, as you might expect, the parenthetical syntax is integrated deeply into Racket through the mechanism of *quotation*. That is, `'<expr>`—which you saw a moment ago in the above example—acts as if you had run `(read)` and typed `<expr>` at the prompt (and, of course, evaluates to the value the `(read)` would have).

2.3 Types for Parsing

Actually, I've lied a little. I said that `(read)`—or equivalently, using quotation—will produce a list, etc. That's true in regular Racket, but in Typed PLAI, the type it returns a distinct type called an *s-expression*, written in Typed PLAI as `s-expression`:

```
> (read)
- s-expression
[type in (+ 23 (- 5 6))]
'(+ 23 (- 5 6))
```

Racket has a very rich language of s-expressions (it even has notation to represent cyclic structures), but we will use only the simple fragment of it.

In the typed language, an s-expression is treated distinctly from the other types, such as numbers and lists. Underneath, an s-expression is a large recursive datatype that consists of all the base printable values—numbers, strings, symbols, and so on—and printable collections (lists, vectors, etc.) of s-expressions. As a result, base types like numbers, symbols, and strings are *both* their own type and an instance of s-expression. Typing such data can be fairly problematic, as we will discuss later [REF].

Typed PLAI takes a simple approach. When written on their own, values like numbers are of those respective types. But when written inside a complex s-expression—in particular, as created by `read` or quotation—they have type `s-expression`. You have to then *cast* them to their native types. For instance:

```
> '+
- symbol
'+
> (define l '(+ 1 2))
> l
- s-expression
'(+ 1 2)
> (first l)
. typecheck failed: (listof '_a) vs s-expression in:
  first
  (quote (+ 1 2))
  l
  first
> (define f (first (s-exp->list l)))
> f
- s-expression
'+
```

This is similar to the casting that a Java programmer would have to insert. We will study casting itself later [REF].

Observe that the first element of the list is still not treated by the type checker as a symbol: a list-shaped s-expression is a list of *s-expressions*. Thus,

```
> (symbol->string f)
. typecheck failed: symbol vs s-expression in:
  symbol->string
  f
```

```
  symbol->string
  f
  first
  (first (s-exp->list l))
  s-exp->list
```
whereas again, casting does the trick:
```
> (symbol->string (s-exp->symbol f))
- string
"+"
```
The need to cast s-expressions is a bit of a nuisance, but some complexity is unavoidable because of what we're trying to accomplish: to convert an *untyped* input stream into a *typed* output stream through robustly *typed* means. Somehow we have to make explicit our assumptions about that input stream.

Fortunately we will use s-expressions only in our parser, and our goal is to *get away from parsing as quickly as possible*! Indeed, if anything this should be inducement to get away even quicker.

2.4 Completing the Parser

In principle, we can think of `read` as a complete parser. However, its output is generic: it represents the token structure without offering any comment on its intent. We would instead prefer to have a representation that tells us something about the *intended meaning* of the terms in our language, just as we wrote at the very beginning: "representing addition", "represents a number", and so on.

To do this, we must first introduce a datatype that captures this representation. We will separately discuss (section 3.1) how and why we obtained this datatype, but for now let's say it's given to us:

```
(define-type ArithC
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])
```

We now need a function that will convert s-expressions into instances of this datatype. This is the other half of our parser:

```
(define (parse [s : s-expression]) : ArithC
  (cond
    [(s-exp-number? s) (numC (s-exp->number s))]
    [(s-exp-list? s)
     (let ([sl (s-exp->list s)])
       (case (s-exp->symbol (first sl))
         [(+) (plusC (parse (second sl)) (parse (third sl)))]
         [(*) (multC (parse (second sl)) (parse (third sl)))]
         [else (error 'parse "invalid list input")]))]
    [else (error 'parse "invalid input")]))
```

Thus:

```
> (parse '(+ (* 1 2) (+ 2 3)))
- ArithC
(plusC
 (multC (numC 1) (numC 2))
 (plusC (numC 2) (numC 3)))
```

Congratulations! You have just completed your first *representation of a program*. From now on we can focus entirely on programs represented as recursive trees, ignoring the vagaries of surface syntax and how to get them into the tree form. We're finally ready to start studying programming languages!

Exercise

What happens if you forget to quote the argument to the parser? Why?

2.5 Coda

Racket's syntax, which it inherits from Scheme and Lisp, is controversial. Observe, however, something deeply valuable that we get from it. While parsing traditional languages can be very complex, parsing this syntax is virtually trivial. Given a sequence of tokens corresponding to the input, it is absolutely straightforward to turn parenthesized sequences into s-expressions; it is equally straightforward (as we see above) to turn s-expressions into proper syntax trees. I like to call such two-level languages *bicameral*, in loose analogy to government legislative houses: the lower-level does rudimentary well-formedness checking, while the upper-level does deeper validity checking. (We haven't done any of the latter yet, but we will [REF].)

The virtues of this syntax are thus manifold. The amount of code it requires is small, and can easily be embedded in many contexts. By integrating the syntax into the language, it becomes easy for programs to manipulate representations of programs (as we will see more of in [REF]). It's therefore no surprise that even though many Lisp-based syntaxes have had wildly different semantics, they all share this syntactic legacy.

Of course, we could just use XML instead. That would be much better. Or JSON. Because that wouldn't be anything like an s-expression at all.

3 A First Look at Interpretation

Now that we have a representation of programs, there are many ways in which we might want to manipulate them. We might want to display a program in an attractive way ("pretty-print"), convert into code in some other format ("compilation"), ask whether it obeys certain properties ("verification"), and so on. For now, we're going to focus on asking what value it corresponds to ("evaluation"—the reduction of programs to *values*).

Let's write an evaluator, in the form of an *interpreter*, for our arithmetic language. We choose arithmetic first for three reasons: (a) you already know how it works, so we can focus on the mechanics of writing evaluators; (b) it's contained in every language

we will encounter later, so we can build upwards and outwards from it; and (c) it's at once both small and big enough to illustrate many points we'd like to get across.

3.1 Representing Arithmetic

Let's first agree on how we will represent arithmetic expressions. Let's say we want to support only two operations—addition and multiplication—in addition to primitive numbers. We need to represent arithmetic *expressions*. What are the rules that govern nesting of arithmetic expressions? We're actually free to nest any expression inside another.

Do Now!

> Why did we not include division? What impact does it have on the remarks above?

We've ignored division because it forces us into a discussion of what expressions we might consider legal: clearly the representation of `1/2` ought to be legal; the representation of `1/0` is much more debatable; and that of `1/(1-1)` seems even more controversial. We'd like to sidestep this controversy for now and return to it later [REF].

Thus, we want a representation for numbers and arbitrarily nestable addition and multiplication. Here's one we can use:

```
(define-type ArithC
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])
```

3.2 Writing an Interpreter

Now let's write an interpreter for this arithmetic language. First, we should think about what its type is. It clearly consumes a `ArithC` value. What does it produce? Well, an interpreter evaluates—and what kind of value might arithmetic expressions reduce to? Numbers, of course. So the interpreter is going to be a function from arithmetic expressions to numbers.

Exercise

> Write your test cases for the interpreter.

Because we have a recursive datatype, it is natural to structure our interpreter as a recursive function over it. Here's a first template:

Templates are explained in great detail in *How to Design Programs*.

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) ...]
    [multC (l r) ...]))
```

You're probably tempted to jump straight to code, which you can:

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) (+ l r)]
    [multC (l r) (* l r)]))
```

Do Now!

Do you spot the errors?

Instead, let's expand the template out a step:

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) ... (interp l) ... (interp r) ...]
    [multC (l r) ... (interp l) ... (interp r) ...]))
```

and now we can fill in the blanks:

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) (+ (interp l) (interp r))]
    [multC (l r) (* (interp l) (interp r))]))
```

Later on [REF], we're going to wish we had returned a more complex datatype than just numbers. But for now, this will do.

Congratulations: you've written your first interpreter! I know, it's very nearly an anticlimax. But they'll get harder—much harder—pretty soon, I promise.

3.3 Did You Notice?

I just slipped something by you:

Do Now!

What is the "meaning" of addition and multiplication in this new language?

That's a pretty abstract question, isn't it. Let's make it concrete. There are many kinds of addition in computer science:

- First of all, there's many different kinds of *numbers*: fixed-width (e.g., 32-bit) integers, signed fixed-width (e.g., 31-bits plus a sign-bit) integers, arbitrary precision integers; in some languages, rationals; various formats of fixed- and floating-point numbers; in some languages, complex numbers; and so on. After the numbers have been chosen, addition may support only some combinations of them.

- In addition, some languages permit the addition of datatypes such as matrices.
- Furthermore, many languages support "addition" of strings (we use scare-quotes because we don't really mean the mathematical concept of addition, but rather the operation performed by an operator with the syntax +). In some languages this always means concatenation; in some others, it can result in numeric results (or numbers stored in strings).

These are all different meanings for addition. *Semantics* is the mapping of syntax (e.g., +) to meaning (e.g., some or all of the above).

This brings us to our first game of:

Which of these is the same?

- `1 + 2`
- `1 + 2`
- `'1' + '2'`
- `'1' + '2'`

Now return to the question above. What semantics do we have? We've adopted whatever semantics Racket provides, because we map + to Racket's +. In fact that's not even quite true: Racket may, for all we know, also enable + to apply to strings, so we've chosen the restriction of Racket's semantics to numbers (though in fact Racket's + doesn't tolerate strings).

If we wanted a different semantics, we'd have to implement it explicitly.

Exercise

> What all would you have to change so that the number had signed- 32-bit arithmetic?

In general, we have to be careful about too readily borrowing from the host language. We'll return to this topic later [REF].

3.4 Growing the Language

We've picked a very restricted first language, so there are many ways we can grow it. Some, such as representing data structures and functions, will clearly force us to add new features to the interpreter itself (assuming we don't want to use Gödel numbering). Others, such as adding more of arithmetic itself, can be done without disturbing the core language and hence its interpreter. We'll examine this next (section 4).

4 A First Taste of Desugaring

We've begun with a very spartan arithmetic language. Let's look at how we might extend it with more arithmetic operations that can nevertheless be expressed in terms of existing ones. We'll add just two, because these will suffice to illustrate the point.

4.1 Extension: Binary Subtraction

First, we'll add subtraction. Because our language already has numbers, addition, and multiplication, it's easy to define subtraction: $a - b = a + -1 \times b$.

Okay, that was easy! But now we should turn this into concrete code. To do so, we face a decision: where does this new subtraction operator reside? It is tempting, and perhaps seems natural, to just add one more rule to our existing `ArithC` datatype.

Do Now!

What are the negative consequences of modifying `ArithC`?

This creates a few problems. The first, obvious, one is that we now have to modify all programs that process `ArithC`. So far that's only our interpreter, which is pretty simple, but in a more complex implementation, that could already be a concern. Second, we were trying to add new constructs that we can define in terms of existing ones; it feels slightly self-defeating to do this in a way that isn't modular. Third, and most subtly, there's something *conceptually* wrong about modifying `ArithC`. That's because `ArithC` represents our *core* language. In contrast, subtraction and other additions represent our user-facing, surface language. It's wise to record conceptually different ideas in distinct datatypes, rather than shoehorn them into one. The separation can look a little unwieldy sometimes, but it makes the program much easier for future developers to read and maintain. Besides, for different purposes you might want to layer on different extensions, and separating the core from the surface enables that.

Therefore, we'll define a new datatype to reflect our intended surface syntax terms:

```
(define-type ArithS
  [numS (n : number)]
  [plusS (l : ArithS) (r : ArithS)]
  [bminusS (l : ArithS) (r : ArithS)]
  [multS (l : ArithS) (r : ArithS)])
```

This looks almost exactly like `ArithC`, other than the added case, which follows the familiar recursive pattern.

Given this datatype, we should do two things. First, we should modify our parser to also parse `-` expressions, and always construct `ArithS` terms (rather than any `ArithC` ones). Second, we should implement a `desugar` function that translates `ArithS` values into `ArithC` ones.

Let's write the obvious part of `desugar`:

<***desugar***> **::=**

```
(define (desugar [as : ArithS]) : ArithC
  (type-case ArithS as
    [numS (n) (numC n)]
    [plusS (l r) (plusC (desugar l)
                        (desugar r))]
    [multS (l r) (multC (desugar l)
                        (desugar r))]
    <bminusS-case>))
```

Now let's convert the mathematical description of subtraction above into code:

<bminusS-case> **::=**

```
[bminusS (l r) (plusC (desugar l)
                      (multC (numC -1) (desugar r)))]
```

Do Now!

It's a common mistake to forget the recursive calls to `desugar` on `l` and `r`. What happens when you forget them? Try for yourself and see.

4.2 Extension: Unary Negation

Now let's consider another extension, which is a little more interesting: unary negation. This forces you to do a little more work in the parser because, depending on your surface syntax, you may need to look ahead to determine whether you're in the unary or binary case. But that's not even the interesting part!

There are many ways we can desugar unary negation. We can define it naturally as $-b = 0 - b$, or we could abstract over the desugaring of binary subtraction with this expansion: $-b = 0 + -1 \times b$.

Do Now!

Which one do you prefer? Why?

It's tempting to pick the first expansion, because it's much simpler. Imagine we've extended the `ArithS` datatype with a representation of unary negation:

```
[uminusS (e : ArithS)]
```

Now the implementation in `desugar` is straightforward:

```
[uminusS (e) (desugar (bminusS (numS 0) e))]
```

Let's make sure the types match up. Observe that `e` is a `ArithS` term, so it is valid to use as an argument to `bminusS`, and the entire term can legally be passed to `desugar`. It is therefore important to *not* desugar `e` but rather embed it directly in the generated term. This embedding of an input term in another one and recursively calling desugar is a common pattern in desugaring tools; it is called a *macro* (specifically, the "macro" here is this definition of `uminusS`).

However, there are two problems with the definition above:

1. The first is that the recursion is *generative*, which forces us to take extra care. We might be tempted to fix this by using a different rewrite:

   ```
   [uminusS (e) (bminusS (numS 0) (desugar e))]
   ```

   which does indeed eliminate the generativity.

   Do Now!

If you haven't heard of generative recursion before, read the section on it in *How to Design Programs*. Essentially, in generative recursion the sub-problem is a computed function of the input, rather than a structural piece of it. This is an especially simple case of generative recursion, because the "function" is simple: it's just the `bminusS` constructor.

Unfortunately, this desugaring transformation won't work at all! Do you see why? If you don't, try to run it.

2. The second is that we are implicitly depending on exactly what `bminusS` means; if its meaning changes, so will that of `uminusS`, even if we don't want it to. In contrast, defining a functional abstraction that consumes two terms and generates one representing the addition of the first to -1 times the second, and using this to define the desugaring of both `uminusS` and `bminusS`, is a little more fault-tolerant.

   You might say that the meaning of subtraction is never going to change, so why bother? Yes and no. Yes, it's *meaning* is unlikely to change; but no, its *implementation* might. For instance, the developer may decide to log all uses of binary subtraction. In the macro expansion, all uses of unary negation would also get logged, but they would not in the second expansion.

Fortunately, in this particular case we have a much simpler option, which is to define $-b = -1 \times b$. This expansion works with the primitives we have, and follows structural recursion. The reason we took the above detour, however, is to alert you to these problems, and warn that you might not always be so fortunate.

5 Adding Functions to the Language

Let's start turning this into a real programming language. We could add intermediate features such as conditionals, but to do almost anything interesting we're going to need functions or their moral equivalent, so let's get to it.

Exercise

Add conditionals to your language. You can either add boolean datatypes or, if you want to do something quicker, add a conditional that treats `0` as false and everything else as true.

What are the important test cases you should write?

Imagine, therefore, that we're modeling a system like DrRacket. The developer defines functions in the definitions window, and uses them in the interactions window. For now, let's assume all definitions go in the definitions window only (we'll relax this soon [REF]), and all expressions in the interactions window only. Thus, running a program simply loads definitions. Because our interpreter corresponds to the interactions window prompt, we'll therefore assume it is supplied with a set of definitions.

A *set* of definitions suggests no ordering, which means, presumably, any definition can refer to any other. That's what I intend here, but when you are designing your own language, be sure to think about this.

5.1 Defining Data Representations

To keep things simple, let's just consider functions of one argument. Here are some Racket examples:

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

```
(define (quadruple x) (double (double x)))

(define (const5 _) 5)
```

Exercise

When a function has multiple arguments, what simple but important criterion governs the names of those arguments?

What are the parts of a function definition? It has a name (above, `double`, `quadruple`, and `const5`), which we'll represent as a symbol (`'double`, etc.); its *formal parameter* or *argument* has a name (e.g., `x`), which too we can model as a symbol (`'x`); and it has a body. We'll determine the body's representation in stages, but let's start to lay out a datatype for function definitions:

<***fundef***> **::=**

```
(define-type FunDefC
  [fdC (name : symbol) (arg : symbol) (body : ExprC)])
```

What is the body? Clearly, it has the form of an arithmetic expression, and sometimes it can even be represented using the existing `ArithC` language: for instance, the body of `const5` can be represented as `(numC 5)`. But representing the body of `double` requires something more: not just addition (which we have), but also "`x`". You are probably used to calling this a *variable*, but we will *not* use that term for now. Instead, we will call it an *identifier*.

I promise we'll return to this issue of nomenclature later [REF].

Do Now!

Anything else?

Finally, let's look at the body of `quadruple`. It has yet another new construct: a function *application*. Be very careful to distinguish between a function *definition*, which describes what the function is, and an *application*, which uses it. These are uses. The *argument* (or *actual parameter*) in the inner application of `double` is `x`; the argument in the outer application is `(double x)`. Thus, the argument can be any complex expression.

Let's commit all this to a crisp datatype. Clearly we're extending what we had before (because we still want all of arithmetic). We'll give a new name to our datatype to signify that it's growing up:

<***exprC***> **::=**

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

Identifiers are closely related to formal parameters. When we apply a function by giving it a value for its parameter, we are in effect asking it to replace all instances

of that formal parameter in the body—i.e., the identifiers with the same name as the formal parameter—with that value. To simplify this process of search-and-replace, we might as well use the same datatype to represent both. We've already chosen symbols to represent formal parameters, so:

Observe that we are being coy about a few issues: what kind of "value" [REF] and when to replace [REF].

<***idC-def***> **::=**

```
[idC (s : symbol)]
```

Finally, applications. They have two parts: the function's name, and its argument. We've already agreed that the argument can be any full-fledged expression (including identifiers and other applications). As for the function name, it again makes sense to use the same datatype as we did when giving the function its name in a function definition. Thus:

<***app-def***> **::=**

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

identifying which function to apply, and providing its argument.

Using these definitions, it's instructive to write out the representations of the examples we defined above:

- `(fdC 'double 'x (plusC (idC 'x) (idC 'x)))`
- `(fdC 'quadruple 'x (appC 'double (appC 'double (idC 'x))))`
- `(fdC 'const5 '_ (numC 5))`

We also need to choose a representation for a set of function definitions. It's convenient to represent these by a list.

Look out! Did you notice that we spoke of a *set* of function definitions, but chose a *list* representation? That means we're using an ordered collection of data to represent an unordered entity. At the very least, then, when testing, we should use any and all permutations of definitions to ensure we haven't subtly built in a dependence on the order.

5.2 Growing the Interpreter

Now we're ready to tackle the interpreter proper. First, let's remind ourselves of what it needs to consume. Previously, it consumed only an expression to evaluate. Now it also needs to take a list of function definitions:

<***interp***> **::=**

```
(define (interp [e : ExprC] [fds : (listof FunDefC)]) : number
  <interp-body>)
```

Let's revisit our old interpreter (section 3). In the case of numbers, clearly we still return the number as the answer. In the addition and multiplication case, we still need to recur (because the sub-expressions might be complex), but which set of function definitions do we use? Because the act of evaluating an expression neither adds nor removes function *definitions*, the set of definitions remains the same, and should just be passed along unchanged in the recursive calls.

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

```
(type-case ExprC e
  [numC (n) n]
  <idC-interp-case>
  <appC-interp-case>
  [plusC (l r) (+ (interp l fds) (interp r fds))]
  [multC (l r) (* (interp l fds) (interp r fds))])
```

Now let's tackle application. First we have to look up the function definition, for which we'll assume we have a helper function of this type available:

```
; get-fundef : symbol * (listof FunDefC) -> FunDefC
```

Assuming we find a function of the given name, we need to evaluate its body. However, remember what we said about identifiers and parameters? We must "search-and-replace", a process you have seen before in school algebra called *substitution*. This is sufficiently important that we should talk first about substitution before returning to the interpreter (section 5.4).

5.3 Substitution

Substitution is the act of replacing a name (in this case, that of the formal parameter) in an expression (in this case, the body of the function) with another expression (in this case, the actual parameter). Let's define its type:

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

It helps to also give its parameters informative names:
<***subst***> **::=**

```
(define (subst [what : ExprC] [for : symbol] [in : ExprC]) : ExprC
  <subst-body>)
```

The first argument is what we want to replace the name with; the second is for what name we want to perform substitution; and the third is in which expression we want to do it.

Do Now!

> Suppose we want to substitute `3` for the identifier `x` in the bodies of the three example functions above. What should it produce?

In `double`, this should produce `(+ 3 3)`; in `quadruple`, it should produce `(double (double 3))`; and in `const5`, it should produce `5` (i.e., no substitution happens because there are no instances of `x` in the body).

These examples already tell us what to do in almost all the cases. Given a number, there's nothing to substitute. If it's an identifier, we haven't seen an example with a *different* identifier but you've guessed what should happen: it stays unchanged. In the other cases, descend into the sub-expressions, performing substitution.

A common mistake is to assume that the result of substituting, e.g., `3` for `x` in `double` is `(define (double x) (+ 3 3))`. This is incorrect. We only substitute *at the point when we apply the function*, at which point the function's invocation is replaced by its body. The header enables us to find the function and ascertain the name of its parameter; but

Before we turn this into code, there's an important case to consider. Suppose the name we are substituting happens to be the name of a function. Then what should happen?

Do Now!

What, indeed, should happen?

There are many ways to approach this question. One is from a design perspective: function names live in their own "world", distinct from ordinary program identifiers. Some languages (such as C and Common Lisp, in slightly different ways) take this perspective, and partition identifiers into different *namespaces* depending on how they are used. In other languages, there is no such distinction; indeed, we will examine such languages soon [REF].

For now, we will take a pragmatic viewpoint. Because expressions evaluate to numbers, that means a function name could turn into a number. However, numbers cannot name functions, only symbols can. Therefore, it makes no sense to substitute in that position, and we should leave the function name unmolested irrespective of its relationship to the variable being substituted. (Thus, a function could have a parameter named `x` as well as refer to another *function* called `x`, and these would be kept distinct.)

Now we've made all our decisions, and we can provide the body code:

<***subst-body***> **::=**

```
(type-case ExprC in
  [numC (n) in]
  [idC (s) (cond
             [(symbol=? s for) what]
             [else in])]
  [appC (f a) (appC f (subst what for a))]
  [plusC (l r) (plusC (subst what for l)
                      (subst what for r))]
  [multC (l r) (multC (subst what for l)
                      (subst what for r))])
```

Exercise

Observe that, whereas in the `numC` case the interpreter returned `n`, substitution returns `in` (i.e., the original expression, equivalent at that point to writing `(numC n)`. Why?

5.4 The Interpreter, Resumed

Phew! Now that we've completed the definition of substitution (or so we think), let's complete the interpreter. Substitution was a heavyweight step, but it also does much of the work involved in applying a function. It is tempting to write

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

```
[appC (f a) (local ([define fd (get-fundef f fds)])
              (subst a
```

```
                    (fdC-arg fd)
                    (fdC-body fd)))]
```

Tempting, but wrong.

Do Now!

Do you see why?

Reason from the types. What does the interpreter return? Numbers. What does substitution return? Oh, that's right, expressions! For instance, when we substituted in the body of `double`, we got back the representation of `(+ 5 5)`. This is not a valid answer for the interpreter. Instead, it must be reduced to an answer. That, of course, is precisely what the interpreter does:

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

```
[appC (f a) (local ([define fd (get-fundef f fds)])
              (interp (subst a
                             (fdC-arg fd)
                             (fdC-body fd))
                      fds))]
```

Okay, that leaves only one case: identifiers. What could possibly be complicated about them? They should be just about as simple as numbers! And yet we've put them off to the very end, suggesting something subtle or complex is afoot.

Do Now!

Work through some examples to understand what the interpreter should do in the identifier case.

Let's suppose we had defined `double` as follows:

```
(define (double x) (+ x y))
```

When we substitute `5` for `x`, this produces the expression `(+ 5 y)`. So far so good, but what is left to substitute `y`? As a matter of fact, it should be clear from the very outset that this definition of `double` is *erroneous*. The identifier `y` is said to be *free*, an adjective that in this setting has negative connotations.

In other words, the interpreter should never confront an identifier. All identifiers ought to be parameters that have already been substituted (known as *bound* identifiers—here, a positive connotation) before the interpreter ever sees them. As a result, there is only one possible response given an identifier:

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

```
[idC (_) (error 'interp "shouldn't get here")]
```

And that's it!

Finally, to complete our interpreter, we should define `get-fundef`:

```
(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]

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)
```

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 desugaring. 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 desugaring 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 subterm 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 `refs`). 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:

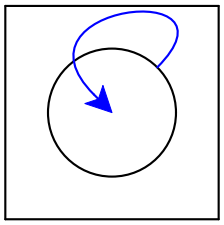
1. The ability to create compound structures (such as nodes that have references to children).

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:



Let's try to define this in Racket. Here's one attempt:

```
(let ([b b])
  b)
```

But this doesn't work: `b` on the right-hand side of the `let` isn't bound. It's easy to see if we desugar it:

```
((lambda (b)
   b)
  b)
```

and, for clarity, we can rename the `b` in the function:

```
((lambda (x)
   x)
 b)
```

Now it's patently clear that `b` is unbound.

Absent some magical Racket construct we haven't yet seen, it becomes clear that we can't create a cyclic datum in one shot. Instead, we need to first create a "place" for the datum, then refer to that place within itself. The use of "then"—i.e., the introduction of time—should suggest a mutation operation. Indeed, let's try it with boxes.

That construct would be `shared`, but virtually no other language has this notational mechanism, so we won't dwell on it here. In fact, what we are studying is the main idea behind how `shared` actually works.

Our plan is as follows. First, we want to create a box and bind it to some identifier, say `b`. Now, we want to mutate the content of the box. What do we want it to contain? A reference to itself. How does it obtain that reference? By using the name, `b`, that is already bound to it. In this way, the mutation creates the cycle:

```
(let ([b (box 'dummy)])
  (begin
    (set-box! b b)
    b))
```

Note that this program will *not* run in Typed PLAI as written. We'll return to typing such programs later [REF]. For now, run it in the untyped (`#lang plai`) language.

When the above program is Run, Racket prints this as: `#0='#&#0#`. This notation is in fact precisely what we want. Recall that `#&` is how Racket prints boxes. The `#0=` (and similarly for other numbers) is how Racket names pieces of cyclic data. Thus, Racket is saying, "`#0` is bound to a box whose content is `#0#`, i.e., whatever is bound to `#0`, i.e., itself".

Exercise

Run the equivalent program through your interpreter for boxes and make sure it produces a *cyclic* value. How do you check this?

The idea above generalizes to other datatypes. In this same way we can also produce cyclic lists, graphs, and so on. The central idea is this two-step process: first name an vacant placeholder; then mutate the placeholder so its content is itself; to obtain "itself", use the name previously bound. Of course, we need not be limited to "self-cycles": we can also have mutually-cyclic data (where no one element is cyclic but their combination is).

9.2 Recursive Functions

In a shift in terminology, a recursive function is not a reference to a same *kind* of function but rather to the same function *itself*. It's useful to first ensure we've first extended our language with conditionals (even of the kind that only check for `0`, as described earlier: section 5), so we can write non-trivial programs that terminate.

Let's now try to write a recursive factorial:

```
(let ([fact (lambda (n)
              (if0 n
                   1
                   (* n (fact (- n 1)))))])
  (fact 10))
```

But this doesn't work at all! The inner `fact` gives an unbound identifier error, just as in our cyclic datum example.

It is no surprise that we should encounter the same error, because it has the same cause. Our traditional binding mechanism does not automatically make function definitions cyclic (indeed, in some early programming languages, they were not: misguidedly, recursion was considered a special *feature*). Instead, if we want recursion—i.e., for a function definition to cyclically refer to itself—we must implement it by hand.

Because you typically write *top-level* definitions, you don't encounter this issue. At the top-level, every binding is implicitly a variable or a box. As a result, the pattern below is more-or-less automatically put in place for you. This is why, when you want a recursive local binding, you must use `letrec` or `local`, not `let`.

The means to do so is now clear: the problem is the same one we diagnosed before, so we can reuse the same solution. We again have to follow a three-step process: first create a placeholder, then refer to the placeholder where we want the cyclic reference, and finally mutate the placeholder before use. Thus:

```
(let ([fact (box 'dummy)])
  (let ([fact-fun
         (lambda (n)
           (if (zero? n)
               1
               (* n ((unbox fact) (- n 1)))))])
    (begin
      (set-box! fact fact-fun)
      ((unbox fact) 10))))
```

In fact, we don't even need `fact-fun`: I've used that binding just for clarity. Observe that because it isn't recursive, and we have identifiers rather than variables, its use can simply be substituted with its value:

```
(let ([fact (box 'dummy)])
  (begin
    (set-box! fact
              (lambda (n)
                (if (zero? n)
                    1
                    (* n ((unbox fact) (- n 1))))))
    ((unbox fact) 10)))
```

There is the small nuisance of having to repeatedly unbox `fact`. In a language with variables, this would be even more seamless:

```
(let ([fact 'dummy])
    (begin
      (set! fact
            (lambda (n)
              (if (zero? n)
                  1
                  (* n (fact (- n 1))))))
      (fact 10)))
```

Indeed, one use for variables is that they simplify the desugaring of the above pattern, instead of requiring every use of a cyclically-bound identifier to be unboxed. On the other hand, with a little extra effort the desugaring process could take care of doing the unboxing, too.

9.3 Premature Observation

Our preceding discussion of this pattern shows a clear temporal sequencing: create, update, use. We can capture it in a desugaring rule. Suppose we add the following new syntax:

```
(rec name value body)
```

As an example of its use,

```
(rec fact
     (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))
     (fact 10))
```

would evaluate to the factorial of 10. This new syntax would desugar to:

```
(let ([name (box 'dummy)])
  (begin
    (set-box! name value)
     body))
```

Where we assume that all references to `name` in `value` and `body` have been rewritten to `(unbox name)`, or alternatively that we instead use variables:

```
(let ([name 'dummy])
  (begin
    (set! name value)
    body))
```

This naturally inspires a question: what if we get these out of order? Most interestingly, what if we try to use `name` before we're done updating its true value into place? Then we observe the state of the system right after creation, i.e., we can see the placeholder in its raw form.

The simplest example that demonstrates this is as follows:

```
(letrec ([x x])
  x)
```

or equivalently,

```
(local ([define x x])
  x)
```

In most Racket variants, this *leaks* the initial value given to the placeholder—a value that was never meant for public consumption. This is troubling because it is, after all, a legitimate *value*, which means it can probably be used in at least some computations. If a developer accesses and uses it inadvertently, however, they are effectively computing with nonsense.

There are generally three solutions to this problem:

1. Make sure the value is sufficiently obscure so that it can never be used in a meaningful context. This means values like `0` are especially bad, and indeed most common datatypes should be shunned. Instead, the language might create a new type of value just for use here. Passed to any other operation, this will result in an error.

2. Explicitly check every use of an identifier for belonging to this special "premature" value. While this is technically feasible, it imposes an enormous performance penalty on a program. Thus, it is usually only employed in teaching languages.

3. Allow the recursion constructor to be used only in the case of binding functions, and then make sure that the right-hand side of the binding is *syntactically* a function. Unfortunately, this solution can be a bit drastic because it precludes writing, for instance, structures to create graphs.

9.4 Without Explicit State

As you may be aware, there is another way to define recursive functions (and hence recursive data) that does not leverage explicit mutation operations.

Do Now!

> You've already seen what goes wrong when we try to use just `let` to define a recursive function. Try harder. Hint: Substitute more. And then some more. And more!

Obtaining recursion from just functions is an amazing idea, and I use the term literally. It's written up well by Daniel P. Friedman and Matthias Felleisen in their book, *The Little Schemer*. Read about it in their sample chapter online.

Exercise

Does the above solution use state anywhere? Implicitly?

10 Objects

When a language admits functions as values, it provides developers the most natural way to represent a unit of computation. Suppose a developer wants to parameterize some function `f`. Any language lets `f` be parameterized by *passive* data, such as numbers and strings. But it is often attractive to parameterize it over *active* data: a datum that can *compute* an answer, perhaps in response to some information. Furthermore, the function passed to `f` can—assuming lexically-scoped functions—refer to data from the caller without those data having to be revealed to `f`, thus providing a foundation for security and privacy. Thus, lexically-scoped functions are central to the design of many secure programming techniques.

While a function is a splendid thing, it suffers from excessive terseness. Sometimes we might want multiple functions to all close over to the same *shared* data; the sharing especially matters if some of the functions mutate it and expect the others to see the result of those mutations. In such cases, it becomes unwieldly to send just a single function as a parameter; it is more useful to send a group of functions. The recipient then needs a way to choose between the different functions in the group. This grouping of functions, and the means to select one from the group, is the essence of an *object*. We are therefore perfectly placed to study objects having covered functions (section 7) and mutation (section 8)—and, it will emerge, recursion (section 9).

Let's add this notion of objects to our language. Then we'll flesh it out and grow it, and explore the many dimensions in the design space of objects. We'll first show how to add objects to the core language, but because we'll want to prototype many different ideas quickly, we'll soon shift to a desugaring-based strategy. Which one you use depends on whether you think understanding them is critical to understanding the essence of your language. One way to measure this is how complex your desugaring strategy becomes, and whether by adding some key core language enhancements, you can greatly reduce the complexity of desugaring.

I cannot hope to do justice to the enormous space of object systems. Please read *Object-Oriented Programming Languages: Application and Interpretation* by Éric Tanter, which goes into more detail and covers topics ignored here.

10.1 Objects Without Inheritance

The simplest notion of an object—pretty much the only thing everyone who talks about objects agrees about—is that an object is

- a value, that
- maps names to
- stuff: either other values or "methods".

From a minimalist perspective, methods seem to be just functions, and since we already have those in the language, we can put aside this distinction.

We're about to find out that "methods" are awfully close to functions but differ in important ways in how they're called and/or what's bound in them.

10.1.1 Objects in the Core

Therefore, starting from the language with first-class functions, let's define this very simple notion of objects by adding it to the core language. We clearly have to extend our notion of values:

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

We'll extend the expression grammar to support literal object construction expressions:

Observe that this is already a design decision. In some languages, like JavaScript, a developer can write literal objects: a notion so popular that a subset of the syntax for it in JavaScript has become a Web standard, JSON. In other languages, like Java, objects can only be created by invoking a constructor on a class. We can simulate both by assuming that to model the latter kind of language, we must write object literals only in special positions following a stylized convention, as we do when desugaring below.

```
[objC (ns : (listof symbol)) (es : (listof ExprC))]
```

Evaluating such an object expression is easy: we just evaluate each of its expression positions:

```
[objC (ns es) (objV ns (map (lambda (e)
                              (interp e env))
                            es))]
```

Unfortunately, we can't actually *use* an object, because we have no way of obtaining its content. For that reason, we could add an operation to extract members:

```
[msgC (o : ExprC) (n : symbol)]
```

whose behavior is intuitive:

```
[msgC (o n) (lookup-msg n (interp o env))]
```

Exercise

Implement

```
; lookup-msg : symbol * Value -> Value
```

where the second argument is expected to be a `objV`.

In principle, `msgC` can be used to obtain any kind of member but for simplicity, we need only assume that we have functions. To use them, we must apply them to values. This is cumbersome to write in the concrete syntax, so let's assume desugaring has taken care of it for us: the concrete syntax for message invocation includes both the name of the message to fetch and its argument expression,

```
[msgS (o : ExprS) (n : symbol) (a : ExprS)]
```

and this desugars into `msgC` composed with application:

```
[msgS (o n a) (appC (msgC (desugar o) n) (desugar a))]
```

With this we have a full first language with objects. For instance, here is an object definition and invocation:

```
(letS 'o (objS (list 'add1 'sub1)
               (list (lamS 'x (plusS (idS 'x) (numS 1)))
                     (lamS 'x (plusS (idS 'x) (numS -1)))))
      (msgS (idS 'o) 'add1 (numS 3)))
```

and this evaluates to `(numV 4)`.

10.1.2 Objects by Desugaring

While defining objects in the core language may be worthwhile, it's an unwieldy way to go about studying them. Instead, we'll use Racket to represent objects, sticking to the parts of the language we already know how to implement in our interpreter. That is, we'll assume that we are looking at the *output of desugaring*. (For this reason, we'll also stick to stylized code, potentially writing unnecessary expressions on the grounds that this is what a simple program generator would produce.)

Alert: All the code that follows will be in `#lang plai`, *not* in the typed language.

Exercise

> Why `#lang plai`? What problem*s* do you encounter when you try to type the following code? Are some of them amenable to easy fixes, such as introducing a new datatype and applying it consistently? How about if we make simplifications for the purposes of modeling, such as assuming methods have only one argument? Or are some of them less tractable?

10.1.3 Objects as Named Collections

Let's begin by reproducing the object language we had above. An object is just a value that dispatches on a given name. For simplicity, we'll use `lambda` to represent the object and `case` to implement the dispatching.

Observe that basic objects are a generalization of `lambda` to have multiple "entry-points". Conversely, a `lambda` is an object with just one entry-point, so it doesn't need a "method name" to disambiguate.

```
(define o-1
  (lambda (m)
    (case m
      [(add1) (lambda (x) (+ x 1))]
      [(sub1) (lambda (x) (- x 1))])))
```

This is the same object we defined earlier, and we use its method in the same way:

```
(test ((o-1 'add1) 5) 6)  ;; the test succeeds
```

Of course, writing method invocations with these nested function calls is unwieldy (and is about to become even more so), so we'd be best off equipping ourselves with a convenient syntax for invoking methods—the same one we saw earlier (`msgS`), but here we can simply define it as a function:

We've taken advantage of Racket's variable-arity syntax: `. a` says "bind all the remaining—zero or more—arguments to a list named `a`". `apply` "splices" in such lists of arguments to call functions.

```
(define (msg o m . a)
  (apply (o m) a))
```

This enables us to rewrite our test:

```
(test (msg o-1 'add1 5) 6)
```

Do Now!

Something very important changed when we switched to the desugaring strategy. Do you see what it is?

Recall the syntax definition we had earlier:

```
[msgC (o : ExprC) (n : symbol)]
```

The "name" position of a message was very explicitly a *symbol*. That is, the developer had to write the literal name of the symbol there. In our desugared version, the name position is just an expression that must evaluate to a symbol; for instance, one could have written

```
(test ((o-1 (string->symbol "add1")) 5) 6)  ;; this also succeeds
```

This is a general problem with desugaring: the target language may allow expressions that have no counterpart in the source, and hence cannot be mapped back to it. Fortunately we don't often need to perform this inverse mapping, though it does arise in some debugging and program comprehension tools. More subtly, however, we must ensure that the target language does not produce *values* that have no corresponding equivalent in the source.

Now that we have basic objects, let's start adding the kinds of features we've come to expect from most object systems.

10.1.4 Constructors

A constructor is simply a function that is invoked at object construction time. We currently lack such a function. by turning an object from a literal into a function that takes constructor parameters, we achieve this effect:

```
(define (o-constr-1 x)
  (lambda (m)
    (case m
      [(addX) (lambda (y) (+ x y))])))

(test (msg (o-constr-1 5) 'addX 3) 8)
(test (msg (o-constr-1 2) 'addX 3) 5)
```

In the first example, we pass `5` as the constructor's argument, so adding `3` yields `8`. The second is similar, and shows that the two invocations of the constructors don't interfere with one another.

10.1.5 State

Many people believe that objects primarily exist to encapsulate state. We certainly haven't lost that ability. If we desugar to a language with variables (we could equivalently use boxes, in return for a slight desugaring overhead), we can easily have multiple methods mutate common state, such as a constructor argument:

Alan Kay, who won a Turing Award for inventing Smalltalk and modern object technology, disagrees. In *The Early History of Smalltalk*, he says, "[t]he small scale [motivation for OOP] was to find a more flexible version of assignment, and then to try to eliminate it altogether". He adds, "It is unfortunate that much of what is called 'object-oriented programming' today is simply old style programming with fancier constructs. Many programs are loaded with 'assignment-style' operations now done by more expensive attached procedures."

```
(define (o-state-1 count)
  (lambda (m)
    (case m
      [(inc) (lambda () (set! count (+ count 1)))]
      [(dec) (lambda () (set! count (- count 1)))]
      [(get) (lambda () count)])))
```

For instance, we can test a sequence of operations:

```
(test (let ([o (o-state-1 5)])
        (begin (msg o 'inc)
               (msg o 'dec)
               (msg o 'get)))
      5)
```

and also notice that mutating one object doesn't affect another:

```
(test (let ([o1 (o-state-1 3)]
            [o2 (o-state-1 3)])
        (begin (msg o1 'inc)
               (msg o1 'inc)
               (+ (msg o1 'get)
                  (msg o2 'get))))
      (+ 5 3))
```

10.1.6 Private Members

Another common object language feature is private members: ones that are visible only inside the object, not outside it. These may seem like an additional feature we need to implement, but we already have the necessary mechanism in the form of locally-scoped, lexically-bound variables:

Except that, in Java, instances of other classes of the same type are privy to "private" members. Otherwise, you would simply never be able to implement an Abstract Data Type.

```
(define (o-state-2 init)
  (let ([count init])
    (lambda (m)
      (case m
        [(inc) (lambda () (set! count (+ count 1)))]
```

```
      [(dec) (lambda () (set! count (- count 1)))]
      [(get) (lambda () count)])))))
```

The desugaring above provides no means for accessing `count`, and lexical scoping ensures that it remains hidden to the world.

10.1.7 Static Members

Another feature often valuable to users of objects is *static* members: those that are common to all instances of the "same" type of object. This, however, is merely a lexically-scoped identifier (making it private) that lives outside the constructor (making it common to all uses of the constructor):

We use quotes because there are many notions of sameness for objects. And then some.

```
(define o-static-1
  (let ([counter 0])
    (lambda (amount)
      (begin
        (set! counter (+ 1 counter))
        (lambda (m)
          (case m
            [(inc) (lambda (n) (set! amount (+ amount n)))]
            [(dec) (lambda (n) (set! amount (- amount n)))]
            [(get) (lambda () amount)]
            [(count) (lambda () counter)])))))))
```

We've written the counter increment where the "constructor" for this object would go, though it could just as well be manipulated inside the methods.

To test it, we should make multiple objects and ensure they each affect the global count:

```
(test (let ([o (o-static-1 1000)])
        (msg o 'count))
      1)

(test (let ([o (o-static-1 0)])
        (msg o 'count))
      2)
```

10.1.8 Objects with Self-Reference

Until now, our objects have simply been packages of named functions: functions with multiple named entry-points, if you will. We've seen that many of the features considered important in object systems are actually simple patterns over functions and scope, and have indeed been used—without names assigned to them—for decades by programmers armed with `lambda`.

One characteristic that actually distinguishes object systems is that each object is automatically equipped with a reference to the same object, often called `self` or `this`.

I prefer this slightly dry way of putting it to the anthropomorphic "knows about itself" terminology often adopted by object advocates. Indeed, note that we have gotten this far into object system properties without ever needing to resort to

Can we implement this easily?

Self-Reference Using Mutation

Yes, we can, because we have seen just this very pattern when we implemented recursion; we'll just generalize it now to refer not just to the same box or function but to the same object.

```
(define o-self!
  (let ([self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(first) (lambda (x) (msg self 'second (+ x 1)))]
                [(second) (lambda (x) (+ x 1))])))
      self)))
```

Observe that this is precisely the recursion pattern (section 9.2), adapted slightly. We've tested it having `first` send a method to its own `second`. Sure enough, this produces the expected answer:

```
(test (msg o-self! 'first 5) 7)
```

Self-Reference Without Mutation

If you studied how to implement recursion without mutation, you'll notice that the same solution applies here, too. Observe:

```
(define o-self-no!
  (lambda (m)
    (case m
      [(first) (lambda (self x) (msg/self self 'second (+ x 1)))]
      [(second) (lambda (self x) (+ x 1))])))
```

Each method now takes `self` as an argument. That means method invocation must be modified to follow this new pattern:

```
(define (msg/self o m . a)
  (apply (o m) o a))
```

That is, when invoking a method on `o`, we must pass `o` as a parameter to the method. Obviously, this approach is dangerous because we can potentially pass a *different* object as the "`self`". Exposing this to the developer is therefore probably a bad idea; if this implementation technique is used, it should only be done in desugaring.

Nevertheless, Python exposes just this *in its surface syntax*. While this tribute to the Y-combinator is touching, perhaps the resultant brittleness was unnecessary.

10.1.9 Dynamic Dispatch

Finally, we should make sure our objects can handle a characteristic attribute of object systems, which is the ability to invoke a method without the caller having to know or decide which object will handle the invocation. Suppose we have a binary

tree data structure, where a tree consists of either empty nodes or leaves that hold a value. In traditional functions, we are forced to implement the equivalent some form of conditional—either a `cond` or a `type-case` or pattern-match or other moral equivalent—that exhaustively lists and selects between the different kinds of trees. If the definition of a tree grows to include new kinds of trees, each of these code fragments must be modified. Dynamic dispatch solves this problem by making that conditional branch disappear from the user's program and instead be handled by the method selection code *built into the language*. The key feature that this provides is an *extensible conditional*. This is one dimension of the extensibility that objects provide.

This property—which appears to make systems more *black-box extensible* because one part of the system can grow without the other part needing to be modified to accommodate those changes—is often hailed as a key benefit of object-orientation. While this is indeed an advantage objects have over functions, there is a dual advantage that functions have over objects, and indeed many object programmers end up contorting their code—using the Visitor pattern—to make it look more like a function-based organization. Read *Synthesizing Object-Oriented and Functional Design to Promote Re-Use* for a running example that will lay out the problem in its full glory. Try to solve it in your favorite language, and see the Racket solution.

Let's now defined our two kinds of tree objects:

```
(define (mt)
  (let ([self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(add) (lambda () 0)])))
      self)))

(define (node v l r)
  (let ([self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(add) (lambda () (+ v
                                     (msg l 'add)
                                     (msg r 'add)))])))
      self)))
```

With these, we can make a concrete tree:

```
(define a-tree
  (node 10
        (node 5 (mt) (mt))
        (node 15 (node 6 (mt) (mt)) (mt))))
```

And finally, test it:

```
(test (msg a-tree 'add) (+ 10 5 15 6))
```

Observe that both in the test case and in the `add` method of `node`, there is a reference to `'add` without checking whether the recipient is a `mt` or `node`. Instead, the run-time system extracts the recipient's `add` method and invokes it. This missing conditional in the user's program is the essence of dynamic dispatch.

10.2 Member Access Design Space

We already have two orthogonal dimensions when it comes to the treatment of member names. One dimension is whether the name is provided statically or computed, and the other is whether the set of names is fixed or variable:

	Name is Static	**Name is Computed**
Fixed Set of Members	As in base Java.	As in Java with reflection to compute the name.
Variable Set of Members	Difficult to envision (what use would it be?).	Most scripting languages.

Only one case does not quite make sense: if we force the developer to specify the member name in the source file explicitly, then no new members would be accessible (and some accesses to previously-existing, but deleted, members would fail). All other points in this design space have, however, been explored by languages.

The lower-right quadrant corresponds closely with languages that use hash-tables to represent objects. Then the name is simply the index into the hash-table. Some languages carry this to an extreme and use the same representation even for numeric indices, thereby (for instance) conflating objects with dictionaries and even arrays. Even when the object only handles "member names", this style of object creates significant difficulty for type-checking [REF] and is hence not automatically desirable.

Therefore, in the rest of this section, we will stick with "traditional" objects that have a fixed set of names and even static member name references (the top-left quadrant). Even then, we will find there is much, much more to study.

10.3 What (Goes In) Else?

Until now, our `case` statements have not had an `else` clause. One reason to do so would be if we had a variable set of members in an object, though that is probably better handled through a different representation than a conditional: a hash-table, for instance, as we've discussed above. In contrast, if an object's set of members is fixed, desugaring to a conditional works well for the purpose of illustration (because it *emphasizes* the fixed nature of the set of member names, which a hash table leaves open to interpretation—and also error). There is, however, another reason for an `else` clause, which is to "chain" control to another, *parent*, object. This is called *inheritance*.

Let's return to our model of desugared objects above. To implement inheritance, the object must be given "something" to which it can delegate method invocations that it does not recognize. A great deal will depend on what that "something" is.

One answer could be that it is simply another object.

```
(case m
  ...
  [else (parent-object m)])
```

Due to our representation of objects, this application effectively searches for the method in the parent object (and, presumably, recursively in its parents). If a method matching the name is found, it returns through this chain to the original call in `msg` that sought the method. If none is found, the final object presumably signals a "message not found" error.

Exercise

Observe that the application `(parent-object m)` is like "half a `msg`", just like an l-value was "half a value lookup" [REF]. Is there any connection?

Let's try this by extending our trees to implement another method, `size`. We'll write an "extension" (you may be tempted to say "sub-class", but hold off for now!) for each `node` and `mt` to implement the `size` method. We intend these to extend the existing definitions of `node` and `mt`, so we'll use the extension pattern described above.

We're not editing the existing definitions because that is supposed to be the whole point of object inheritance: to reuse code in a black-box fashion. This also means different parties, who do not know one another, can each extend the same base code. If they had to edit the base, first they have to find out about each other, and in addition, one might dislike the edits of the other. Inheritance is meant to sidestep these issues entirely.

10.3.1 Classes

Immediately we see a difficulty. Is this the constructor pattern?

```
(define (node/size parent-object v l r)
  ...)
```

That suggests that the parent is at the "same level" as the object's constructor fields. That seems reasonable, in that once all these parameters are given, the object is "fully defined". However, we also still have

```
(define (node v l r)
  ...)
```

Are we going to write all the parameters twice? (Whenever we write something twice, we should worry that we may not do so consistently, thereby inducing subtle errors.) Here's an alternative: `node/size` can *construct the instance of* `node` that is its parent. That is, `node/size`'s parent parameter is not the parent *object* but rather the parent's object *maker*.

```
(define (node/size parent-maker v l r)
  (let ([parent-object (parent-maker v l r)]
        [self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(size) (lambda () (+ 1
                                      (msg l 'size)
                                      (msg r 'size)))]
                [else (parent-object m)])))
      self)))

(define (mt/size parent-maker)
  (let ([parent-object (parent-maker)]
        [self 'dummy])
    (begin
      (set! self
            (lambda (m)
```

```
            (case m
              [(size) (lambda () 0)]
              [else (parent-object m)])))
      self)))
```

Then the object constructor must remember to pass the parent-object maker on every invocation:

```
(define a-tree/size
  (node/size node
             10
             (node/size node 5 (mt/size mt) (mt/size mt))
             (node/size node 15
                        (node/size node 6 (mt/size mt) (mt/size mt))
                        (mt/size mt))))
```

Obviously, this is something we might simplify with appropriate syntactic sugar. We can confirm that both the old and new tests still work:

```
(test (msg a-tree/size 'add) (+ 10 5 15 6))
(test (msg a-tree/size 'size) 4)
```

Exercise

Rewrite this block of code using self-application instead of mutation.

What we have done is capture the essence of a *class*. Each function parameterized over a parent is...well, it's a bit tricky, really. Let's call it a *blob* for now. A blob corresponds to what a Java programmer defines when they write a `class`:

```
class NodeSize extends Node { ... }
```

Do Now!

So why are we going out of the way to not call it a "class"?

When a developer invokes a Java class's constructor, it in effect constructs objects all the way up the inheritance chain (in practice, a compiler might optimize this to require only one constructor invocation and one object allocation). These are private copies of the objects corresponding to the parent classes (private, that is, up to the presence of static members). There is, however, a question of how much of these objects is visible. Java chooses that—unlike in our implementation above—only one method of a given name (and signature) remains, no matter how many there might have been on the inheritance chain, whereas every field remains in the result, and can be accessed by casting. The latter makes some sense because each field presumably has invariants governing it, so keeping them separate (and hence all present) is wise. In contrast, it is easy to imagine an implementation that also makes all the methods available, not only the ones lowest (i.e., most refined) in the inheritance hierarchy. Many scripting languages take the latter approach.

Exercise

The code above is fundamentally broken. The `self` reference is to the same *syntactic* object, whereas it needs to refer to the most-refined object: this is known as *open recursion*. Modify the object representations so that `self` always refers to the most refined version of the object. Hint: You will find the self-application method (section 10.1.8.2) of recursion handy.

This demonstrates the other form of extensibility we get from traditional objects: *extensible recursion*.

10.3.2 Prototypes

In our description above, we've supplied each class with a description of its parent *class*. Object construction then makes instances of each as it goes up the inheritance chain. There is another way to think of the parent: not as a class to be instantiated but, instead, directly as an object itself. Then all children with the same parent would observe the very same object, which means changes to it from one child object would be visible to another child. The shared parent object is known as a *prototype*.

The archetypal prototype-based language is Self. Though you may have read that languages like JavaScript are "based on" Self, there is value to studying the idea from its source, especially because Self presents these ideas in their purest form.

Some language designers have argued that prototypes are more primitive than classes in that, with other basic mechanisms such as functions, one can recover classes from prototypes—but not the other way around. That is essentially what we have done above: each "class" function contains inside it an object description, so a class is an object-returning-function. Had we exposed these are two different operations and chosen to inherit directly an object, we would have something akin to prototypes.

Exercise

Modify the inheritance pattern above to implement a Self-like, prototype-based language, instead of a class-based language. Because classes provide each object with distinct copies of their parent objects, a prototype-language might provide a *clone* operation to simplify creation of the operation that simulates classes atop prototypes.

10.3.3 Multiple Inheritance

Now you might ask, why is there only one fall-through option? It's easy to generalize this to there being many, which leads naturally to *multiple inheritance*. In effect, we have multiple objects to which we can chain the lookup, which of course raises the question of what order in which we should do so. It would be bad enough if the ascendants were arranged in a tree, because even a tree does not have a canonical order of traversal: take just breadth-first and depth-first traversal, for instance (each of which has compelling uses). Worse, suppose a blob `A` extends `B` and `C`; but now suppose `B` and `C` each extend `D`. Now we have to confront this question: will there be one or two `D` objects in the instance of `A`? Having only one saves space and might interact better with our expectations, but then, will we visit this object once or twice? Visiting it twice should not make any difference, so it seems unnecessary. But visiting it once means the behavior of one of `B` or `C` might change. And so on. As a result, virtually every multiple-inheritance language is accompanied by a subtle algorithm merely to define the lookup order.

This infamous situation is called *diamond inheritance*. If you choose to include multiple inheritance in your language you can lose yourself for days in design decisions on this. Because it is highly unlikely you will find a canonical answer, your pain will have only begun.

Multiple inheritance is only attractive until you've thought it through.

10.3.4 Super-Duper!

Many languages have a notion of super-invocations, i.e., the ability to invoke a method or access a field higher up in the inheritance chain. This includes doing so at the point of object construction, where there is often a requirement that all constructors be invoked, to make sure the object is properly defined.

Note that I say "the" and "chain". When we switch to multiple inheritance, these concepts are replaced with something much more complex.

We have become so accustomed to thinking of these calls as going "up" the chain that we may have forgotten to ask whether this is the most natural direction. Keep in mind that constructors and methods are expected to enforce *invariants*. Whom should we trust more: the super-class or the sub-class? One argument would say that the sub-class is most refined, so it has the most global view of the object. Conversely, each super-class has a vested interest in protecting its invariants against violation by ignorant sub-classes.

These are two fundamentally opposed views of what inheritance means. Going up the chain means we view the extension as *replacing* the parent. Going down the chain means we view the extension as *refining* the parent. Because we normally associate sub-classing with refinement, why do our languages choose the "wrong" order of calling? Some languages have, therefore, explored invocation in the downward direction by default.

gbeta is a modern programming language that supports `inner`, as well as many other interesting features. It is also interesting to consider combining both directions.

10.3.5 Mixins and Traits

Let's return to our "blobs".

When we write a `class` in Java, what are we really defining between the opening and closing braces? It is not the entire class: that depends on the parent that it extends, and so on recursively. Rather, what we define inside the braces is a *class extension*. It only becomes a full-blown class because we *also* identify the parent class in the same place.

Naturally, we should ask: Why? Why not separate the act of *defining an extension* from *applying the extension to a base class*? That is, suppose instead of

```
class C extends B { ... }
```

we instead write:

```
classext E { ... }
```

and separately

```
class C = E(B)
```

where `B` is some already-defined class.

Thusfar, it looks like we've just gone to great lengths to obtain what we had before. However, the function-application-like syntax is meant to be suggestive: we can "apply" this extension to several different base classes. Thus:

```
class C1 = E(B1);
class C2 = E(B2);
```

and so on. What we have done by separating the definition of `E` from that of the class it extends is to *liberate class extensions from the tyranny of the fixed base class*. We have a name for these extensions: they're called *mixins*.

The term "mixin" originated in Common Lisp, where it was a particular pattern of using multiple inheritance. Lipstick on a pig.

Mixins make class definition more compositional. They provide many of the benefits of multiple-inheritance (reusing multiple fragments of functionality) but within the aegis of a single-inheritance language (i.e., no complicated rules about lookup order). Observe that when desugaring, it's actually quite easy to add mixins to the language. A mixin is primarily a "function over classes';. Because we have already determined how to desugar classes, and our target language for desugaring also has functions, and classes desugar to expressions that can be nested inside functions, it becomes almost trivial to implement a simple model of mixins.

This is a case where the greater generality of the target language of desugaring can lead us to a *better* construct, if we reflect it back into the source language.

In a typed language, a good design for mixins can actually improve object-oriented programming practice. Suppose we're defining a mixin-based version of Java. If a mixin is effectively a class-to-class function, what is the "type" of this "function"? Clearly, mixin ought to use *interfaces* to describe what it expects and provides. Java already enables (but does not require) the latter, but it does not enable the former: a class (extension) extends another *class*—with all its members visible to the extension—not its *interface*. That means it obtains all of the parent's behavior, not a specification thereof. In turn, if the parent changes, the class might break.

In a mixin language, we can instead write

```
mixin M extends I { ... }
```

where `I` is an interface. Then `M` can only be applied to a class that satisfies the interface `I`, and in turn the language can *ensure that only members specified in `I` are visible in `M`*. This follows one of the important principles of good software design.

"Program to an interface, not an implementation." —*Design Patterns*

A good design for mixins can go even further. A class can only be used once in an inheritance chain, by definition (if a class eventually referred back to itself, there would be a cycle in the inheritance chain, causing potential infinite loops). In contrast, when we compose functions, we have no qualms about using the same function twice (e.g.: `(map ... (filter ... (map ...)))`). Is there value to using a mixin twice?

There certainly is! See sections 3 and 4 of *Classes and Mixins*.

Mixins solve an important problem that arises in the design of libraries. Suppose we have a dozen different features which can be combined in different ways. How many classes should we provide? Furthermore, not all of these can be combined with each other. It is obviously impractical to generate the entire combinatorial explosion of classes. It would be better if the devleoper could pick and choose the features they care about, with some mechanism to prevent unreasonable combinations. This is precisely the problem that mixins solve: they provide the class extensions, which the developers can combine, in an interface-preserving way, to create just the classes they need.

Mixins are used extensively in the Racket GUI library. For instance, `color:text-mixin` consumes basic text editor interfaces and implements the colored text editor interface. The latter is iself a basic text editor interface, so additional basic text mixins can be applied to the result.

Exercise

How does your favorite object-oriented library solve this problem?

Mixins do have one limitation: they enforce a linearity of composition. This strictness is sometimes misplaced, because it puts a burden on programmers that may not be necessary. A generalization of mixins called *traits* says that instead of extending a single mixin, we can extend a *set* of them. Of course, the moment we extend more

than one, we must again contend with potential name-clashes. Thus traits must be equipped with mechanisms for resolving name clashes, often in the form of some name-combination algebra. Traits thus offer a nice complement to mixins, enabling programmers to choose the mechanism that best fits their needs. As a result, Racket provides both mixins and traits.

11 Memory Management

11.1 Garbage

We use the term *garbage* to refer to allocated memory that is no longer necessary. There are two distinct kinds of allocations that a typical programming language runtime system performs. One kind is for the environment; this follows a push-and-pop discipline consistent with the nature of static scope. Returning from a procedure returns that procedure's allocated environment space for subsequent use, seemingly free of cost. In contrast, allocation on the store has to follow an value's lifetime, which could outlive that of the scope in which it was created—indeed, it may live forever. Therefore, we need a different strategy for recovering space consumed by store-allocated garbage.

It's not free! The machine has to execute an explicit "pop" instruction to recover that space. As a result, it is not *necessarily* cheaper than other memory management strategies.

There are many methods for recovering this space, but they largely fall into two camps: manual and automatic. Manual collection depends on the developer being able to know and correctly discard unwated memory. Traditionally, humans have not proven especially good at this (though in some cases they have knowledge a machine might not [REF]). Over several decades, therefore, automated methods have become nearly ubiquitous.

11.2 What is "Correct" Garbage Recovery?

Garbage recovery should neither recover space too early (*soundness*) nor too late (*completeness*). While both can be regarded as flaws, they are not symmetric in their impact: arguably, recovering too early is much worse. That is because if we recover a store location prematurely, the computation will continue to use it and potentially write other data into it, thereby working with nonsensical data. This leads at the very least to program incorrectness, but in more extreme situations leads to much worse phenomena such as security violations. In contrast, holding on to memory for too long decreases performance and eventually causes the program to terminate even though, in a platonic sense, it had memory available. This performance degradation and premature termination is always annoying, and in certain mission-critical systems can be deeply problematic, but at least the program does not compute nonsense.

Ideally we would love to have all three: automation, soundness, and completeness. However, we face a classic "pick two" tradeoff. Ideal humans are capable of attaining both soundness and completeness, but in practice rarely achieve either. A computer can offer automation and either soundness or completeness, but computability arguments demonstrate that automation can't be accompanied by both of the others. In practice, therefore, automated techniques offer soundness, on the grounds that: (a) it does the

You, surely, are perfect, but what of your fellow developers? And by the way, the economics discipline has been looking for you.

least harm, (b) it is relatively easy to implement, and (c) with some human intervention it can more closely approximate completeness.

11.3 Manual Reclamation

The most manual approach would be to entrust all de-allocation to the human. In C, for instance, there are two basic primitives: `malloc` for allocation and `free` to reclaim. `malloc` consumes a size and returns a reference to a store-allocated value; `free` consumes the references and reclaims its assocated memory.

"Moloch has been used figuratively in English literature from John Milton's *Paradise Lost* (1667) to Allen Ginsberg's 'Howl' (1955), to refer to a person or thing demanding or requiring a very costly sacrifice." —Wikipedia on Moloch

11.3.1 The Cost of Fully-Manual Reclamation

Let's start by asking what the cost of these operations is. We might begin by assuming that `malloc` has an associated register pointing into the store (like `new-loc` [REF]), and on every allocation it simply obtains the next free locations. This model is extremely simple—in fact, deceptively so. The problem arises when we `free` these values. Provided the first `free` is the last `malloc`, we would encounter no problem; but store data often do not follow a stack discipline. When we free anything but the most recently allocated value, we leave holes in the store. These holes lead to *fragmentation*, and in the worst case we become unable to allocate any objects even though there is ample space in the store—just split up across many fragments, no one of which is large enough.

"I do not consider it coincidental that this name sounds like `malloc`." —Ian Barland

Exercise

> In principle, we could eliminate fragmentation by making all the free space be contiguous. What does it take to do so? Think through all the consequences and sketch whether you can in fact do this manually.

While fragmentation remains an insuperable problem in most manual memory management schemes, there is more to consider even in this seemingly simple discipline. What happens when we `free` a value? The run-time system has to somehow record that it is now available for future allocation. It does by maintaining a *free list*: a linked-list of the free spaces. A little reflection immediately suggests a question: where is the free list itself stored, and who manages *its* memory? The answer is that the free list references are stored in the freed cells, which immediately implies a minimum size for each allocation.

In principle, then, each `malloc` must now traverse the free list to find a suitable freed spot. I say "suitable" because the allocator must make a complex decision. Should it take the first slot that matches or a later one? And either way, what does "matches" mean? Should it take only slots the exact right size, or take larger slots and break them up into smaller ones (thereby increasing the likelihood of creating unusably small holes)? And so on.

Developers like allocation to be cheap. Therefore, in practice, allocation systems tend to use just a fixed set of sizes, often in powers of two. This makes it possible to maintain not one but many free list*s*, each of holes of the same size (which is a power of two). A table refers to each of these lists, and indexing in the table is cheap by using bit-shifting. In return, developers sacrifice space, because objects not a power-of-two

Failing to make allocation cheap makes developers try to encode tricks based on reusing values, thereby reducing clarity and quite possibly also correctness.

size will end up being needlessly padded. (This is a classic computer science trade-off: trading space for time.) Also, `free` must put the freed memory in the right slot, and perhaps even break up larger blocks into multiple smaller blocks to prepare for future allocations. Nothing about this model is inherently as cheap as it seems.

In particular, `free` is not free.

Of course, all this assumes that developers can function in even a sound, much less complete, fashion. But they don't.

11.3.2 Reference Counting

Because entirely manual reclamation puts an undue burden on developers, some semi-automated techniques have seen long-standing use, most notably *reference counting*.

In reference counting, every value has associated with it a count of how many references it has. The developer is responsible for incrementing and decrementing these counts. When the count reaches zero, the value's space can safely be restored for future reuse.

Observe, immediately, that there are two significant assumptions lurking under this simple definition.

1. That the developer can track every reference. Recall that every alias is also a reference. Thus, a developer who writes

   ```
   (let ([x <some value>])
     (let ([y x])
       ...))
   ```

   has to remember that `y` is a second reference to the same value being referred to by `x`, and increment the count accordingly.

2. That every value has only a finite number of references. This assumption fails when a value has cycles.

Because of the need to manually increment and decrement references, this technique suffers from a lack of both soundness and completeness. Indeed, the second assumption above naturally leads to lack of completeness, while the first assumption points to the simplest way to break soundness.

The perils of manual memory management are subtle and run deeper. Because developers are charged with freeing memory (or, equivalently, managing reference counts), the policy of memory management has to become part of every library's interface: effectively, "Who's going to de-allocate values allocated by this library, and will the library de-allocate values passed to it?" It is unfortunately difficult to document and follow these policies precisely, but even worse, it pollutes the description of the library with low-level details that usually have nothing to do with its intended behavior.

One intriguing idea is to *automate* the insertion of reference increments and decrements. Another is to add cycle-detection to the implementation. Doing both solves many of the above problems, but reference counting suffers from others, too:

- The reference count increases the size of each object. It has to be large enough to not overflow, yet small enough to not appreciably increase the program's memory footprint.

- The time spent to increase and decrease these counts can become significant.
- If an object's reference count becomes zero, everything it refers to must also have their reference counts decreased—possibly recursively. This means a single deallocation action can have a large time impact, barring clever "lazy" techniques (which then increase the program's memory footprint).
- To decrement the reference count, we have to walk objects that are *garbage*. This seems highly counterproductive: to traverse objects we are *no longer interested in*. This has practical consequences: objects we are not interested in may not have been accessed in a while, which means they might have been paged out. The reference counter has to page them back in, just to inform them that they are no longer needed.

For all these reasons, reference counting should be used with utmost care. You should not accept it as a default, but rather ask yourself why it is you reject what are generally better automated techniques.

Exercise

If the reference count overflows, which correctness properties are hurt and how? Examine tradeoffs.

11.4 Automated Reclamation, or Garbage Collection

Some people call reference counting a "garbage collection" technique. I prefer to use the latter term to refer only to fully-automated techniques. But do beware this potential for confusion when browsing the Web.

Now let's briefly examine the idea of having the language's run-time system automate the process of reclaiming garbage. We'll use the abbrevation GC (for *garbage collection*) to refer to both the algorithm and the process, letting context disambiguate.

11.4.1 Overview

Depth-first search is generally preferred because it works well with stack-based implementations. Of course, you might (and should) wonder where the GC's own stack is stored!

The key idea behind all GC algorithms is to traverse memory by following references between values. Traversal begins at a *root set*, which is all the places from which a program can possibly refer to a value in the store. Typically the root set consists of every bound variable in the environment, and any global variables. In an actual working implementation, the implementor must be careful to also note ephemeral values such as references in registers. From this root set, the algorithm walks all accessible values using a variety of algorithms that are usually variations on depth-first search to identify everything that is *live* (i.e., usable through some sequence of program operations). Everything else, by definition, is garbage. Again, different algorithms address the recovery of this space in different ways.

11.4.2 Truth and Provability

If you read carefully, you'll notice that I slipped an *algorithm* into the above description. This is an *implementation detail*, not part of the *specification*! Indeed, the specification of garbage collection is in terms of *truth*: we want to collect precisely all the values that are garbage, no more and no less. But we cannot obtain truth for any Turing-complete programming language, so we must settle for *provability*. And the

style of algorithm described above gives us an efficient "proof" of liveness, rendering the complement garbage. There are of course variations on this scheme that enable us to collect more or less garbage, which correspond to different *strengths* of proof a value's "garbageness".

This last remark highlights a weakness of the strict specification, which says nothing about how much garbage should be collected. It is actually useful to think about the extreme cases.

Do Now!

> It is trivial to define a sound garbage collection strategy. Similarly, it is also trivial to define a complete garbage collection strategy. Do you see how?

To be sound, we simply have to make sure we don't accidentally remove anything that is live. The one way to be absolutely certain of this is to *collect no garbage at all*. Dually, the trivial complete GC collects *everything*. Obviously neither of these is useful (and the latter is certain to be highly dangerous). But this highlights that in practice, we want a GC that is not only sound but as complete as possible, while also being efficient.

11.4.3 Central Assumptions

Being able to soundly perform GC depends on two critical assumptions. The first is one about the language's implementation and the other about the language's semantics.

1. When confronted with a value, the GC needs to know what kind of value it is, and how its memory representation is laid out. For instance, when the traversal reaches a `cons` cell, it must know:

   (a) that this is a `cons` cell; and hence,

   (b) that the `first` is at, say, a four byte offset, and

   (c) that the `rest` is at, say, an eight byte offset.

   Obviously, this property must hold recursively, thus enabling a traversal algorithm to correctly map the values in memory.

2. That programs cannot *manufacture* references in two ways:

   (a) Object references cannot reside outside the implementation's pre-defined root set.

   (b) Object references can only refer to well-defined points in objects.

   When the second property is violated, the GC can effectively go haywire, misinterpreting data. The first property sounds obvious: when it is violated, it seems the run-time system has clearly failed to obey the language's semantics. However, the consequences of this property are subtle, as we discuss below [REF].

11.5 Convervative Garbage Collection

We've explained that the typical root set consists of the environment, global variables, and some choice ephemerals. Where else might references reside?

In most languages, nowhere else. But some languages (I'm looking at you, C and C++) allow references to be turned into arbitrary numbers, and arbitrary numbers to be turned back into references. As a result, in principle, *any* numeric value in the program (which, because of the nature of C and C++'s types, virtually *any* value in the program) could potentially be treated as a reference.

This is problematic for two reasons. First, it means the GC can no longer limit its attention to the small root set; instead, the entire store is now potentially the root set. Second, if the GC tries to modify the object in any way—e.g., to record a "visited" bit during traversal—then it is potentially changing *non-reference* values: e.g., it might actually be changing an innocent numeric constant in the program. As a result, the particular confluence of features in languages like C and C++ conspire to make sound, efficient GC extremely difficult.

But not impossible. A stimulating line of research called *conservative* GC has managed to create reasonably effective GC systems for such languages. The principle behind conservative GC notes that, while in principle every store location might be a root, in practice many of them are not. It then proceeds through a series of increasingly clever observations to deduce what must *not* be a reference (the opposite of a traditional GC) and can hence be safely *ignored*: for instance, on a word-aligned architecture, no odd number can never be a reference. By skipping most of the store, by making some basic assumptions about program behavior (e.g., that it will not manufacture certain kinds of references), and by being careful to not modify the store—e.g., changing bits in values, or moving data around—it can actually yield a reasonably effective GC strategy.

Nevertheless, it is a bit of a dog walking on its hind legs.

Conservative GC is often popular with programming language implementations that are written in, or rely on a base of code in, C or C++. For instance, early versions of Racket relied exclusively on it. There are many good reasons for this:

1. It offers a quick bootstrapping technique, so the language implementor can focus on other, more innovative, features.

2. A language that controls all references (as Racket does) can easily create memory representations that are especially conducive to increasing the effectiveness of the GC (e.g., padding all true numbers with a one in the least-significant-bit).

3. It enables easy interoperation with useful libraries written in C and C++ (provided, of course, that they too meet the expectations of the technique).

A word on vocabulary is in order. As we have argued [REF], *all* practical GC techniques are "conservative" in that they approximate truth with reachability. The word "conservative" has, however, become a term-of-art to refer to a GC technique that operates in an *uncooperative* (and hopefully not *hostile*) run-time system.

11.6 Precise Garbage Collection

In conventional GC terminology, the opposite of "conservative" is *precise*. This, too, is a misnomer, because a GC cannot be precise, i.e., both sound and complete. Rather, precision here is a statement about the ability to identify references: when confronted with a value, a precise GC knows exactly what is and isn't a reference, and where the references are. This removes the monumental effort that a conservative GC has to put into guessing non-references (and hoping to eliminate as many potential references as possible through this process).

Within the space of precise GC, which is what most contemporary language run-time systems use, there is a wide range of implementation techniques. I refer you to Paul Wilson's survey (which, despite its relative age in this fast-moving field, remains an excellent resource), as well as the book and other materials from Richard Jones. In particular, for a quick and readable overview of a generational garbage collector, read *Simple Generational Garbage Collection and Fast Allocation*.

12 Representation Decisions

Go back and look again at our interpreter for function as values [REF]. Do you see something curiously non-uniform about it?

Do Now!

No, really, do. Do you?

Consider how we chose to represent our two different kinds of values: numbers and functions. Ignoring the superficial `numV` and `closV` wrappers, focus on the underlying data representations. We represented the interpreted language's numbers as Racket numbers, but we did not represent the interpreted language's functions (closures) as Racket functions (closures).

That's our non-uniformity. It would have been more uniform to use Racket's representations for both, or also to *not* use Racket's representation for either. So why did we make this particular choice?

We were trying to illustrate and point, and that point is what we will explore right now.

12.1 Changing Representations

For a moment, let's explore numbers. Racket's numbers make a good target for reuse because they are so powerful: we get arbitrary-sized integers (*bignums*), rationals (which benefit from the bignum representation of integers), complex numbers, and so on. Therefore, they can represent most ordinary programming language number systems. However, that doesn't mean they are what we *want*: they could be too little or too much.

- They are too much if what we want is a more restricted number system. For instance, Java prescribes a very specific set of fixed-size representations (e.g., `int` is specified to be 32-bit). Numbers that fall outside these sets cannot be

directly represented as numbers, and arithmetic must respect these sets (e.g., overflowing so that adding `1` to `2147483647` does *not* produce `2147483648`).

- They are too little if we want even richer numbers, whether quaternions or numbers with associated probabilities.

Worse, we didn't even stop and ask what we wanted, but blithely proceeded with Racket numbers.

The reason we did so is because we weren't really interested in the study of numbers; rather, we were interested in programming language features such as functions-as-values. As language designers, however, you should be sure to ask these hard questions up front.

Now let's talk about our representation of closures. We could have instead represented closures by exploiting Racket's corresponding concept, and correspondingly, function application with unvarnished Racket application.

Do Now!

Replace the closure data structure with Racket functions representing functions-as-values.

Here we go:

```
(define-type Value
  [numV (n : number)]
  [closV (f : (Value -> Value))])

(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)]
                        [define a-value (interp a env)])
                  ((closV-f f-value) a-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 (lambda (arg-val)
                         (interp b
                                 (extend-env (bind a arg-val)
                                             env))))]))
```

Exercise

Observe a curious shift. In our previous implementation, the environment was extended in the `appC` case. Here, it's extended in the `lamC` case. Is one of these incorrect? If not, why did this change occur?

This is certainly concise, but we've lost something very important: *understanding*. Saying that a source language function corresponds to `lambda` tells us virtually nothing: if we already knew precisely what `lambda` does we might not be studying it, and

if we didn't, this mapping would teach us absolutely nothing (and might, in fact, pile confusion on top of our ignorance). For the same reason, we did not use Racket's state to understand the varieties of stateful operators [REF].

Once we've understood the feature, however, we should feel to use it as a representation. Indeed, doing so might yield much more concise interpreters because we aren't doing everything manually. In fact, some later interpreters [REF] will become virtually unreadable if we did not exploit these richer representations. Nevertheless, exploiting host language features has perils that we should safeguard against.

It's a little like saying, "Now that we understand addition in terms of increment-by-one, we can use addition to define multiplication: we don't have to use only increment-by-one to define it."

12.2 Errors

When programs go wrong, programmers need a careful presentation of errors. Using host language features runs the risk that users will see host language errors, which they will not understand. Therefore, we have to carefully translate error conditions into terms that the user of our language will understand, without letting the host language "leak through".

Worse, programs that should error might not! For instance, suppose we decide that functions should only appear in top-level positions. If we fail to expressly check for this, desugaring into the more permissive `lambda` may result in an interpreter that produces answers where it should have halted with an error. Therefore, we have to take great care to permit *only the intended surface language to be mapped to the host language*.

As another example, consider the different mutation operations. In our language, attempting to mutate an unbound variable produces an error. In some languages, doing so results in the variable being defined. Failing to pin down our intended semantics is a common language designer error, saying instead, "It is whatever the implementation does". This attitude (a) is lazy and sloppy, (b) may yield unexpected and negative consequences, and (c) makes it hard for you to move your language from one implementation platform to another. Don't ever make this mistake!

12.3 Changing Meaning

Mapping functions-as-values to `lambda` works especially because we intend for the two to *have the same meaning*. However, this makes it difficult to change the meaning of what a function does. Lemme give ya' a hypothetic: suppose we wanted our language to implement dynamic scope. In our original interpreter, this was easy (almost too easy, as history shows). But try to make the interpreter that uses `lambda` implement dynamic scope. It can similarly be difficult or at least subtle to map eager evaluation onto a language with lazy application [REF].

Don't let this go past the hypothetical stage, please.

Exercise

Convert the above interpreter to use dynamic scope.

The point is that the raw data structure representation does not make anything especially easy, but it usually doesn't get in the way, either. In contrast, mapping to host language features can make some intents—mainly, those match what the host language

already does!—especially easy, and others subtle or difficult. There is the added danger that we may not be certain of what the host language's feature does (e.g., does its "lambda" actually implement static scope?).

The moral is that this is a good property to exploit only we want to "pass through" the base language's meaning—-and then it is especially wise because it ensures that we don't accidentally change its meaning. If, however, we want to exploit a significant part of the base language and only augment its meaning, perhaps other implementation strategies [REF] will work just as well instead of writing an interpreter.

12.4 One More Example

Let's consider one more representation change. What is an environment?

An environment is a *map* from names to values (or locations, once we have mutation). We've chosen to implement the mapping through a data structure, but...do we have another way to represent maps? As functions, of course! An environment, then, is a function that takes a name as an argument and return its bound value (or an error):

```
(define-type-alias Env (symbol -> Value))
```

What is the empty environment? It's the one that returns an error no matter what name you try to look up:

```
(define (mt-env [name : symbol])
  (error 'lookup "name not found"))
```

(In principle we should put a type annotation on the return, and it should be `Value`, except of course thiis is vacuous.) Extending an environment with a binding creates a function that takes a name and checks whether it is the name just extended; if so it returns the corresponding value, otherwise it punts to the environment being extended:

```
(define (extend-env [b : Binding] [e : Env])
  (lambda ([name : symbol]) : Value
    (if (symbol=? name (bind-name b))
        (bind-val b)
        (lookup name e))))
```

Finally, how do we look up a name in an environment? We simply *apply* the environment!

```
(define (lookup [n : symbol] [e : Env]) : Value
  (e n))
```

And with that, we're done!

13 Desugaring as a Language Feature

We have thus far extensively discussed and relied upon desugaring, but our current desugaring mechanism have been weak. We have actually used desugaring in two

different ways. One, we have used it to *shrink* the language: to take a large language and distill it down to its core [REF]. But we have also used it to *grow* the language: to take an existing language and add new features to it [REF]. This just shows that desugaring is a tremendously useful feature to have. Indeed, it is so useful that we might ask two questions:

- Because we create languages to simplify the creation of common tasks, what would a language designed to support desugaring look like? Note that by "look" we don't mean only syntax but also its key behavioral properties.
- Given that general-purpose languages are often used as a target for desugaring, why don't they offer desugaring capabilities *in the language itself*? For instance, this might mean extending a base language with the additional language that is the response to the previous question.

We are going to explore the answer to both questions simultaneously, by studying the facilities provided for this by Racket.

13.1 A First Example

Remember that in [REF] we added `let` as syntactic sugar over `lambda`. The pattern we followed was this:

```
(let (var val) body)
```

is transformed into

```
((lambda (var) body) val)
```

DrRacket has a very useful tool called the Macro Stepper, which shows the step-by-step expansion of programs. You should try all the examples in this chapter using the Macro Stepper. For now, however, you should run them in `#lang plai` rather than `#lang plai-typed`.

Do Now!

If this doesn't sound familiar, now would be a good time to refresh your memory of why this works.

The simplest way of describing this transformation would be to state it directly: to write, somehow,

```
(let (var val) body)
->
((lambda (var) body) val)
```

In fact, this is almost precisely what Racket enables you to do.

We'll use the name `my-let` instead of `let` Because the latter is already defined in Racket.

```
(define-syntax my-let-1
  (syntax-rules ()
    [(my-let-1 (var val) body)
     ((lambda (var) body) val)]))
```

`syntax-rules` tells Racket that whenever it sees an expression with `my-let-1` immediately after the opening parenthesis, it should check that it follows the pattern `(my-let-1 (var val) body)`. The `var`, `val` and `body` are *syntactic variables*: they are variables that stand for bodies of code. In particular, they match whatever s-expression is in that position. If the expression matches the pattern, then the syntactic variables are bound to the corresponding parts of the expression, and become available for use in the right-hand side.

The right-hand side—in this case, `((lambda (var) body) val)`—is the output generated. Each of the syntactic variables are replaced with the corresponding parts of the input using our old friend, substitution. This substitution is utterly simplistic: it makes no attempt to. Thus, if we were to try using it with

You may have noticed some additional syntax, such as `()`. We'll explain this later.

```
(my-let-1 (3 4) 5)
```

Racket would not initially complain that `3` is provided in an identifier position; rather, it would let the identifier percolate through, desugaring this into

```
((lambda (3) 5) 4)
```

which in turn produces an error:

```
lambda: expected either <id> or `[<id> : <type>]'
  for function argument in: 3
```

This immediately tells us that the desugaring process is straightforward in its function: it doesn't attempt to guess or be clever, but instead simply rewrites while substituting. The output is an expression that is again subject to desugaring.

As a matter of terminology, this simple form of expression-rewriting is often called a *macro*, as we mentioned earlier in [REF]. Traditionally this form of desugaring is called macro *expansion*, though this term is misleading because the output of desugaring can be smaller than the input (though it is usually larger).

Of course, in Racket, a `let` can bind multiple identifiers, not just one. If we were to write this informally, say on a board, we might write something like `(let ([var val] ...) body) -> ((lambda (var ...) body) val ...)` with the `...` meaning "zero or more", and the intent being that the `var ...` in the output should correspond to the sequence of `vars` in the input. Again, this is almost precisely Racket syntax:

```
(define-syntax my-let-2
  (syntax-rules ()
    [(my-let-2 ([var val] ...) body)
     ((lambda (var ...) body) val ...)]))
```

Observe the power of the `...` notation: the sequence of "pairs" in the input is turned into a pair of sequences in the output; put differently, Racket "unzips" the input sequence. Conversely, this same notation can be used to zip together sequences.

13.2 Syntax Transformers as Functions

Earlier we saw that `my-let-1` does not even attempt to ensure that the syntax in the identifier position is truly (i.e., syntactically) an identifier. We cannot remedy that with

the `syntax-rules` mechanism, but we can with a much more powerful mechanism called `syntax-case`. Because `syntax-case` exhibits many other useful features as well, we'll introduce it and then grow it gradually.

The first thing to understand is that a macro is actually a *function*. It is not, however, a function from regular run-time values to other run-time values, but rather a function *from syntax to syntax*. These functions execute in a world whose purpose is to *create the program to execute*. Observe that we're talking about the program *to* execute: the actual execution of the program may only occur much later (or never at all). This point is actually extremely clear when we examine desugaring, which is very explicitly a function from (one kind of) syntax to (another kind of) syntax. This is perhaps obscured above in two ways:

- The notation of `syntax-rules`, with no explicit parameter name or other "function header", may not make clear that it is a functional transformation (though the rewriting rule format does allude to this fact).
- In desugaring, we specify one atomic function for the entire process. Here, we are actually writing several little functions, one for each kind of new syntactic construct (such as `my-let-1`), and these pieces are woven together by an invisible function that controls the overall rewriting process. (As a concrete example, it is not inherently clear that the output of a macro is expanded further—though a simple example immediately demonstrates that this is indeed the case.)

Exercise

Write one or more macros to confirm that the output of a macro is expanded further.

There is one more subtlety. Because the form of a macro looks rather like Racket code, it is not immediately clear that it "lives in another world". In the abstract, it may be helpful to imagine that the macro definitions are actually written in an entirely different language that processes only syntax. This simplicity is, however, misleading. In practice, program transformers—also called *compilers*—are full-blown programs, too, and need all the power of ordinary programs. This would have necessitated the creation of a parallel language purely for processing programs. This would be wasteful and pointless; therefore, Racket instead endows syntax-transforming programs with the full power of Racket itself.

With that prelude, let's now introduce `syntax-case`. We'll begin by simply rewriting `my-let-1` (under the name `my-let-3`) using this new notation. First, we have to write a header for the definition; notice already the explicit parameter:

<***sc-macro-eg***> **::=**

```
(define-syntax (my-let-3 x)
  <sc-macro-eg-body>)
```

This binds `x` to the entire `(my-let-3 ...)` expression.

As you might imagine, `define-syntax` simply tells Racket you're about to define a new macro. It does not pick precisely how you want to implement it, leaving you free

to use any mechanism that's convenient. Earlier we used `syntax-rules`; now we're going to use `syntax-case`. In particular, `syntax-case` needs to explicitly be given access to the expression to pattern-match:

<*sc-macro-eg-body*> **::=**

```
(syntax-case x ()
  <sc-macro-eg-rule>)
```

Now we're ready to express the rewrite we wanted. Previously a rewriting rule had two parts: the structure of the input and the corresponding output. The same holds here. The first (matching the input) is the same as before, but the second (the output) is a little different:

<*sc-macro-eg-rule*> **::=**

```
[(my-let-3 (var val) body)
 #'((lambda (var) body) val)]
```

Observe the crucial extra characters: `#'`. Let's examine what that means.

In `syntax-rules`, the entire output part simply specifies the structure of the output. In contrast, because `syntax-case` is laying bare the functional nature of transformation, the output part is in fact an arbitrary expression that may perform any computations it wishes. It must simply evaluate to a piece of syntax.

Syntax is actually a distinct datatype. As with any distinct dataype, it has its own rules for construction. Concretely, we construct syntax values by writing `#'`; the following s-expression is treated as a syntax value. (In case you were wondering, the `x` bound in the macro definition above is also of this datatype.)

The syntax constructor, `#'`, enjoys a special property. Inside the output part of the macro, all syntax variables in the input are automatically bound, and replaced on occurrence. As a result, when the expander encounters `var` in the output, say, it replaces `var` with the corresponding part of the input expression.

Do Now!

> Remove the `#'` and try using the above macro definition. What happens?

So far, `syntax-case` merely appears to be a more complicated form of `syntax-rules`: perhaps slightly better in that it more cleanly delineates the functional nature of expansion, and the type of output, but otherwise simply more unwieldy. As we will see, however, it also offers significant power.

Exercise

> `syntax-rules` can actually be expressed as a *macro* over `syntax-case`. Define it.

13.3 Guards

Now we can return to the problem that originally motivated the introduction of `syntax-case`: ensuring that the binding position of a `my-let-3` is syntactically an identifier. For this, you need to know one new feature of `syntax-case`: each rewriting rule can

have two parts (as above), or three. If there are three present, the *middle* one is treated as a *guard*: a predicate that must evaluate to true for expansion to proceed rather than signal a syntax error. Especially useful in this context is the predicate `identifier?`, which determines whether a syntax object is syntactically an identifier (or variable).

Do Now!

Write the guard and rewrite the rule to incorporate it.

Hopefully you stumbled on a subtlety: the argument to `identifier?` is of type *syntax*. It needs to refer to the actual fragment of syntax bound to `var`. Recall that `var` is bound in the syntax space, and `#'` substitutes identifiers bound there. Therefore, the correct way to write the guard is:

```
(identifier? #'var)
```

With this information, we can now write the entire rule:

<***sc-macro-eg-guarded-rule***> **::=**

```
[(my-let-3 (var val) body)
 (identifier? #'var)
 #'((lambda (var) body) val)]
```

Do Now!

Now that you have a guarded rule definition, try to use the macro with a non-identifier in the binding position and see what happens.

13.4 Or: A Simple Macro with Many Features

Consider `or`, which implements disjunction. It is natural, with prefix syntax, to allow `or` to have an arbitrary number of sub-terms. We expand `or` into nested conditionals that determine the truth of the expression.

13.4.1 A First Attempt

Let's try a first version of `or`:

```
(define-syntax (my-or-1 x)
  (syntax-case x ()
    [(my-or-1 e0 e1 ...)
     #'(if e0
           e0
           (my-or-1 e1 ...))]))
```

It says that we can provide any number of sub-terms (more on this in a moment). Expansion rewrites this into a conditional that tests the first sub-term; if this is a true value it returns that value (more on *this* in a moment!), otherwise it is the disjunction of the remaining terms.

Let's try this on a simple example. We would expect this to evaluate to `true`, but instead:

```
> (my-or-1 #f #t)
my-or-1: bad syntax in: (my-or-1)
```
What happened? This expression turned into

```
(if #f
    #f
    (my-or-1 #t))
```

which in turn expanded into

```
(if #f
    #f
    (if #t
        #t
        (my-or-1)))
```

for which there is no definition. That's because the pattern `e0 e1 ...` means *one or more* sub-terms, but we ignored the case when there are zero.

What *should* happen when there are no sub-terms? The identity for disjunction is falsehood.

Exercise

Why is `#f` the right default?

By filling it in below, we illustrate a macro that has more than one rule. Macro rules are matched sequentially, so we should be sure to put the most specific rules first, lest they get overridden by more general ones (though in this particular case, the two rules are non-overlapping). This yields our improved macro:

```
(define-syntax (my-or-2 x)
  (syntax-case x ()
    [(my-or-2)
     #'#f]
    [(my-or-2 e0 e1 ...)
     #'(if e0
           e0
           (my-or-2 e1 ...))]))
```

which now expands as we expect. Though it isn't necessary, we will add a rule for the case when there is only a single sub-term:

```
(define-syntax (my-or-3 x)
  (syntax-case x ()
    [(my-or-3)
     #'#f]
    [(my-or-3 e)
     #'e]
    [(my-or-3 e0 e1 ...)
```

```
      #'(if e0
            e0
            (my-or-3 e1 ...))]))
```

This keeps the output of expansion more concise, which we will find useful below.

Observe that in this version of the macro, the patterns are *not* disjoint: the third (one-or-more sub-terms) subsumes the second (one sub-term). Therefore, it is essential that the second rule not swap with the third.

13.4.2 Guarding Evaluation

We said above that this expands as we expect. Or does it? Let's try the following example:

```
(let ([init #f])
  (my-or-3 (begin (set! init (not init))
                  init)
           #f))
```

Observe that `or` returns the actual value of the first "truthy" value, so the developer can use it in further computations. Therefore, this returns the value of `init`. What do we expect it to be? Naturally, because we've negated the value of `init` once, we expect it to be `#t`. But evaluating it produces `#f`!

This problem is not an artifact of `set!`. If instead of internal mutation we had, say, printed output, the printing would have occurred twice.

To understand why, we have to examine the expanded code. It is this:

```
(let ([init #f])
  (if (begin (set! init (not init))
             init)
      (begin (set! init (not init))
             init)
      #f))
```

Aha! Because we've written the output pattern as

```
#'(if e0
      e0
      ...)
```

This looked entirely benign when we first wrote it, but it illustrates a very important principle when writing macros (or indeed any other program transformation systems): *do not copy code*! In our setting, a syntactic variable should never be repeated; if you need to repeat it in a way that might cause multiple execution of that code, make sure you have considered the consequences of this. Alternatively, if you meant to work with the *value* of the expression, bind it once and use the bound identifier's name subsequently. This is easy to demonstrate:

```
(define-syntax (my-or-4 x)
  (syntax-case x ()
    [(my-or-4)
     #'#f]
    [(my-or-4 e)
```

```
 #'e]
[(my-or-4 e0 e1 ...)
 #'(let ([v e0])
     (if v
         v
         (my-or-4 e1 ...)))]))
```

This pattern of introducing a binding creates a new potential problem: you may end up evaluating expressions that weren't necessary. In fact, it creates a second, even more subtle one: even if it going to be evaluated, you may evaluate it in the wrong context! Therefore, you have to reason carefully about *whether* an expression will be evaluated, and if so, evaluate it once in just the right place, then store that value for subsequent use.

When we repeat our previous example, that contained the `set!`, with `my-or-4`, we see that the result is `#t`, as we would have hoped.

13.4.3 Hygiene

Hopefully now you're nervous about something else.

Do Now!

What?

Consider the macro `(let ([v #t]) (my-or-4 #f v))`. What would we expect this to compute? Naturally, `#t`: the first branch is `#f` but the second is `v`, which is bound to `#t`. But let's look at the expansion:

```
(let ([v #t])
  (let ([v #f])
    (if v
        v
        v)))
```

This expression, when run directly, evaluates to `#f`. However, `(let ([v #t]) (my-or-4 #f v))` evaluates to `#t`. In other words, the macro seems to magically produce the right value: the names of identifiers chosen in the macro seem to be independent of those introduced by the macro! This is unsurprising when it happens in a *function*; the macro expander enjoys a property called *hygiene* that gives it the same property.

One way to think about hygiene is that it effectively automatically renames all bound identifiers. That is, it's as if the program expands as follows:

```
(let ([v #t])
  (or #f v))
```

turns into

```
(let ([v1 #t])
  (or #f v1))
```

(notice the *consistent* renaming of `v` to `v1`), which turns into

```
(let ([v1 #t])
  (let ([v #f])
       v
       v1))
```

which, after renaming, becomes

```
(let ([v1 #t])
  (let ([v2 #f])
       v2
       v1))
```

when expansion terminates. Observe that each of the programs above, if run directly, will produce the correct answer.

13.5 Identifier Capture

Hygienic macros address a routine and important pain that creators of syntactic sugar confront. On rare instances, however, a developer wants to intentionally break hygiene. Returning to objects, consider this input program:

```
(define os-1
  (object/self-1
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

What does the macro look like? Here's an obvious candidate:

```
(define-syntax object/self-1
  (syntax-rules ()
    [(object [mtd-name (var) val] ...)
     (let ([self (lambda (msg-name)
                   (lambda (v) (error 'object "nothing here")))])
       (begin
         (set! self
               (lambda (msg)
                 (case msg
                   [(mtd-name) (lambda (var) val)]
                   ...)))
         self))]))
```

Unfortunately, this macro produces the following error:
`self: unbound identifier in module in: self`
which is referring to the `self` in the body of the method bound to `first`.

Exercise

Work through the hygienic expansion process to understand why error is the expected outcome.

Before we solve this directly, let's consider a variant of the input term that makes the binding explicit:

```
(define os-2
  (object/self-2 self
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

The corresponding macro is a small variation on what we had before:

```
(define-syntax object/self-2
  (syntax-rules ()
    [(object self [mtd-name (var) val] ...)
     (let ([self (lambda (msg-name)
                   (lambda (v) (error 'object "nothing here")))])
       (begin
         (set! self
               (lambda (msg)
                 (case msg
                   [(mtd-name) (lambda (var) val)]
                   ...)))
         self))]))
```

This macro expands without difficulty.

Exercise

Work through the expansion of this version and see what's different.

This offers a critical insight: *had the identifier that goes in the binding position been written by the macro user*, there would have been no problem. Therefore, we want to be able to *pretend* that the introduced identifier was written by the user. The function `datum->syntax` converts the s-expression in its second argument; its first argument is which syntax to pretend it was a part of (in our case, the original macro use, which is bound to `x`). To introduce the result into the environment used for expansion, we use `with-syntax` to bind it in that environment:

```
(define-syntax (object/self-3 x)
  (syntax-case x ()
    [(object [mtd-name (var) val] ...)
     (with-syntax ([self (datum->syntax x 'self)])
       #'(let ([self (lambda (msg-name)
                       (lambda (v) (error 'object "nothing here")))])
           (begin
             (set! self
                   (lambda (msg-name)
```

```
                    (case msg-name
                      [(mtd-name) (lambda (var) val)]
                      ...)))
              self)))]))
```

With this, we can go back to having `self` be implicit:

```
(define os-3
  (object/self-3
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

13.6 Influence on Compiler Design

The use of macros in a language's definition has an impact on all tools, especially compilers. As a working example, consider `let`. `let` has the virtue that it can be compiled efficiently, by just extending the current environment. In contrast, the expansion of `let` into function application results in a much more expensive operation: the creation of a closure and its application to the argument, achieving effectively the same result but at the cost of more time (and often space).

This would seem to be an argument against using the macro. However, a smart compiler recognizes that this pattern occurs often, and instead internally effectively converts left-left-lambda [REF] back into the equivalent of `let`. This has two advantages. First, it means the language designer can freely use macros to obtain a smaller core language, rather than having to trade that off against the execution cost.

It has a second, much subtler, advantage. Because the compiler recognizes this pattern, *other* macros can also exploit it and obtain the same optimization; they don't need to contort their output to insert `let` terms if the left-left-lambda pattern occurs naturally, as they would have to do otherwise. For instance, the left-left-lambda pattern occurs naturally when writing certain kinds of pattern-matchers, but it would take an extra step to convert this into a `let` in the expansion—which is no longer necessary.

13.7 Desugaring in Other Languages

Many modern languages define operations via desugaring, not only Racket. In Python, for instance, iterating using `for` is simply a syntactic pattern. A developer who writes `for x in o` is

- introducing a new identifier (call it `i`—but be sure to not capture any other `i` the programmer has already defined, i.e., bind `i` hygienically!),
- binding it to an iterator obtained from `o`, and
- creating a (potentially) infinite `while` loop that repeatedly invokes the `.next` method of `i` until the iterator raises the `StopIteration` exception.

There are many such patterns in modern programming languages.

14 Control Operations

The term *control* refers to any programming language instruction that causes evaluation to proceed, because it "controls" the program counter of the machine. In that sense, even a simple arithmetic expression should qualify as "control", and operations such as sequential program execution, or function calls and returns, most certainly do. However, in practice we use the term to refer primarily to those operations that cause *non-local* transfer of control, especially beyond that of mere functions and procedures, and the next step up, namely exceptions. We will study such operations in this chapter.

As we study the following control operators, it's worth remembering that even without them, we still have languages that are Turing-complete, and therefore have no more "power". Therefore, what control operators do is change and potentially improve the way we express our intent, and therefore enhance the structure of programs. Thus, it pays to being our study by focusing on program structure.

14.1 Control on the Web

Let us begin our study by examining the structure of Web programs. Consider the following program:

```
(display
  (+ (read-number "First number")
     (read-number "Second number")))
```

Henceforth, we'll call this our "addition server". You should, of course, understand this as a stand-in for more sophisticated applications. For instance, the two prompts might ask for starting and ending points for a trip, and in place of addition we might compute a route or compute airfares. There might even be computation between the two steps: e.g., after entering the first city, the airline might prompt us with choices of where it flies from there.

To test these ideas, here's an implementation of `read-number`:

```
(define (read-number [prompt : string]) : number
  (begin
    (display prompt)
    (let ([v (read)])
      (if (s-exp-number? v)
          (s-exp->number v)
          (read-number prompt)))))
```

When run at the console or in DrRacket, this program prompts us for one number, then another, and then displays their sum.

Now suppose we want to run this on a Web server. We immediately encounter a difficulty: the structure of server-side Web programs is such that they generate a single Web page—such as the one asking for the first number—and then *halt*. As a result, the *rest of the program*—which in this case prompts for the second number, then adds them, and then prints that result, is lost.

Do Now!

Why do Web servers behave in such a strange way?

There are at least two reasons for this behavior: one perhaps historical, and the other technical. The historical reason is that Web servers were initially designed to serve *pages*, i.e., static content. Any program that ran had to generate its output to a

file, from which a server could offer it. Naturally, developers wondered why that same program couldn't run on demand. This made Web content *dynamic*. Terminating the program after generating a single piece of output was the simplest incremental step towards programs, not pages, on the Web.

The more important reason—and the one that has stayed with us—is technical. Imagine our addition server has generated its first prompt. Recall that there is considerable pending computation: the second prompt, the addition, and the display of the result. This computation must suspend waiting for the user's input. If there are millions of users, then millions of computations must be suspended, creating an enormous performance problem. Furthermore, suppose a user does not actually complete the computation—analogous to searching at an on-line bookstore or airline site, but not completing the purchase. How does the server know when or even whether to terminate the computation? And until it does, the resources associated with that computation remain in use.

Conceptually, therefore, the Web protocol was designed to be *stateless*: it would not store state on the server associated with intermediate computations. Instead, Web program developers would be forced to maintain all necessary state elsewhere, and each request would need to be able to resume the computation in full. In practice, the Web has not proven to be stateless at all, but it still hews largely in this direction, and studying the structure of such programs is very instructive.

Now consider client-side Web programs: those that run inside the browser, written in or compiled to JavaScript. Suppose such a computation needs to communicate with a server. The primitive for this is called `XMLHttpRequest`. The user makes an instance of this primitive and invokes its `send` method to send a message to the server. Communicating with a server is not, however, instantaneous (and indeed may never complete, depending on the state of the network). This leaves the sending process suspended.

The designers of JavaScript decided to make the language *single-threaded*: i.e., there would be only one thread of execution at a time. This avoids the various perils that arise from combining mutation with threads. As a result, however, the JavaScript process locks up awaiting the response, and nothing else can happen: e.g., other handlers on the page no longer respond.

Due to the structuring problems this causes, there are now various proposals to, in effect, add "safe" threads to JavaScript. The ideas described in this chapter can be viewed as an alternative that offer similar structuring benefits.

To avoid this problem, the design of `XMLHttpRequest` demands that the developer provide a procedure that responds to the request if and when it arrives. This callback procedure is registered with the system. It needs to embody the *rest of the processing* of that request. Thus, for entirely different reasons—not performance, but avoiding the problems of synchronization, non-atomicity, and deadlocks—the client-side Web has evolved to demand the same pattern of developers. Let us now better understand that pattern.

14.1.1 Program Decomposition into Now and Later

Let us consider what it takes to make our above program work in a stateless setting, such as on a Web server. First we have to determine the *first* interaction. This is the prompt for the first number, because Racket evaluates arguments from left to right. It is instructive to divide the program into two parts: what happens to generate the first interaction (which can all run now), and what needs to happen after it (which must be

"remembered" somehow). The former is easy:

```
(read-number "First number")
```

We've already explained in prose what's left, but now it's time to write it *as a program*. It seems to be something like

```
(display
  (+ <the result from the first interaction>
     (read-number "Second number")))
```

We're intentionally ignoring `read-number` for now, but we'll return to it. For now, let's pretend it's built-in.

A Web server can't execute the above, however, because it evidently isn't a *program*. We instead need some way of writing this as one.

Let's observe a few characteristics of this computation:

- It needs to be a legitimate program.
- It needs to stay suspended until the request comes in.
- It needs a way—such as a parameter—to refer to the value from the first interaction.

Put together these characteristics and we have a clear representation—a *function*:

```
(lambda (v1)
  (display
    (+ v1
       (read-number "Second number"))))
```

14.1.2 A Partial Solution

On the Web, there is an additional wrinkle: each Web page with input elements needs to refer to a program stored on the Web, which will receive the data from the form and process it. This program is named in the `action` field of a form. Thus, imagine that the server generates a fresh label, stores the above function in a table associated with that label, and refers to the table in the `action` field. If and when the client actually submits the form, the server extracts the associated function, supplies it with the form's values, and thus resumes execution.

Do Now!

Is the solution above stateless?

Let's imagine that we have a custom Web server that maintains the above table. In such a server, we might have a special version of `read-number`—call it `read-number/suspend`—that records the rest of the program:

```
(read-number/suspend "First number"
                     (lambda (v1)
                       (display
                        (+ v1
                           (read-number "Second number")))))
```

To test this, let's implement such a procedure. First, we need a representation for labels; numbers are an easy substitute:

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

Let's say `new-label` generates a fresh label on each invocation.

Exercise

Define `new-label`. You might use `new-loc` for inspiration.

We need a table to store the procedures representing the rest of the program.

```
(define table (make-hash empty))
```

Now we can store these procedures:

```
(define (read-number/suspend [prompt : string] rest)
  (let ([g (new-label)])
    (begin
      (hash-set! table g rest)
      (display prompt)
      (display " To enter it, use the action field label ")
      (display g))))
```

If we now run the above invocation of `read-number/suspend`, the system prints
`First number To enter it, use the action field label 1`
This is tantamount to printing the prompt in a Web page, and putting the label `1` in the `action` field. Because we're simulating it, we need something to represent the browser's submission process. This needs both the label (from the `action` field) and the value entered in the form. Given these two values, this procedure needs to extract the relevant procedure from the table, and apply it to the form value.

```
(define (resume [g : label] [n : number])
  ((some-v (hash-ref table g)) n))
```

With this, we can now simulate the act of entering `3` and clicking on a "Submit" button by running:
`> (resume 1 3)`
where `1` is the label and `3` is the user's input. Unfortunately, this simply produces another prompt, because we haven't fully converted the program. If we delete `read-number`, we're forced to convert the entire program:

```
(read-number/suspend "First number"
                     (lambda (v1)
                       (read-number/suspend "Second number"
                                            (lambda (v2)
                                              (display
                                               (+ v1 v2))))))
```

Just to be safe, we can also make sure the computation terminates after each output by adding an `error` invocation at the end of `read-number/suspend` (to truly ensure the most extreme form of "suspension").

When we execute this program, we have to use `resume` twice:

```
First number To enter it, use the action field label 1
halting: Program shut down
> (resume 1 3)
Second number To enter it, use the action field label 2
halting: Program shut down
> (resume 2 10)
13
```

where the two user inputs are `3` and `10`, giving a total of `13`, and the `halting` messages are generated by the `error` command we inserted.

We've purposely played a little coy with the types of the interesting parts of our program. Let's examine what these types should be. The second argument to `read-number/suspend` needs to be a procedure that consumes numbers and returns whatever the computation eventually produces: `(number -> 'a)`. Similarly, the return type of `resume` is the same `'a`. How do these `'a`s communicate with one another? This is done by `table`, which maps labels to `(number -> 'a)`. That is, at every step the computation makes progress towards the same outcome. `read-number/suspend` writes into this table, and `resume` reads from it.

14.1.3 Achieving Statelessness

We haven't actually achieved statelessness yet, because we have this large table residing on the server, with no clear means to remove entries from it. It would be better if we could avoid the server state entirely. This means we have to move the relevant state to the client.

There are actually two ways in which the server holds state. One is that we have reserved the right to create as many entries in the hash table as we wish, rather than a constant number (i.e., linear in the size of the program itself). The other is what we're storing in the table: honest-to-goodness closures, which might be holding on to an arbitrary amount of state. We'll see this more clearly soon.

Let's start by eliminating the closure. Instead, let's have each of the funtion arguments to be named, top-level functions (which immediately forces us to have only a fixed number of them, because the program's size cannot be unbounded):

```
(read-number/stateless "First number" prog1)

(define (prog1 v1)
  (read-number/stateless "Second number" prog2))

(define (prog2 v2)
  (display (+ v1 v2)))
```

Observe how each code block refers only to the *name* of the next, rather than to a real closure. The value of the argument comes from the form. There's just one problem: `v1` in `prog2` is a free identifier!

The way to fix this problem is, instead of creating a closure after one step, to send `v1` to the client to be stored there. Where do we store this? The browser offers two mechanisms for doing this: *cookies* and *hidden fields*. Which one do we use?

14.1.4 Interaction with State

The fundamental difference between cookies and hidden fields is that *all pages share the same cookie, but each page has its own hidden fields*.

First, let's consider a sequence of interactions with the existing program that uses `read-number/suspend` (at both interaction points). It looks like this:

```
First number To enter it, use the action field label 1
> (resume 1 3)
Second number To enter it, use the action field label 2
> (resume 2 10)
13
```

Thus, resuming with label `2` appears to represent adding `3` to the given argument (i.e., form field value). To be sure,

```
> (resume 2 15)
18
```

So far, so good. Now suppose we use label `1` again:

```
> (resume 1 5)
Second number To enter it, use the action field label 3
```

Observe that this new program execution needs to be resumed by using label `3`, not `1`. Indeed,

```
> (resume 3 10)
15
```

But we ought to ask, what happens if we reuse label `2`?

Do Now!

> Try `(resume 2 10)`.

Doing this is tantamount to resuming the old computation. We therefore expect it produce the same answer as before:

```
> (resume 2 10)
13
```

Now let's create a stateful implementation. We can simulate this by observing that each closure has its own environment, but all closures share the same mutable state. We can simulate this using our existing `read-number/suspend` by making sure we don't rely on the closure behavior of `lambda`, i.e., by not having any free identifiers in the body.

```
(define cookie '-100)

(read-number/suspend "First number"
```

```
(lambda (v1)
  (begin
    (set! cookie v1)
    (read-number/suspend "Second number"
                         (lambda (v2)
                           (display
                            (+ cookie v2)))))))
```

Exercise

What do we *expect* for the same sequence as before?

Do Now!

What happens?

Initially, nothing seems different:

```
First number To enter it, use the action field label 1
> (resume 1 3)
Second number To enter it, use the action field label 2
> (resume 2 10)
13
```

When we reuse the initial computation, we indeed get a new resumption label:

```
> (resume 1 5)
Second number To enter it, use the action field label 3
```

which, when used, computes what we'd expect:

```
> (resume 3 10)
15
```

Now we come to the critical step:

```
> (resume 2 10)
15
```

It is unsurprising that the two resumptions of label 2 would produce different answers, given that they rely on mutable state. The reason it's problematic is because of what happens when we translate the same behavior to the Web.

Imagine visiting a hotel reservation Web site and searching for hotels in a city. In return, you are shown a list of hotels and the label 1. You explore one of them in a new tab or window; this produces information on that hotel, and label 2 to make the reservation. You decide, however, to return to the hotel listing and explore another hotel in a fresh tab or window. This produces the second hotel's information, with label 3 to reserve at that hotel. You decide, however, to choose the first hotel, return to the first hotel's page, and choose its reservation button—i.e., submit label 2. Which hotel did you expect to be booked into? Though you expected a reservation at the *first* hotel, on most travel sites, this will either reserve at the *second* hotel—i.e., the one you last viewed, but not the one on the page whose reservation button you clicked—or produce an error. This is because of the pervasive use of cookies on Web sites, a practice encouraged by most Web APIs.

14.2 Continuation-Passing Style

The functions we've been writing have a name. Though we've presented ideas in terms of the Web, we're relying on a much older idea: the functions are called *continuations*, and this style of programs is called *continuation-passing style* (CPS). This is worth studying in its own right, because it is the basis for studying a variety of other non-trivial control operations—such as generators.

We will take the liberty of using CPS as both a noun and verb: a particular structure of code and the process that converts code into it.

Earlier, we converted programs so that no Web input operation was nested inside another. The motivation was simple: when the program terminates, all nested computations are lost. A similar argument applies, in a more local sense, in the case of `XMLHttpRequest`: any computation depending on the result of a response from a Web server needs to reside in the callback associated with the request to the server.

In fact, we don't need to transform *every* expression. We only care about expressions that involve actual Web interaction. For example, if we computed a more complex mathematical expression than just addition, we wouldn't need to transform it. If, however, we had a function call, we'd either have to be absolutely certain the function didn't have any Web invocations either inside it, or in the functions in invokes, or the ones *they* invoke...or else, to be defensive, we should transform them all. Therefore, we have to transform every expression that we can't be sure performs no Web interactions.

The heart of our transformation is therefore to turn every one-argument function, `f`, into one with an extra argument. This extra argument is the continuation, which represents the rest of the computation. The continuation is itself a function of one argument. This argument takes the value that *would have been returned* by `f` and passes it to the rest of the computation. `f`, instead of *returning* a value, instead *passes* the value it would have returned to its continuation.

CPS is a general transformation, which we can apply to any program. Because it's a program transformation, we can think of it as a special kind of desugaring: in particular, instead of transforming programs from a larger language to a smaller one (as macros do), or from one language to entirely another (as compilers do), it transforms programs *within* the same language: from the full language to a more restricted version that obeys the pattern we've been discussing. As a result, we can reuse an evaluator for the full language to also evaluate programs in the CPS subset.

14.2.1 Implementation by Desugaring

Because we already have good support for desugaring, let's use to define the CPS transform. Concretely, we'll implement a CPS macro [REF]. To more cleanly separate the source language from the target, we'll use slightly different names for most of the language constructs: a one-armed `with` and `rec` instead of `let` and `letrec`; `lam` instead of `lambda`; `cnd` instead of `if`; `seq` for `begin`; and `set` for `set!`. We'll also give ourselves a sufficiently rich language to write some interesting programs!

<*cps-macro*> **::=**

```
(define-syntax (cps e)
  (syntax-case e (with rec lam cnd seq set quote display read-number)
    <cps-macro-with-case>
    <cps-macro-rec-case>
```

The presentation that follows orders the cases of the macro from what I believe are easiest to hardest. However, the code in the macro must avoid non-overlapping patterns, and hence follows a diffent order.

<cps-macro-lam-case>
<cps-macro-cnd-case>
<cps-macro-display-case>
<cps-macro-read-number-case>
<cps-macro-seq-case>
<cps-macro-set-case>
<cps-macro-quote-case>
<cps-macro-app-1-case>
<cps-macro-app-2-case>
<cps-macro-atomic-case>))

Our representation in CPS will be to turn *every* expression into a procedure of one argument, the continuation. The converted expression will eventually either supply a value to the continuation or will pass the continuation on to some other expression that will—by preserving this invariant inductively—supply it with a value. Thus, all output from CPS will look like `(lambda (k) ...)` (and we will rely on hygiene [REF] to keep all these introduced `k`'s from clashing with one another).

First let's dispatch with the easy case, which is atomic values. Though conceptually easiest, we have written this last because otherwise this pattern would shadow all the other cases. (Ideally, we should have written it first and provided a guard expression that precisely defines the syntactic cases we want to treat as atomic. We're playing loose here becuase our focus is on more interesting cases.) In the atomic case, we already have a value, so we simply need to supply it to the continuation:

<***cps-macro-atomic-case***> **::=**

```
[(_ atomic)
 #'(lambda (k)
     (k atomic))]
```

Similarly for quoted constants:

<***cps-macro-quote-case***> **::=**

```
[(_ 'e)
 #'(lambda (k) (k 'e))]
```

Also, we already know, from [REF] and [REF], that we can treat `with` and `rec` as macros, respectively:

<***cps-macro-with-case***> **::=**

```
[(_ (with (v e) b))
 #'(cps ((lam (v) b) e))]
```

<***cps-macro-rec-case***> **::=**

```
[(_ (rec (v f) b))
 #'(cps (with (v (lam (arg) (error 'dummy "nothing")))
              (seq
               (set v f)
               b)))]
```

Mutation is easy: we have to evaluate the new value, and then perform the actual update:

<*cps-macro-set-case*> **::=**

```
[(_ (set v e))
 #'(lambda (k)
     ((cps e) (lambda (ev)
                (k (set! v ev)))))]
```

Sequencing is also straightforward: we perform each operation in turn. Observe how this preserves the semantics of sequencing: not only does it obey the order of operations, the value of the first sub-term (`e1`) is not mentioned anywhere in the body of the second (`e2`), so the name given to the identifier holding its value is irrelevant.

<*cps-macro-seq-case*> **::=**

```
[(_ (seq e1 e2))
 #'(lambda (k)
     ((cps e1) (lambda (_)
                 ((cps e2) k))))]
```

When handling conditionals, we need to create a new continuation to remember that we are waiting for the test expression to evaluate. Once we have its value, however, we can dispatch on the result and return to the existing continuations:

<*cps-macro-cnd-case*> **::=**

```
[(_ (cnd tst thn els))
 #'(lambda (k)
     ((cps tst) (lambda (tstv)
                  (if tstv
                      ((cps thn) k)
                      ((cps els) k)))))]
```

When we get to applications, we have two cases to consider. We absolutely need to handle the treatment of procedures created in the language: those with one argument. For the purposes of writing example programs, however, it is useful to be able to employ primitives such as `+` and `*`. Thus, we will *assume for simplicity* that one-argument procedures are written by the user, and hence need conversion to CPS, while two-argument ones are primitives that will not perform any Web or other control operations and hence can be invoked directly; we will *also* assume that the primitive will be written in-line (i.e., the application position will not be a complex expression that can itself, say, perform a Web interaction).

For an application we have to evaluate both the function and argument expressions. Once we've obtained these, we are ready to apply the function. Therefore, it is tempting to write

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

```
[(_ (f a))
```

```
#'(lambda (k)
    ((cps f) (lambda (fv)
               ((cps a) (lambda (av)
                          (k (fv av))))))))]
```

Do Now!

Do you see why this is wrong?

The problem is that, though the function is now a value, that value is a closure with a potentially complicated body: evaluating the body can, for example, result in further Web interactions, at which point the rest of the function's body, as well as the pending `(k ...)` (i.e., the rest of the program), will all be lost. To avoid this, we have to supply `k` to the function's value, and let the inductive invariant ensure that `k` will eventually be invoked with the value of applying `fv` to `av`:

<*cps-macro-app-1-case*> **::=**

```
[(_ (f a))
 #'(lambda (k)
     ((cps f) (lambda (fv)
                ((cps a) (lambda (av)
                           (fv av k))))))]
```

Treating the special case of built-in binary operations is easier:

<*cps-macro-app-2-case*> **::=**

```
[(_ (f a b))
 #'(lambda (k)
     ((cps a) (lambda (av)
                ((cps b) (lambda (bv)
                           (k (f av bv))))))))]
```

The very pattern we could not use for user-defined procedures we employ here, because we assume that the application of `f` will always return without any unusual transfers of control.

A function is itself a value, so it should be returned to the pending computation. The application case above, however, shows that we have to transform functions to take an extra argument, namely the continuation at the point of invocation. This leaves us with a quandary: which continuation do we supply to the body?

<*cps-macro-lam-case-take-1*> **::=**

```
[(_ (lam (a) b))
 (identifier? #'a)
 #'(lambda (k)
     (k (lambda (a dyn-k)
          ((cps b) ...)))))]
```

That is, in place of `...`, which continuation do we supply: `k` or `dyn-k`?

Do Now!

Which continuation should we supply?

The former is the continuation *at the point of closure creation*. The latter is the continuation *at the point of closure invocation*. In other words, the former is "static" and the latter is "dynamic". In this case, we need to use the dynamic continuation, otherwise something very strange would happen: the program would return to the point where the closure was created, rather than where it is being used! This would result in seemingly very strange program behavior, so we wish to avoid it. Observe that we are consciously choosing the dynamic continuation just as, where scope was concerned, we chose the static environment.

<***cps-macro-lam-case***> **::=**

```
[(_ (lam (a) b))
 (identifier? #'a)
 #'(lambda (k)
     (k (lambda (a dyn-k)
          ((cps b) dyn-k))))]
```

Finally, for the purpose of modeling Web programming, we can add our input and output procedures. Output follows the application pattern we've already seen:

<***cps-macro-display-case***> **::=**

```
[(_ (display output))
 #'(lambda (k)
     ((cps output) (lambda (ov)
                     (k (display ov)))))]
```

Finally, for input, we can use the pre-existing `read-number/suspend`, but this time *generate* its uses rather than force the programmer to construct them:

<***cps-macro-read-number-case***> **::=**

```
[(_ (read-number prompt))
 #'(lambda (k)
     ((cps prompt) (lambda (pv)
                     (read-number/suspend pv k))))]
```

Notice that the continuation bound to `k` is precisely the continuation that we need to stash at the point of a Web interaction.

Testing any code converted to CPS is slightly annoying because all CPS terms expect a continuation. The initial continuation is one that simply either (a) consumes a value and returns it, or (b) consumes a value and prints it, or (c) consumes a value, prints it, and gets ready for another computation (as the prompt in the DrRacket Interactions window does). All three of these are effectively just the identity function in various guises. Thus, the following definition is helpful for testing:

```
(define (run c) (c identity))
```

For instance,

```
(test (run (cps 3))                              3)
(test (run (cps ((lam ()    5)       )))       5)
(test (run (cps ((lam (x)   (* x x)) 5)))      25)
(test (run (cps (+ 5 ((lam (x) (* x x)) 5)))) 30)
```

We can also test our old Web program:

```
(run (cps (display (+ (read-number "First")
                      (read-number "Second")))))
```

Lest you get lost in the myriad of code, let me highlight the important lesson here: *We've recovered our code structure*. That is, we can write the program in *direct style*, with properly nested expressions, and a compiler—in this case, the CPS converter—takes care of making it work with a suitable underlying API. This is what good programming languages ought to do!

14.2.2 Converting the Example

Let's consider the example above and see what it converts to. You can either do this by hand, or take the easy way out and employ the Macro Stepper of DrRacket. Assuming we include the application to `identity` contained in `run`, we get:

For now, you need to put the code in `#lang racket` to get the full force of the Macro Stepper.

```
(lambda (k)
  ((lambda (k)
     ((lambda (k)
        ((lambda (k)
           (k "First")) (lambda (pv)
                          (read-number/suspend pv k))))
      (lambda (lv)
        ((lambda (k)
           ((lambda (k)
              (k "Second")) (lambda (pv)
                              (read-number/suspend pv k))))
         (lambda (rv)
           (k (+ lv rv)))))))
   (lambda (ov)
     (k (display ov)))))
```

What! This isn't at all the version we wrote by hand!

In fact, this program is full of so-called *administrative* `lambda`s that were introduced by the particular CPS algorithm we used. Fear not! If we stepwise apply each of these `lambda`s and substitute, however—

Designing better CPS algorithms, that eliminate needless administrative `lambda`s, is therefore an ongoing and open research question.

Do Now!

Do it!

—the program reduces to

```
(read-number/suspend "First"
                     (lambda (lv)
                       (read-number/suspend "Second"
                                            (lambda (rv)
                                              (identity
                                               (display (+ lv rv)))))))
```

which is precisely what we wanted.

14.2.3 Implementation in the Core

Now that we've seen how CPS can be implemented through desugaring, we should ask whether it can be put in the core instead.

Recall that we've said that CPS applies to all programs. We have one program we are especially interested in: the interpreter. Sure enough, we can apply the CPS transformation to it, making available what are effectively the same continuations.

First, we'll find it convenient to use a procedural representation of closures [REF]. We'll have the interpreter take an extra argument, which consumes values (those given to the continuation) and eventually returns them:

<***cps-interp***> **::=**

```
(define (interp/k [expr : ExprC] [env : Env] [k : (Value -> Value)]) : Value
  <cps-interp-body>)
```

In the easy cases, instead of returning a value we need to simply pass it to the continuation argument:

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

```
(type-case ExprC expr
  [numC (n) (k (numV n))]
  [idC (n) (k (lookup n env))]
  <cps-interp-plusC-case>
  <cps-interp-appC-case>
  <cps-interp-lamC-case>)
```

(Note that `multC` is handled entirely analogous to `plusC`.)

Let's start with the easy case, `plusC`. First we interpret the left sub-expression. The continuation for this evaluation interprets the right sub-expression. The continuation for that adds the result. What should happen to the result of addition? In `interp`, it was returned to whichever computation caused the `plusC` to be interpreted. Now, remember, we no longer return values; instead we pass them to the continuation:

<***cps-interp-plusC-case***> **::=**

```
[plusC (l r) (interp/k l env
                       (lambda (lv)
                         (interp/k r env
                                   (lambda (rv)
                                     (k (num+ lv rv))))))]
```

Exercise

Implement the code for `multC`.

This leaves the two difficult, and related, pieces.

In an application, we again have to interpret the two sub-expressions, and then apply the resulting closure to the argument. But we've already agreed that every application needs a continuation argument. Therefore, we have to update our definition of a value:

```
(define-type Value
  [numV (n : number)]
  [closV (f : (Value (Value -> Value) -> Value))])
```

Now we have to decide what continuation to pass. In an application, it's the continuation given to the interpreter:

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

```
[appC (f a) (interp/k f env
                      (lambda (fv)
                        (interp/k a env
                                  (lambda (av)
                                    ((closV-f fv) av k)))))]
```

Finally, the `lamC` case. We have to create a `closV` using a `lambda`, as before. However, this procedure needs to take two arguments: the actual value of the argument, and the continuation of the application. The critical question is, what is this latter value?

We have essentially two choices. `k` represents the *static* continuation: the one active at the point of closure *construction*. However, what we want is the continuation at the point of closure *invocation*: the *dynamic* continuation.

<***cps-interp-lamC-case***> **::=**

```
[lamC (a b) (k (closV (lambda (arg-val dyn-k)
                        (interp/k b
                                  (extend-env (bind a arg-val)
                                              env)
                                  dyn-k))))]
```

To test this revised interpreter, we need to invoke `interp/k` with some kind of initial continuation value. This needs to be a procedure that represents nothing remaining in the computation. A natural representation for this is the identity function:

```
(define (interp [expr : ExprC]) : Value
  (interp/k expr mt-env
            (lambda (ans)
              ans)))
```

To signify that this is strictly a top-level interface to `interp/k`, we've dropped the environment parameter and pass the empty environment automatically. If we want to be especially sure we haven't accidentally used this procedure recursively, we could insert a call to `error` at its end to prevent it from returning and its return value being used.

14.3 Generators

Many programming languages now have a notion of *generators*. A generator is like a procedure, in that one can invoke it in an application. Whereas a regular procedure always begins execution at the beginning, a generator *resumes* from where it last left off. Of course, that means a generator needs a notion of "exiting before it's done". This is known as *yielding*, namely returning control to whatever called it.

14.3.1 Design Variations

There are many variations between generators. The points of variation, predictably, have to do with how to enter and exit a generator:

- In some languages a generator is an object that is instantiated like any other object, and its execution is resumed by invoking a method (such as `next` in Python). In others it is just like a procedure, and indeed it is re-entered by applying it like a function.

- In some languages the yielding operation—such as Python's `yield`—is available only inside the syntactic body of the generator. In others, such as Racket, `yield` is an applicable value bound in the body, but by virtue of being a value, it can be passed to abstractions, stored in data structures, and so on.

In languages where values in addition to regular procedures can be used in an application, all such values are collectively called *applicables*.

Python's design represents an extreme point in that a generator is simply *any function that contains the keyword `yield` in its body*. In addition, Python's `yield` cannot be passed as a parameter to another function that performs the yielding on behalf of the generator.

There is also a small issue of naming. In many languages with generators, the yielder is *automatically* called word `yield`: either as a keyword (as in Python) or as an identifier bound to an applicable value (as in Racket). Another possibility is that the user of the generator must indicate in the generator expression what name to give the yielder. That is, a use might look like

```
(generator (yield) (from)
           (rec (f (lam (n)
                     (seq
                       (yield n)
                       (f (+ n 1)))))
             (f from)))
```

Curiously, Python expects users to determine what to call `self` or `this` in objects, but it does not provide the same flexibility for `yield`, because it has no other way to determine which functions are generators!

but it might equivalently be

```
(generator (y) (from)
           (rec (f (lam (n)
                     (seq
                       (y n)
                       (f (+ n 1)))))
             (f from)))
```

and if the yielder is an actual value, a user can also abstract over yielding:

```
(generator (y) (from)
           (rec (f (lam (n)
                     (seq
                       ((yield-helper y) n)
                       (f (+ n 1)))))
             (f from)))
```

where `yield-helper` will presumably perform the actual yielding.

There are actually two more design decisions:

1. Is `yield` a statement or expression? In many languages it is actually an expression, meaning it has a value: the one supplied when resuming the generator. This makes the generator more flexible because the user of a generator can use the parameter(s) to alter the generator's behavior, rather than being *forced* to use state to communicate desired changes.

2. What happens at the end of the generator's execution? In many languages, a generator raises an exception to signal its completion.

14.3.2 Implementing Generators

To implement generators, it will be especially useful to employ our CPS macro language. Let's first decide where we stand regarding the above design decisions. We will use the applicative representation of generators: that is, asking for the next value from the generator is done by applying it to any necessary arguments. Similarly, the yielder will also be an applicable value and will in turn be an expression. Though we have already seen how macros can automatically capture a name [REF], let's make the yielder's name explicit to keep the macro simpler. Finally, we'll raise an error when the generator is done executing.

How do generators work? To yield, a generator must

- remember where in its execution it currently is, and
- know where in its caller it should return to.

while, when invoked, it should

- remember where in its execution its caller currently is, and
- know where in its body it should return to.

Observe the duality between invocation and yielding.

As you might guess, these "where"s correspond to continuations.

Let's build up the generator rule of the `cps` macro incrementally. First a header pattern:

<*cps-macro-generator-case*> **::=**

```
[(_ (generator (yield) (v) b))
 (and (identifier? #'v) (identifier? #'yield))
 <generator-body>]
```

The beginning of the body is easy: all code in CPS needs to consume a continuation, and because a generator is a value, this value should be supplied to the continuation:

<*generator-body*> **::=**

```
#'(lambda (k)
    (k <generator-value>))
```

Now we're ready to tackle the heart of the generator.

Recall that a generator is an applicable value. That means it can occur in an application position, and must therefore have the same "interface" as a procedure: a procedure of two arguments, the first a value and the second the continuation at the point of application. What should this procedure do? We've described just this above. First the generator must remember where the caller is in its execution, which is precisely the continuation at the point of application; "remember" here most simply means "must be stored in state". Then, the generator should return to where it previously was, i.e., its *own* continuation, which must clearly have been stored. Therefore the core of the applicable value is:

<*generator-core*> **::=**

```
(lambda (v dyn-k)
  (begin
    (set! where-to-go dyn-k)
    (resumer v)))
```

Here, `where-to-go` records the continuation of the caller, to resume it upon yielding; `resumer` is the local continuation of the generator. Let's think about what their initial values must be:

- `where-to-go` has no initial value (because the generator has yet to be invoked), so it needs to throw an error if ever used. Fortunately this error will never occur, because `where-to-go` is mutated on the first entry into the generator, so the error is just a safeguard against bugs in the implementation.
- Initially, the rest of the generator is the whole generator, so `resumer` should be bound to the (CPS of) `b`. What is its continuation? This is the continuation of the entire generator, i.e., what to do when the generator finishes. We've agreed that this should also signal an error (except in this case the error truly can occur, in case the generator is asked to produce more values than it's equipped to).

We still need to bind `yield`. It is, as we've pointed out, symmetric to generator resumption: save the local continuation in `resumer` and return by applying `where-to-go`.

Putting together these pieces, we get:

<*generator-value*> **::=**

```
(let ([where-to-go (lambda (v) (error 'where-to-go "nothing"))])
  (letrec([resumer (lambda (v)
                     ((cps b) (lambda (k)
                                (error 'generator "fell through"))))]
          [yield (lambda (v gen-k)
                   (begin
                     (set! resumer gen-k)
                     (where-to-go v)))])
    <generator-core>))
```

Do Now!

Why this pattern of `let` and `letrec` instead of `let`?

Observe the dependencies between these code fragments. `where-to-go` doesn't depend on either of `resumer` or `yield`. `yield` clearly depends on both `where-to-go` and `resumer`. But why are `resumer` and `yield` mutually referential?

Do Now!

Try the alternative!

The subtle dependency you may be missing is that `resumer` contains `b`, the body of the generator, which may contain references to `yield`. Therefore, it needs to be closed over the binding of the yielder.

Exercise

How do generators differ from coroutines and threads? Implement coroutines and threads using a similar strategy.

14.4 Continuations and Stacks

Surprising as it may seem, CPS conversion actually provides tremendous insight into the nature of the program execution *stack*. The first thing to understand is that every continuation is actually *the stack itself*. This might seem odd, given that stacks are low-level machine primitives while continuations are seemingly complex procedures. But what is the stack, really?

- It's a record of what remains to be done in the computation. So is the continuation.
- It's traditionally thought of as a list of stack *frames*. That is, each frame has a reference to the frames remaining after it finishes. Similarly, each continuation is a small procedure that refers to—and hence closes over—its own continuation.

If we had chosen a different representation for program instructions, combining this with the data structure representation of closures, we would obtain a continuation representation that is essentially the same as the machine stack.

- Each stack frame also stores procedure parameters. This is implicitly managed by the procedural representation of continuations, whereas this was done explicitly in the data stucture representation (using `bind`).
- Each frame also has space for "local variables". In principle so does the continuation, though by using the macro implementation of local binding, we've effectively reduced everything to procedure parameters. Conceptually, however, some of these are "true" procedure parameters while others are local bindings turned into procedure parameters by a macro.
- The stack has references to, but does not close over, the heap. Thus changes to the heap are visible across stack frames. In precisely the same way, closures refer to, but do not close over, the store, so changes to the store are visible across closures.

Therefore, traditionally the stack is responsible for maintaining lexical scope, which we get automatically because we are using closures in a statically-scoped language.

Now we can study the conversion of various terms to understand the mapping to stacks. For instance, consider the conversion of a function application [REF]:

```
[(_ (f a))
 #'(lambda (k)
     ((cps f) (lambda (fv)
                ((cps a) (lambda (av)
                           (fv av k))))))]
```

How do we "read" this? As follows:

- Let's use `k` to refer to the stack present before the function application begins to evaluate.
- When we begin to evaluate the function position (`f`), create a new stack frame (`(lambda (fv) ...)`). This frame has one free identifier: `k`. Thus its closure needs to record one element of the environment, namely the rest of the stack.
- The code portion of the stack frame represents what is left to be done once we obtain a value for the function: evaluate the argument, and perform the application, and return the result to the stack expecting the result of the application: `k`.
- When evaluation of `f` completes, we begin to evaluate `a`, which also creates a stack frame: `(lambda (av) ...)`. This frame has *two* free identifiers: `k` and `fv`. This tells us:
  - We no longer need the stack frame for evaluating the function position, but

  - we now need a *temporary* that records the value—hopefully a function value—of evaluating the function position.

- The code portion of this second frame also represents what is left to be done: invoke the function value with the argument, in the stack expecting the value of the application.

Let us apply similar reasoning to conditionals:

```
[(_ (cnd tst thn els))
 #'(lambda (k)
     ((cps tst) (lambda (tstv)
                  (if tstv
                      ((cps thn) k)
                      ((cps els) k)))))]
```

It says that to evaluate the conditional expression we have to create a new stack frame. This frame closes over the stack expecting the value of the entire conditional. This frame makes a decision based on the value of the conditional expression, and invokes one of the other expressions. Once we have examined this value the frame created to evaluate the conditional expression is no longer necessary, so evaluation can proceed in `k`.

Viewed through this lens, we can more easily provide an operational explanation for generators. Each generator has its own private stack, and when execution attempts to return past its end, our implementation raises an error. On invocation, a generator stores a reference to the stack of the "rest of the program" in `where-to-go`, and resumes its own stack, which is referred to by `resumer`. On yielding, the system swaps references to stacks. Coroutines, threads, and generators are all conceptually similar: they are all mechanisms to create "many little stacks" instead of having a single, global stack.

14.5 Tail Calls

Observe that the stack patterns above add a frame to the current stack, perform some evaluation, and eventually always return to the current stack. In particular, observe that in an application, we need stack space to evaluate the function position and then the arguments, but once all these are evaluated, we resume computation using the stack we started out with before the application. In other words, *function calls do not themselves need to consume stack space*: we only need space to compute the arguments.

However, not all languages observe or respect this property. In languages that do, programmers can use *recursion* to obtain *iterative behavior*: i.e., a sequence of function calls can consume no more stack space than no function calls at all. This removes the need to create special looping constructs; indeed, loops can simply be expressed as a syntactic sugar.

Of course, this property does not apply in general. If a call to `f` is performed to compute an argument to a call to `g`, the call to `f` is still consuming space relative to the context surrounding `g`. Thus, we should really speak of a relationship between expressions: one expression is in *tail position* relative to another if its evaluation requires no

additional stack space beyond the other. In our CPS macro, every expression that uses `k` as its continuation—such as a function application after all the sub-expressions have been evaluated, or the then- and else-branches of a conditional—are all in tail position relative to the enclosing application (and perhaps recursively further up). In contrast, every expression that has to create a new stack frame is not in tail position.

Some languages have special support for tail *recursion*: when a procedure calls itself in tail position relative to its body. This is obviously useful, because it enables recursion to efficiently implement loops. However, it hurts "loops" that cannot be squeezed into a single recursive function. For instance, when implementing a scanner or other state machine, it is most convenient to have a set of functions each representing one state, and transitioning to other states by making (tail) function calls. It is onerous (and misses the point) to turn these into a single recursive function. If, however, a language recognizes tail calls as such, it can optimize these cross-function calls just as much as it does intra-function ones.

Racket, in particular, promises to implement tail calls without allocating additional stack space. Though some people refer to this as "tail call optimization", this term is misleading: an optimization is optional, whereas whether or not a language promises to properly implement tail calls is a *semantic* feature. Developers need to know how the language will behave because it affects how they program.

Because of this feature, observe something interesting about the program after CPS transformation: all of its function applications are themselves tail calls! You can see this starting with the `read-number/suspend` example that began this chapter: any pending computation was put into the continuation argument. Assuming the program might terminate at any call is tantamount to not using any stack space at all (because the stack would get wiped out).

Exercise

How is the program able to function in the absence of a stack?

14.6 Continuations as a Language Feature

With this insight into the connection between continuations and stacks, we can now return to the treatment of procedures: we ignored the continuation at the point of closure *creation* and instead only used the one at the point of closure *invocation*. This of course corresponds to normal procedure behavior. But now we can ask, what if we use the creation-time continuation instead? This would correspond to maintaining a reference to (a copy of) the stack at the point of "procedure" creation, and when the procedure is applied, ignoring the dynamic evaluation and going back to the point of procedure creation.

In principle, we are trying to leave `lambda` intact and instead give ourselves a language construct that corresponds to this behavior:

cc = "current continuation"

*<**cps-macro-let/cc-case**>* **::=**

```
[(_ (let/cc kont b))
 (identifier? #'kont)
 #'(lambda (k)
```

```
    (let ([kont (lambda (v dyn-k)
                  (k v))])
      ((cps b) k)))]
```

What this says is that either way, control will return to the expression that immediately surrounds the `let/cc`: either by falling through (because the continuation of the body, `b`, is `k`) or—more interestingly—by invoking the continuation, which discards the dynamic continuation `dyn/k` by simply ignoring it and returning to `k` instead.

Here's the simplest test:

```
(test (run (cps (let/cc esc 3)))
      3)
```

This confirms that if we never use the continuation, evaluation of the body proceeds as if the `let/cc` weren't there at all (because of the `((cps b) k)`). If we use it, the value given to the continuation returns to the point of creation:

```
(test (run (cps (let/cc esc (esc 3))))
      3)
```

This example, of course, isn't revealing, but consider this one:

```
(test (run (cps (+ 1 (let/cc esc (esc 3)))))
      4)
```

This confirms that the addition actually happens. But what about the dynamic continuation?

```
(test (run (cps (let/cc esc (+ 2 (esc 3)))))
      3)
```

This shows that the addition by 2 never happens, i.e., the dynamic continuation is indeed ignored. And just to be sure that the continuation at the point of creation is respected, observe:

```
(test (run (cps (+ 1 (let/cc esc (+ 2 (esc 3))))))
      4)
```

From these examples, you have probably noticed a familiar pattern: `esc` here is behaving like an *exception*. That is, if you do not throw an exception (in this case, invoke a continuation) it's as if it's not there, but if you do throw it, all pending intermediate computation is ignored and computation returns to the point of exception creation.

Exercise

Using `let/cc` and macros, create a `throw/catch` mechanism.

However, these examples only scratch the surface of available power, because the continuation at the point of invocation is always an extension of one at the point of creation: i.e., the latter is just earlier in the stack than the former. However, nothing

actually demands that `k` and `dyn-k` be at all related. That means they are in fact free to be *un*related, which means each can be a distinct stack, so we can in fact easily implement stack-switching procedures with them.

Exercise

To be properly analogous to `lambda`, we should have introduced a construct called, say, `cont-lambda` with the following expansion:

```
[(_ (cont-lambda (a) b))
 (identifier? #'a)
 #'(lambda (k)
     (k (lambda (a dyn-k)
          ((cps b) k))))]
```

Why didn't we? Consider both the static typing implications, and also how we might construct the above exception-like behaviors using this construct instead.

14.6.1 Presentation in the Language

Writing programs in our little toy languages can soon become frustrating. Fortunately, Racket already provides a construct called `call/cc` that reifies continuations. `call/cc` is a procedure of one argument, which is itself a procedure of one argument, which Racket applies to the current continuation—which is a procedure of one argument. Got that?

Fortunately, we can easily write `let/cc` as a macro over `call/cc` and program with that instead. Here it is:

```
(define-syntax let/cc
  (syntax-rules ()
    [(let/cc k b)
     (call/cc (lambda (k) b))]))
```

To be sure, all our old tests still pass:

```
(test (let/cc esc 3) 3)
(test (let/cc esc (esc 3)) 3)
(test (+ 1 (let/cc esc (esc 3))) 4)
(test (let/cc esc (+ 2 (esc 3))) 3)
(test (+ 1 (let/cc esc (+ 2 (esc 3)))) 4)
```

14.6.2 Defining Generators

Now we can start to create interesting abstractions. For instance, let's build generators. Whereas previously we needed to both CPS expressions and pass around continuations, now this is done for us automatically by `call/cc`. Therefore, whenever we need the current continuation, we simply conjure it up without having to transform the program. Thus the extra `...-k` parameters can disappear with a `let/cc` in the same place to capture the same continuation:

```
(define-syntax (generator e)
  (syntax-case e ()
    [(generator (yield) (v) b)
     #'(let ([where-to-go (lambda (v) (error 'where-to-go "nothing"))])
         (letrec ([resumer (lambda (v)
                             (begin b
                                    (error 'generator "fell through")))]
                  [yield (lambda (v)
                           (let/cc gen-k
                             (begin
                               (set! resumer gen-k)
                               (where-to-go v))))])
           (lambda (v)
             (let/cc dyn-k
               (begin
                 (set! where-to-go dyn-k)
                 (resumer v)))))))]))
```

Observe the close similarity between this code and the implementation of generators by desugaring into CPS code. Specifically, we can drop the extra continuation arguments, and replace them with invocations of `let/cc` that will capture precisely the same continuations. The rest of the code is fundamentally unchanged.

Exercise

What happens if we move the `let/cc`s and mutation to be the first statement inside the `begin`s instead?

We can, for instance, write a generator that iterates from the initial value upward:

```
(define g1 (generator (yield) (v)
                      (letrec ([loop (lambda (n)
                                       (begin
                                         (yield n)
                                         (loop (+ n 1))))])
                        (loop v))))
```

whose behavior is:

```
> (g1 10)
10
> (g1 10)
11
> (g1 0)
12
>
```

Because the body refers only to the initial value, ignoring that returned by invoking `yield`, the values we pass on subsequent invocations are irrelevant. In contrast, consider this generator:

```
(define g2 (generator (yield) (v)
                      (letrec ([loop (lambda (n)
                                       (loop (+ (yield n) n)))])
                        (loop v))))
```

On its first invocation, it returns whatever value it was supplied. On subsequent invocations, this value is added to that provided on re-entry into the generator. In other words, this generator additively accumulates all values given to it:

```
> (g2 10)
10
> (g2 15)
25
> (g2 5)
30
```

Exercise

Now that you've seen how to implement generators using `call/cc` and `let/cc`, implement coroutines and threads as well.

14.6.3 Defining Threads

Having done generators, let's do another, similar primitive: threads. That is, let's assume we want to be able to write a program such as this:

```
(define d display) ;; a useful shorthand in what follows

(scheduler-loop-0
 (list
  (thread-0 (y) (d "t1-1  ") (y) (d "t1-2  ") (y) (d "t1-3 "))
  (thread-0 (y) (d "t2-1  ") (y) (d "t2-2  ") (y) (d "t2-3 "))
  (thread-0 (y) (d "t3-1  ") (y) (d "t3-2  ") (y) (d "t3-3 "))))
```

and expect the output

```
t1-1  t2-1  t3-1  t1-2  t2-2  t3-2  t1-3 t2-3 t3-3
```

We'll build all the pieces necessary to achieve this.

Let's start by defining the thread scheduler. It consumes a list of "threads", whose interface we assume will be a procedure that consumes a continuation to which it eventually yields control. Each time the scheduler reactivates the thread, it supplies it with a continuation. The scheduler might be choose between threads in a simple *round-robin* manner, or it might use some more complex algorithm; the details of how it chooses don't concern us here.

As with generators, we'll assume that yielding is done by invoking a procedure named by the user: `y`, above. We could use name capture [REF] to automatically bind a name like `yield`.

More importantly, notice that the user of the thread system manually yields control. This is called *cooperative multitasking*. Instead, we could have chosen to have a timer or other intrinsic mechanism automtically yield without the user's permission; this is

called *preemptive multitasking* (because the system "pre-empts"—i.e., wrests control from—the thread). While the distinction is important for buildling systems, it is not interesting from the perspective of setting up the continuations.

Exercise

> After we are done building cooperative multitasking, implement preemptive multitasking. What changes?

With our stated constraints, we can write a first scheduler. It consumes a lists of threads and continues executing so long as there are threads remaining. Each time, it applies the thread procedure to a continuation that represents returning to the scheduler and proceeding to the next thread:

```
(define (scheduler-loop-0 threads)
  (cond
    [(empty? threads) 'done]
    [(cons? threads)
     (begin
       (let/cc after-thread ((first threads) after-thread))
       (scheduler-loop-0 (append (rest threads)
                                 (list (first threads)))))]))
```

When the recipient thread invokes the continuation bound to `after-thread`, control returns to the end of the first statement in the `begin` sequence. As a result, the value supplied to the continuation is ignored (and can hence be any dummy value; we'll chose `'dummy`, so that we can easily spot it if it shows up in undesired places). Control then proceeds with the rest of the scheduler loop after appending the most recently invoked thread to the end of the list of threads (i.e., treating the list as a circular queue).

Now let's define a thread. As we've said, it will be a procedure of one argument, the scheduler's continuation. Because the thread needs to *resume*, i.e., continue where it left off, presumably it must store where it last was: we'll call this `thread-resumer`. Initially this is the entire thread body, but on subsequent instances it will be a continuation: specifically, the continuation of invoking `yield`. Thus, we obtain the following skeleton:

```
(define-syntax thread-0
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...))])
       (lambda (sched-k)
         (thread-resumer 'dummy)))]))
```

That still leaves the yielder. It needs to be a procedure of no arguments that stores the thread's continuation in `thread-resumer`, and then invoke the scheduler continuation with `'dummy`. However, *which* scheduler continuation does it need to invoke? Not the one provided on thread initiation, but rather the most recent one. Thus, we must somehow "thread" the value in `sched-k` to the yielder. There are many ways to accomplish

it, but the simplest, perhaps most brutal, way is to simply reconstruct the yielder on each thread resumption, always closed over the most recent value of `sched-k`:

```
(define-syntax thread-0
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...))]
              [yielder (lambda () (error 'yielder "nothing here"))])
       (lambda (sched-k)
         (begin
           (set! yielder
                 (lambda ()
                   (let/cc thread-k
                     (begin
                       (set! thread-resumer thread-k)
                       (sched-k 'dummy)))))
           (thread-resumer 'tres))))]))
```

When we run this ensemble, we get:
`t1-1 t2-1 t3-1 t1-2 t2-2 t3-2 t1-3 t2-3 t3-3`
Hey, that's what we wanted! But it continues:
`t1-3 t2-3 t3-3 t1-3 t2-3 t3-3 t1-3 t2-3 t3-3`
Hmmm.

What's happening? Well, we've been quiet all along about what needs to happen when a thread reaches its end. In fact, control just returns to the thread scheduler, which appends the thread to the end of the queue, and when the thread comes to the head of the queue again, control resumes from the same previously stored continuation: the one corresponding to printing the third value. This prints, control returns, the thread is appended...ad infinitum.

Clearly, we need the thread scheduler to be notified when a thread has terminated, so that the scheduler can remove it from the thread queue. We'll create a simple datatype to represent this signal:

```
(define-type ThreadStatus
  [Tsuspended]
  [Tdone])
```

(In a real system, of course, these status messages might also carry informative values from the computation.) We must now modify our scheduler to actually check for and use these values:

```
(define (scheduler-loop-1 threads)
  (cond
    [(empty? threads) 'done]
    [(cons? threads)
```

```
     (type-case ThreadStatus (let/cc after-thread ((first threads) after-
thread))
       [Tsuspended () (scheduler-loop-1 (append (rest threads)
                                                (list (first threads))))]
       [Tdone () (scheduler-loop-1 (rest threads))]))))
```

We have to now modify our thread representation in two ways: it must provide `Tsuspended` to the scheduler's continuation on intermediate returns, and provide `Tdone` when it terminates. Where does it terminate? After executing the code in the body, `b ...`. Observe, finally, that the termination process must be sure to use the latest scheduler continuation, just as yielding does. Thus:

```
(define-syntax thread-1
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...
                                       (finisher)))]
              [finisher (lambda () (error 'finisher "nothing here"))]
              [yielder (lambda () (error 'yielder "nothing here"))])
       (lambda (sched-k)
         (begin
           (set! finisher
                 (lambda ()
                   (let/cc thread-k
                     (sched-k (Tdone)))))
           (set! yielder
                 (lambda ()
                         (let/cc thread-k
                           (begin
                             (set! thread-resumer thread-k)
                             (sched-k (Tsuspended))))))
           (thread-resumer 'tres))))]))
```

If we now replace `scheduler-loop-0` with `scheduler-loop-1` and `thread-0` with `thread-1` and re-run our example program above, we get just the output we desire.

14.6.4 Better Primitives for Web Programming

Finally, to tie the knot back to where we began, let's return to `read-number`: observe that, if the language running the server program has `call/cc`, instead of having to CPS the entire program, we can simply capture the current continuation and save it in the hash table, leaving the program structure again intact.

15 Checking Program Invariants Statically: Types

Contents

As programs grow larger or more subtle, developers need tools to help them describe and validate statements about program *invariants*. Invariants, as the name suggests, are statements about program elements that are expected to always hold of those elements. For example, when we write `x : number` in our typed language, we mean that `x` will always hold a `number`, and that all parts of the program that depend on `x` can rely on this statement being enforced. As we will see, types are just one point in a spectrum of invariants we might wish to state, and static type checking—itself a diverse family of techniques—is also a point in a spectrum of methods we can use to enforce the invariants.

15.1 Types as Static Disciplines

In this chapter, we will focus especially on *static type checking*: that is, checking (declared) types before the program even executes. We have already experienced a form of this in our programs by virtue of using a typed programming language. We will explore some of the design space of types and their trade-offs. Finally, though static typing is an especially powerful and important form of invariant enforcement, we will also examine some other techniques that we have available.

Consider this program in our typed language:

```
(define (f [n : number]) : number
  (+ n 3))

(f "x")
```

We get a static type error before the program begins execution. The same program (without the type annotations) in ordinary Racket fails only at runtime:

```
(define (f n)
  (+ n 3))

(f "x")
```

Exercise

> How would you test the assertions that one fails before the program executes while the other fails during execution?

Now consider the following Racket program:

```
(define f n
  (+ n 3))
```

This too fails before program execution begins, with a parse error. Though we think of parsing as being somehow distinct from type-checking—usually because the type-checker assumes it has a parsed program to begin with—it can be useful to think of parsing as being simply the very simplest kind of type-checking: determining (typically) whether the program obeys a context-*free* syntax. Type-checking then asks

whether it obeys a context-*sensitive* (or richer) syntax. In short, type-checking is a generalization of parsing, in that both are concerned with *syntactic* methods for enforcing disciplines on programs.

15.2 A Classical View of Types

We will begin by introducing a traditional core language of types. Later, we will explore both extensions [REF] and variations [REF].

15.2.1 A Simple Type Checker

Before we can define a type checker, we have to fix two things: the syntax of our *typed* core language and, hand-in-hand with that, the syntax of types themselves.

To begin with, we'll return to our language with functions-as-values [REF] but before we added mutation and other complications (some of which we'll return to later). To this language we have to add type annotations. Conventionally, we don't impose type annotations on constants or on primitive operations such as addition; instead, we impose them on the boundaries of functions or methods. Over the course of this study we will explore why this is a good locus for annotations.

Given this decision, our typed core language becomes:

```
(define-type TyExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [appC (fun : TyExprC) (arg : TyExprC)]
  [plusC (l : TyExprC) (r : TyExprC)]
  [multC (l : TyExprC) (r : TyExprC)]
  [lamC (arg : symbol) (argT : Type) (retT : Type) (body : TyExprC)])
```

That is, every procedure is annotated with the type of argument it expects and type of argument it purports to produce.

Now we have to decide on a language of types. To do so, we follow the tradition that the types *abstract over the set of values*. In our language, we have two kinds of values:

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

It follows that we should have two kinds of types: one for numbers and the other for functions.

Even numeric types may not be straightforward: What information does a number type need to record? In most languages, there are actually *many* numeric types, and indeed there may not even be a single one that represents "numbers". However, we have ignored these gradations between numbers [REF], so it's sufficient for us to have just one. Having decided that, do we record additional information about *which* number? We could in principle, but we would soon run into decidability problems.

As for functions, we have more information: the type of expected argument, and the type of claimed result. We might as well record this information we have been given until and unless it has proven to not be useful. Combining these, we obtain the following abstract language of types:

```
(define-type Type
  [numT]
  [funT (arg : Type) (ret : Type)])
```

Now that we've fixed both the term and type structure of the language, let's make sure we agree on what constitute type errors in our language (and, by fiat, everything not a type error must pass the type checker). There are three obvious forms of type errors:

- One or both arguments of + is not a number, i.e., is not a `numT`.
- One or both arguments of * is not a number.
- The expression in the function position of an application is not a function, i.e., is not a `funT`.

Do Now!

Any more?

We're actually missing one:

- The expression in the function position of an application is a function but the type of the actual argument does not match the type of the formal argument expected by the function.

It seems clear all other programs in our language ought to type-check.

A natural starting signature for the type-checker would be that it is a procedure that consumes an expression and returns a boolean value indicating whether or not the expression type-checked. Because we know expressions contain identifiers, it soon becomes clear that we will want a *type environment*, which maps names to types, analogous to the value environment we have seen so far.

Exercise

Define the types and functions associated with type environments.

Thus, we might begin our program as follows:

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

```
(define (tc [expr : TyExprC] [tenv : TyEnv]) : boolean
  (type-case TyExprC expr
    <tc-take-1-numC-case>
    <tc-take-1-idC-case>
    <tc-take-1-appC-case>))
```

As the abbreviated set of cases above suggests, this approach will not work out. We'll soon see why.

Let's begin with the easy case: numbers. Does a number type-check? Well, on its own, of course it does; it may be that the surrounding context is not expecting a number, but that error would be signaled elsewhere. Thus:

<***tc-take-1-numC-case***> **::=**

```
[numC (n) true]
```

Now let's handle identifiers. Is an identifier well-typed? Again, on its own it would appear to be, provided it is actually a bound identifier; it may not be what the context desires, but hopefully that too would be handled elsewhere. Thus we might write

<***tc-take-1-idC-case***> **::=**

```
[idC (n) (if (lookup n tenv)
             true
             (error 'tc "not a bound identifier"))]
```

This should make you a little uncomfortable: `lookup` already signals an error if an identifier isn't bound, so there's no need to repeat it (indeed, we will never get to this `error` invocation). But let's push on.

Now we tackle applications. We should type-check both the function part, to make sure it's a function, then ensure that the actual argument's type is consistent with what the function declares to be the type of its formal argument. For instance, the function could be a number and the application could itself be a function, or vice versa, and in either case we want to prevent such mis-applications.

How does the code look?

<***tc-take-1-appC-case***> **::=**

```
[appC (f a) (let ([ft (tc f tenv)])
              ...)]
```

The recursive call to `tc` can only tell us whether the function expression type-checks or not. If it does, how do we know what type it has? If we have an immediate function, we could reach into its syntax and pull out the argument (and return) types. But if we have a complex expression, we need some procedure that will *calculate* the resulting type of that expression. Of course, such a procedure could only provide a type if the expression is well-typed; otherwise it would not be able to provide a coherent answer. In other words, *our type "calculator" has type "checking" as a special case*! We should therefore strengthen the inductive invariant on `tc`: that it not only tells us whether an expression is typed but also what its type is. Indeed, by giving any type at all it confirms that the expression types, and otherwise it signals an error.

Let's now define this richer notion of a type "checker".

<***tc***> **::=**

```
(define (tc [expr : TyExprC] [tenv : TyEnv]) : Type
  (type-case TyExprC expr
```

<tc-numC-case>
<tc-idC-case>
<tc-plusC-case>
<tc-multC-case>
<tc-appC-case>
<tc-lamC-case>))

Now let's fill in the pieces. Numbers are easy: they have the numeric type.

<tc-numC-case> **::=**

```
[numC (n) (numT)]
```

Similarly, identifiers have whatever type the environment says they do (and if they aren't bound, this signals an error).

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

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

Observe, so far, the similarity to and difference from interpreting: in the identifier case we did essentially the same thing (except we returned a type rather than an actual value), whereas in the numeric case we returned the abstract "number" rather than indicate which specific number it was.

Let's now examine addition. We must make sure both sub-expressions have numeric type; only if they do will the overall expression evaluate to a number itself.

<tc-plusC-case> **::=**

```
[plusC (l r) (let ([lt (tc l tenv)]
                   [rt (tc r tenv)])
               (if (and (equal? lt (numT))
                        (equal? rt (numT)))
                   (numT)
                   (error 'tc "+ not both numbers")))]
```

We've usually glossed over multiplication after considering addition, but now it will be instructive to handle it explicitly:

<tc-multC-case> **::=**

```
[multC (l r) (let ([lt (tc l tenv)]
                   [rt (tc r tenv)])
               (if (and (equal? lt (numT))
                        (equal? rt (numT)))
                   (numT)
                   (error 'tc "* not both numbers")))]
```

Do Now!

Did you see what's different?

That's right: practically *nothing*! (The `multC` instead of `plusC` in the `type-case`, and the slightly different error message, notwithstanding.) That's because, from the perspective of type-checking (in this type language), there is no difference between addition and multiplication, or indeed between *any* two functions that consume two numbers and return one.

Observe another difference between interpreting and type-checking. Both care that the arguments be numbers. The interpreter then returns a precise sum or product, but the type-checker is indifferent to the differences between them: therefore the expression that computes what it returns (`(numT)`) is a constant, and the same constant in both cases.

Finally, the two hard cases: application and funcions. We've already discussed what application must do: compute the value of the function and argument expressions; ensure the function expression has function type; and check that the argument expression is of compatible type. If all this holds up, then the type of the overall application is whatever type the function body would return (because the value that eventually returns at run-time is the result of evaluating the function's body).

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

```
[appC (f a) (let ([ft (tc f tenv)]
                  [at (tc a tenv)])
              (cond
                [(not (funT? ft))
                 (error 'tc "not a function")]
                [(not (equal? (funT-arg ft) at))
                 (error 'tc "app arg mismatch")]
                [else (funT-ret ft)]))]
```

That leaves function definitions. The function has a formal parameter, which is presumably used in the body; unless this is bound in the environment, the body most probably will not type-check properly. Thus we have to extend the type environment with the formal name bound to its type, and in that extended environment type-check the body. Whatever value this computes must be the same as the declared type of the body. If that is so, then the function itself has a function type from the type of the argument to the type of the body.

Exercise

Why do I say "most probably" above?

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

```
[lamC (a argT retT b)
      (if (equal? (tc b (extend-ty-env (bind a argT) tenv)) retT)
          (funT argT retT)
          (error 'tc "lam type mismatch"))]
```

Observe another curious difference between the interpreter and type-checker. In the interpreter, application was responsible for evaluating the argument expression, extending the environment, and evaluating the body. Here, the application case does check

the argument expression, but leaves the environment alone, and simply returns the type of the body *without traversing it*. Instead, the body is actually traversed by the checker when checking a function *definition*, so this is the point at which the environment actually extends.

15.2.2 Type-Checking Conditionals

Suppose we extend the above language with conditionals. Even the humble `if` introduces several design decisions. We'll discuss two here, and return to one of them later [REF].

1. What should be the type of the test expression? In some languages it must evaluate to a boolean value, in which case we have to enrich the type language to include booleans (which would probably be a good idea anyway). In other languages it can be any value, and some values are considered "truthy" while others "falsy".

2. What should be the relationship between the then- and else-branches? In some languages they must be of the same type, so that there is a single, unambiguous type for the overall expression (which is that one type). In other languages the two branches can have distinct types, which greatly changes the design of the type-language and -checker, but also of the nature of the programming language itself.

Exercise

Add booleans to the type language. What does this entail at a minimum, and what else might be expected in a typical language?

Exercise

Add a type rule for conditionals, where the test expression is expected to evaluate to a boolean and both then- and else-branches must have the same type, which is the type of the overall expression.

15.2.3 Recursion in Code

Now that we've obtained a basic programming language, let's add recursion to it. We saw earlier [REF] that this could be done easily through desugaring. It'll prove to be a more complex story here.

A First Attempt at Typing Recursion

Let's now try to express a simple recursive function. The simplest is, of course, one that loops forever. Can we write an infinite loop with just functions? We already could simply with this program—

```
((lambda (x) (x x))
 (lambda (x) (x x)))
```

—which we know we can represent in our language with functions as values.

Exercise

> Why does this construct an infinite loop? What subtle dependency is it making about the nature of function calls?

Now that we have a typed language, and one that forces us to annotate all functions, let's annotate it. For simplicity, from now on we'll assume we're writing programs in a typed surface syntax, and that desugaring takes care of constructing core language terms.

Observe, first, that we have two identical terms being applied to each other. Historically, the overall term is called Ω (capital omega in Greek) and each of the identical sub-terms is called ω (lower-case omega in Greek). It is not a given that identical terms must have precisely the same type, because it depends on what invariants we want to assert of the context of use. In this case, however, observe that `x` binds to ω, so the second ω goes into both the first and second positions. As a result, typing one effectively types both.

Therefore, let's try to type ω; let's call this type γ. It's clearly a function type, and the function takes one argument, so it must be of the form ϕ `->` ψ. Now what is that argument? It's ω itself. That is, the type of the value going into ϕ is itself γ. Thus, the type of ω is γ, which is ϕ `->` ψ, which expands into `(`ϕ `->` ψ`)` `->` ψ, which further expands to `((`ϕ `->` ψ`)` `->` ψ`)` `->` ψ, and so on. In other words, this type cannot be written as any finite string!

Do Now!

> Did you notice the subtle but important leap we just made?

Program Termination

We observed that the obvious typing of Ω, which entails typing γ, seems to run into serious problems. From that, however, we jumped to the conclusion that this type cannot be written as any finite string, for which we've given only an intuition, not a proof. In fact, something even stranger is true: in the type system we've defined so far, *we cannot type* Ω *at all*!

This is a strong statement, but we can actually say something much stronger. The *typed* language we have so far has a property called *strong normalization*: every expression that has a type will terminate computation after a finite number of steps. In other words, this special (and peculiar) infinite loop program isn't the only one we can't type; we can't type *any* infinite loop (or even potential infinite loop). A rough intuition that might help is that any type—which must be a finite string—can have only a finite number of `->`'s in it, and each application discharges one, so we can perform only a finite number of applications.

If our language permitted only straight-line programs, this would be unsurprising. However, we have conditionals and even functions being passed around as values, and with those we can encode any datatype we want. Yet, we still get this guarantee! That makes this a somewhat astonishing result.

Exercise

Try to encode lists using functions in the untyped and then in the typed language. What do you see? And what does that tell you about the impact of this type system on the encoding?

This result also says something deeper. It shows that, contrary to what you may believe—that a type system only prevents a few buggy programs from running—a type system can *change the semantics* of a language. Whereas previously we could write an infinite loop in just one to two lines, now we cannot write one at all. It also shows that the type system can establish invariants not just about a particular program, but *about the language itself*. If we want to absolutely ensure that a program will terminate, we simply need to write it in this language and pass the type checker, and the guarantee is ours!

What possible use is a language in which all programs terminate? For general-purpose programming, none, of course. But in many specialized domains, it's a tremendously useful guarantee to have. For instance, suppose you are implementing a complex scheduling algorithm; you would like to know that your scheduler is guaranteed to terminate so that the tasks being scheduled will actually run. There are many other domains, too, where we would benefit from such a guarantee: a packet-filter in a router; a real-time event processor; a device initializer; a configuration file; the callbacks in single-threaded JavaScript; and even a compiler or linker. In each case, we have an almost unstated expectation that these programs will always terminate. And now we have a language that can offer this guarantee—something it is impossible to test for, no less!

These are not hypothetical examples. In the Standard ML language, the language for linking modules uses essentially this typed language for writing module linking specifications. This means developers can write quite sophisticated abstractions—they have functions-as-values, after all!—while still being guaranteed that linking will always terminate, producing a program.

Typing Recursion

What this says is, whereas before we were able to handle `rec` entirely through desugaring, now we must make it an explicit part of the typed language. For simplicity, we will consider a special case of `rec`—which nevertheless covers the common uses—whereby the recursive identifier is bound to a function. Thus, in the surface syntax, one might write

```
(rec (Σ num (n num)
        (if0 n
             0
             (n + (Σ (n + -1)))))
  (Σ 10))
```

for a summation function, where `Σ` is the name of the function, `n` its argument, and `num` the type consumed by and returned from the function. The expression `(Σ 10)` represents the use of this function to sum the number from `10` until `0`.

How do we type such an expression? Clearly, we must have `n` bound in the body of the function as we type it (but not of course, in the use of the function); this much we know from typing functions. But what about `Σ`? Obviously it must be bound in the type environment when checking the use (`(Σ 10)`), and its type must be `num -> num`. But it must *also* be bound, to the same type, when checking the body of the function. (Observe, too, that the type returned by the body must match its declared return type.)

Now we can see how to break the shackles of the finiteness of the type. It is certainly true that we can write only a finite number of `->`'s in types in the program source.

However, this rule for typing recursion duplicates the `->` in the body that refers to itself, thereby ensuring that there is an inexhaustible supply of applications. It's our infinite quiver of arrows.

The code to implement this rule would be as follows. Assuming `f` is bound to the function's name, `aT` is the function's argument type and `rT` is its return type, `b` is the function's body, and `u` is the function's use:

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

```
[recC (f a aT rT b u)
      (let ([extended-env
             (extend-ty-env (bind f (funT aT rT)) tenv)])
        (cond
          [(not (equal? rT (tc b
                               (extend-ty-env
                                (bind a aT)
                                extended-env))))
           (error 'tc "body return type not correct")]
          [else (tc u extended-env)]))]
```

15.2.4 Recursion in Data

We have seen how to type recursive programs, but this doesn't yet enable us to create recursive data. We already have one kind of recursive datum—the function type—but this is built-in. We haven't yet seen how developers can create their own recursive datatypes.

Recursive Datatype Definitions

When we speak of allowing programmers to create recursive data, we are actually talking about three different facilities at once:

- Creating a new type.
- Letting instances of the new type have one or more fields.
- Letting some of these fields refer to instances of the same type.

In fact, once we allow the third, we must allow one more:

- Allowing non-recursive base-cases for the type.

This confluence of design criteria leads to what is commonly called an *algebraic datatype*, such as the types supported by our typed language. For instance, consider the following definition of a binary tree of numbers:

Later [REF], we will discuss how types can be parameterized.

```
(define-type BTnum
  [BTmt]
  [BTnd (n : number) (l : BTnum) (r : BTnum)])
```

Observe that without a name for the new datatype, `BTnum`, we would not have been able to refer back ot it in `BTnd`. Similarly, without the ability to have more than one

kind of BTnum, we would not have been able to define BTmt, and thus wouldn't have been able to terminate the recursion. Finally, of course, we need multiple fields (as in BTnd) to construct useful and interesting data. In other words, all three mechanisms are packaged together because they are most useful in conjunction. (However, some langauges do permit the definition of stand-alone structures. We will return to the impact of this design decision on the type system later [REF].)

This concludes our initial presentation of recursive types, but it has a fatal problem. We have not actually explained where this new type, BTnum, comes from. That is because we have had to pretend it is baked into our type-checker. However, it is simply impractical to keep changing our type-checker for each new recursive type definition—it would be like modifying our interpreter each time the program contains a recursive function! Instead, we need to find a way to make such definitions intrinsic to the type language. We will return to this problem later [REF].

This style of data definition is sometimes also known as a *sum of products*. "Product" refers to the way fields combine in one variant: for instance, the legal values of a BTnd are the cross-product of legal values in each of the fields, supplied to the BTnd constructor. The "sum" is the aggregate of all these variants: any given BTnum value is just one of these. (Think of "product" as being "and", and "sum" as being "or".)

Introduced Types

Now, what impact does a datatype definition have? First, it introduces a new type; then it uses this to define several constructors, predicates, and selectors. For instance, in the above example, it first introduces BTnum, then uses it to ascribe the following types:

```
BTmt : -> BTnum
BTnd : number * BTnum * BTnum -> BTnum
BTmt? : BTnum -> boolean
BTnd? : BTnum -> boolean
BTnd-n : BTnum -> number
BTnd-l : BTnum -> BTnum
BTnd-r : BTnum -> BTnum
```

Observe a few salient facts:

- Both the constructors create instances of BTnum, not something more refined. We will discuss this design tradeoff later [REF].
- Both predicates consume values of type BTnum, not "any". This is because the type system can already tell us what type a value is. Thus, we only need to distinguish between the variants of that one type.
- The selectors really only work on instances of the relevant variant—e.g., BTnd-n can work only on instances of BTnd, not on instances of BTmt—but we don't have a way to express this in the static type system for lack of a suitable static type. Thus, applying these can only result in a dynamic error, not a static one caught by the type system.

There is more to say about recursive types, which we will return to shortly [REF].

Pattern-Matching and Desugaring

Once we observe that these are the types, the only thing left is to provide an account of pattern-matching. For instance, we can write the expression

```
(type-case BTnum t
  [BTmt () e1]
  [BTnd (nv lt rt) e2])
```

We have already seen [REF] that this can be written in terms of the functions defined above. We can simulate the binding done by this pattern-matcher using `let`:

```
(cond
  [(BTmt? t) e1]
  [(BTnd? t) (let ([nv (BTnd-n t)]
                   [lt (BTnd-l t)]
                   [rt (BTnd-r t)])
               e2)])
```

In short, this can be done by a macro, so pattern-matching does not need to be in the core language and can instead be delegated to desugaring. This, in turn, means that one language can have many different pattern-matching mechanisms.

Except, that's not quite true. Somehow, the macro that generates the code above in terms of `cond` needs to know that the three positional selectors for a `BTnd` are `BTnd-n`, `BTnd-l`, and `BTnd-r`, respectively. This information is explicit in the type definition but only implicitly present in the use of the pattern-matcher (that, indeed, being the point). Thus, somehow this information must be communicated from definition to use. Thus the macro expander needs something akin to the type environment to accomplish its task.

Observe, furthermore, that expressions such as `e1` and `e2` cannot be type-checked—indeed, cannot even be reliable identified as *expressions*—until macro expansion expands the use of `type-case`. Thus, expansion depends on the type environment, while type-checking depends on the result of expansion. In other words, the two are symbiotic and need to happen, not quite in "parallel", but rather in lock-step. Thus, building desugaring for a typed language, where the syntactic sugar makes assumptions about types, is a little more intricate than doing so for an untyped language.

15.2.5 Types, Time, and Space

It is evident that types already bestow a performance benefit in safe languages. That is because the checks that would have been performed at run-time—e.g., `+` checking that both its arguments are indeed numbers—are now performed statically. In a typed language, an annotation like `: number` already answers the question of whether or not something is of a particular a type; there is nothing to ask at run-time. As a result, these type-level predicates can (and need to) disappear entirely, and with them any need to use them in programs.

This is at some cost to the developer, who must convince the static type system that their program does not induce type errors; due to the limitations of decidability, even programs that might have run without error might run afoul of the type system. Nevertheless, for programs that meet this requirement, types provide a notable execution time saving.

Now let's discuss space. Until now, the language run-time system has needed to store information attached to every value indicating what its type is. This is how it can implement type-level predicates such as `number?`, which may be used both by developers and by primitives. If those predicates disappear, so does the space needed to hold information to implement them. Thus, type-tags are no longer necessary.

They would, however, still be needed by the garbage collector, though other representations such as BIBOP can greatly reduce their space impact.

The type-like predicates still left are those for variants: `BTmt?` and `BTnd?`, in the example above. These must indeed be applied at run-time. For instance, as we have noted, selectors like `BTnd-n` must perform this check. Of course, some more optimizations are possible. Consider the code generated by desugaring the pattern-matcher: there is no need for the three selectors to implement this check, because control could only have gotten to them after `BTnd?` returned a true vlaue. Thus, the run-time system could provide just the desugaring level access to special *unsafe* primitives that do not perform the check, resulting in generated code such as this:

```
(cond
  [(BTmt? t) e1]
  [(BTnd? t) (let ([nv (BTnd-n/no-check t)]
                   [lt (BTnd-l/no-check t)]
                   [rt (BTnd-r/no-check t)])
               e2)])
```

The net result, however, is that the run-time representation must still store enough information to accurately answer these questions. However, previously it needed to use enough bits to record every possible type (and variant). Now, because the types have been statically segregated, for a type with no variants (e.g., there is only one kind of string), there is no need to store any variant information at all; that means the run-time system can use all available bits to store actual dynamic values.

In contrast, when variants are present, the run-time system must sacrifice bits to distinguish between the variants, but the number of *variants within a type* is obviously far smaller than the number of variants and types *across all types*. In the `BTnum` example above, there are only two variants, so the run-time system needs to use only one bit to record which variant of `BTnum` a value represents.

Observe, in particular, that the type system's segregation prevents confusion. If there are two different datatypes that each have two variants, in the untyped world all these four variants require distinct representations. In contrast, in the typed world these representations can overlap across types, because the static type system will ensure one type's variants are never confused for that the another. Thus, types have a genuine space (saving representation) and time (eliminating run-time checks) performance benefit for programs.

15.2.6 Types and Mutation

We have now covered most of the basic features of our core language other than mutation. In some ways, types have a simple interaction with mutation, and this is because in a classical setting, they don't interact at all. Consider, for instance, the following untyped program:

```
(let ([x 10])
  (begin
    (set! x 5)
    (set! x "something")))
```

What is "the type" of `x`? It doesn't really have one: for some time it's a number, and *later* (note the temporal word) it's a string. We simply can't give it a type. In general, type checking is an *atemporal* activity: it is done once, before the program runs, and must hence be independent of the specific order in which programs execute. Keeping track of the precise values in the store is hence beyond the reach of a type-checker.

The example above is, of course, easy to statically understand, but we should never be mislead by simple examples. Suppose instead we had a program like

```
(let ([x 10])
  (if (even? (read-number "Enter a number"))
    (set! x 5)
    (set! x "something")))
```

Now it is literally impossible to reach any static conclusion about the type of `x` after the conditional finishes, because only at run-time can we know what the user might have entered.

To avoid this morass, traditional type checkers adopt a simple policy: types must be *invariant* across mutation. That is, a mutation operation—whether variable mutation or structure mutation—cannot change the type of the mutant. Thus, the above examples would not type in our type language so far. How much flexibility this gives the programmer is, however, a function of the type language. For instance, if we were to admit a more flexible type that stands for "number or string", then the examples above would type, but `x` would always have this, less precise, type, and all uses of `x` would have to contend with its reduced specificity, an issue we will return to later [REF].

In short, mutation is easy to account for in a traditional type system because its rule is simply that, while the value can change in ways below the level of specificity of the type system, the type cannot change. In the case of an operation like `set!` (or our core language's `setC`), this means the type of the assigned value must match that of the variable. In the case of structure mutation, such as boxes, it means the assigned value must match that the box's contained type.

15.2.7 The Central Theorem: Type Soundness

We have seen earlier [REF] that certain type languages can offer very strong theorems about their programs: for instance, that all programs in the language terminate.

In general, of course, we cannot obtain such a guarantee (indeed, we added general recursion precisely to let ourselves write unbounded loops). However, a meaningful type system—indeed, anything to which we wish to bestow the noble title of a *type system*—ought to provide some kind of meaningful guarantee that all typed programs enjoy. This is the payoff for the programmer: by typing this program, she can be certain that certain bad things will certainly not happen. Short of this, we have just a bug-finder; while it may be useful, it is not a sufficient basis for building any higher-level tools (e.g., for obtaining security or privacy or robustness guarantees).

We have repeatedly used the term "type *system*". A type system is usually a combination of three components: a language of types, a set of type rules, and an algorithm that applies these rules to programs. By largely presenting our type rules embedded in a function, we have blurred the distinction between the second and third of these, but can still be thought of as intellectually distinct.

What theorem might we want of a type system? Remember that the type checker runs over the static program, before execution. In doing so, it is essentially making a *prediction* about the program's behavior: for instance, when it states that a particular complex term has type `num`, it is effectively predicting that when run, that term will produce a numeric value. How do we know this prediction is sound, i.e., that the type checker never lies? Every type system should be accompanied by a theorem that proves this.

There is a good reason to be suspicious of a type system, beyond general skepticism. There are many differences between the way a type checker and a program evaluator work:

- The type checker only sees program text, whereas the evaluator runs over actual stores.
- The type environment binds identifiers to types, whereas the evaluator's environment binds identifiers to values or locations.
- The type checker compresses (even infinite) sets of values into types, whereas the evaluator treats these distinctly.
- The type checker always terminates, whereas the evaluator might not.
- The type checker passes over the body of each expression only once, whereas the evaluator might pass over each body anywhere from zero to infinite times.

Thus, we should not assume that these will always correspond!

The central result we wish to have for a given type-system is called *soundness*. It says this. Suppose we are given an expression (or program) `e`. We type-check it and conclude that its type is `t`. When we run `e`, let us say we obtain the value `v`. Then `v` will also have type `t`.

The standard way of proving this theorem is to prove it in two parts, known as *progress* and *preservation*. Progress says that if a term passes the type-checker, it will be able to make a step of evaluation (unless it is already a value); preservation says that the result of this step will have the same type as the original. If we interleave these steps (first progress, then preservation; repeat), we can conclude that the final answer will indeed have the same type as the original, so the type system is indeed sound.

For instance, consider this expression: `(+ 5 (* 2 3))`. It has the type `num`. In a sound type system, progress offers a proof that, because this term types, and is not already a value, it can take a step of execution—which it clearly can. After one step the program reduces to `(+ 5 6)`. Sure enough, as preservation proves, this has the

same type as the original: `num`. Progress again says this can take a step, producing `11`. Preservation again shows that this has the same type as the previous (intermediate) expressions: `num`. Now progress finds that we are at an answer, so there are no steps left to be taken, and our answer is of the same type as that given for the original expression.

However, this isn't the entire story. There are two caveats:

1. The program may not produce an answer at all; it might loop forever. In this case, the theorem strictly speaking does not apply. However, we can still observe that every intermediate term still has the same type, so the program is computing meaningfully even if it isn't producing a value.

2. Any rich enough language has properties that cannot be decided statically (and others that perhaps could be, but the language designer chose to put off until run-time). When one of these properties fails—e.g., the array index being within bounds—there is no meaningful type for the program. Thus, implicit in every type soundness theorem is some set of published, permitted exceptions or error conditions that may occur. The developer who uses a type system implicitly signs on to accepting this set.

As an example of the latter set, the user of a typical typed language acknowledges that vector dereference, list indexing, and so on may all yield exceptions.

The latter caveat looks like a cop-out. In fact, it is easy to forget that it is really a statement about what *cannot* happen at run-time: any exception not in this set will provably not be raised. Of course, in languages designed with static types in the first place, it is not clear (except by loose analogy) what these exceptions might be, because there would be no need to define them. But when we retrofit a type system onto an existing programming language—especially languages with only dynamic enforcement, such as Racket or Python—then there is already a well-defined set of exceptions, and the type-checker is explicitly stating that some set of those exceptions (such as "non-function found in application position" or "method not found") will simply never occur. This is therefore the payoff that the programmer receives in return for accepting the type system's syntactic restrictions.

15.3 Extensions to the Core

Now that we have a basic typed language, let's explore how we can extend it to obtain a more useful programming language.

15.3.1 Explicit Parametric Polymorphism

Which of these is the same?

- `List<String>`
- `List<String>`
- `(listof string)`

Actually, none of these is quite the same. But the first and third are very alike, because the first is in Java and the third in our typed language, whereas the second, in C++, is different. All clear? No? Good, read on!

Parameterized Types

The language we have been programming in already demonstrates the value of parametric polymorphism. For instance, the type of `map` is given as

```
(('a -> 'b) (listof 'a) -> (listof 'b))
```

which says that for all types `'a` and `'b`, `map` consumes a function that generates `'b` values from `'a` values, and a list of `'a` values, and generates the corresponding list of `'b` values. Here, `'a` and `'b` are not concrete types; rather, they are *type variables* (in our terminology, these should properly be called "type identifiers" because they don't change within the course of an instantiation; however, we will stick to the traditional terminology).

A different way to understand this is that there is actually an infinite family of `map` functions. For instance, there is a `map` that has this type:

```
((number -> string) (listof number) -> (listof string))
```

and another one of this type (nothing says the types have to be base types):

```
((number -> (number -> number)) (listof number) -> (listof (number -
> number)))
```

and yet another one of this type (nothing says `'a` and `'b` can't be the same):

```
((string -> string) (listof string) -> (listof string))
```

and so on. Because they have different types, they would need different names: `map_num_str`, `map_num_num->num`, `map_str_str`, and so on. But that would make them different functions, so we'd have to always refer to a specific `map` rather than each of the generic ones.

Obviously, it is impossible to load all these functions into our standard library: there's an infinite number of these! We'd rather have a way to obtain each of these functions on demand. Our naming convention offers a hint: it is as if `map` takes two *parameters*, which are *types*. Given the pair of types as arguments, we can then obtain a `map` that is customized to that particular type. This kind of *parameterization over types* is called *parametric polymorphism*.

Not to be confused with the "polymorphism" of objects, which we will discuss below [REF].

Making Parameters Explicit

In other words, we're effectively saying that `map` is actually a function of perhaps four arguments, two of them types and two of them actual values (a function and a list). In a language with explicit types, we might try to write

```
(define (map [a : ???] [b : ???] [f : (a -> b)] [l : (listof a)]) : (listof b)
  ...)
```

but this raises some questions. First, what goes in place of the ???? These are the types of `a` and `b`. But if `a` and `b` are themselves going to be replaced with *types*, then what is the type of a type? Second, do we really want to be calling `map` with four arguments on every instantiation? Third, do we really mean to take the type parameters first before any actual values? The answers to these questions actually lead to a very rich space of polymorphic type systems, most of which we will *not* explore here.

I recommend reading Pierce's *Types and Programming Languages* for a modern, accessible introduction.

Observe that once we start parameterizing, more code than we expect ends up being parameterized. For instance, consider the type of the humble `cons`. Its type really is parametric over the type of values in the list (even though it doesn't actually depend on those values!—more on that in a bit [REF]) so every use of `cons` must be instantiated at the appropriate type. For that matter, even `empty` must be instantiated to create an empty list of the correct type! Of course, Java and C++ programmers are familiar with this pain.

Rank-1 Polymorphism

Instead, we will limit ourselves to one particularly useful and tractable point in this space, which is the type system of Standard ML, of the typed language of this book, of earlier versions of Haskell, roughly that of Java and C# with generics, and roughly that obtained using templates in C++. This language defines what is called *predicative*, *rank-1*, or *prenex* polymorphism. It answers the above questions thus: nothing, no, and yes. Let's explore this below.

We first divide the world of types into two groups. The first group consists of the type language we've used until, but extended to include type variables; these are called *monotypes*. The second group, known as *polytypes*, consists of parameterized types; these are conventionally written with a ∀ prefix, a list of type variables, and then a type expression that might use these variables. Thus, the type of `map` would be:

```
∀ a, b : (('a -> 'b) (listof 'a) -> (listof 'b))
```

Since "∀" is the logic symbol for "for all", you would read this as: "for all types `'a` and `'b`, the type of `map` is...".

In rank-1 polymorphism, the type variables can only be substituted with monotypes. (Furthermore, these can only be concrete types, because there would be nothing left to substitute any remaining type variables.) As a result, we obtain a clear separation between the type variable-parameters and regular parameters. We don't need to provide a "type annotation" for the type variables because we know precisely what kind of thing they can be. This produces a relatively clean language that still offers considerable expressive power.

Impredicative languages erase the distinction between monotypes and polytypes, so a type variable can be instantiated with another polymorphic type.

Observe that because type variables can only be replaced with monotypes, they are all independent of each other. As a result, all type parameters can be brought to the front of the parameter list. This is what enables us to write types in the form `∀ tv, ... : t` where the `tv` are type variables and `t` is a monotype (that might refer to those variables). This justifies not only the syntax but also the name "prenex". It will prove to also be useful in the implementation.

Interpreting Rank-1 Polymorphism as Desugaring

The simplest implementation of this feature is to view it as a form of desugaring: this is essentially the interpretation taken by C++. (Put differently, because C++ has a

macro system in the form of templates, by a happy accident it obtains a form of rank-1 polymorphism through the use of templates.) For instance, imagine we had a new syntactic form, `define-poly`, which takes a name, a type variable, and a body. On every provision of a type to the name, it replaces the type variable with the given type in the body. Thus:

```
(define-poly (id t) (lambda ([x : t]) : t x))
```

defines an identity function by first defining `id` to be polymorphic: given a concrete type for `t`, it yields a procedure of one argument of type `(t -> t)` (where `t` is appropriately substituted). Thus we can instantiate `id` at many different types—

```
(define id_num (id number))
(define id_str (id string))
```

—thereby obtaining identity functions at each of those types: `(test (id_num 5) 5) (test (id_str "x") "x")` In contrast, expressions like `(id_num "x") (id_str 5)` will, as we would expect, *fail to type-check* (rather than fail at run-time).

In case you're curious, here's the implementation. For simplicity, we assume there is only one type parameter; this is easy to generalize using `...`. We will not only define a macro for `define-poly`, *it* will in turn define a macro:

```
(define-syntax define-poly
  (syntax-rules ()
    [(_ (name tyvar) body)
     (define-syntax (name stx)
       (syntax-case stx ()
         [(_ type)
          (with-syntax ([tyvar #'type])
            #'body)]))]))
```

Thus, given a definition such as

```
(define-poly (id t) (lambda ([x : t]) : t x))
```

the language creates a *macro* named `id`: the part that begins with `(define-syntax (name ...) ...)` (where, in this example, `name` is `id`). An instantiation of `id`, such as `(id number)`, replaces `t` the type variable, `tyvar`, with the given type. To circumvent hygiene, we use `with-syntax` to force all uses of the type variable (`tyvar`) to actually be replaced with the given type. Thus, in effect,

```
(define id_num (id number))
```

turns into

```
(define id_num (lambda ([x : number]) : number x))
```

However, this approach has two important limitations.

1. Let's try to define a recursive polymorphic function, such as `filter`. Earlier we have said that we ought to instantiate every single polymorphic value (such as even `cons` and `empty`) with types, but to keep our code concise we'll rely on the fact that the underlying typed language already does this, and focus just on type parameters for `filter`. Here's the code:

```
(define-poly (filter t)
  (lambda ([f : (t -> boolean)] [l : (listof t)]) : (listof t)
    (cond
      [(empty? l) empty]
      [(cons? l) (if (f (first l))
                     (cons (first l)
                           ((filter t) f (rest l)))
                     ((filter t) f (rest l)))])))
```

   Observe that at the recursive uses of `filter`, we must instantiate it with the appropriate type.

   This is a perfectly good definition. There's just one problem. When we try to use it—e.g.,

```
(define filter_num (filter number))
```

   DrRacket does not terminate. Specifically, macro expansion does not terminate, because it is repeatedly trying to make new *copies of the code of* `filter`. If, in contrast, we write the function as follows, expansion terminates—

```
(define-poly (filter2 t)
  (letrec ([fltr
            (lambda ([f : (t -> boolean)] [l : (listof t)]) : (listof t)
              (cond
                [(empty? l) empty]
                [(cons? l) (if (f (first l))
                               (cons (first l) (fltr f (rest l)))
                               (fltr f (rest l)))])])
    fltr))
```

   but this needlessly pushes pain onto the user. Indeed, some template expanders will cache previous values of expansion and avoid re-generating code when given the same parameters. (Racket cannot do this because, in general, the body of a macro can depend on mutable variables and values and even perform input-output, so Racket cannot guarantee that a given input will always generate the same output.)

2. Consider two instantiations of the identity function. We cannot compare `id_num` and `id_str` because they are of different types, but even if they are of the same type, they are not `eq?`:

```
(test (eq? (id number) (id number)) #f)
```

> This is because each use of `id` creates a new copy of the body. Now even if the optimization we mentioned above were applied, so for the *same* type there is only one code body, there would still be different code bodies for different types—but even this is unnecessary! There's absolutely nothing in the body of `id`, for instance, that actually depends on the type of the argument. Indeed, the entire infinite family of `id` functions can share just one implementation. The simple desugaring strategy fails to provide this.

Indeed, C++ templates are notorious for creating code bloat; this is one of the reasons.

In other words, the desugaring based strategy, which is essentially an implementation by substitution, has largely the same problems we saw earlier with regards to substitution as an implementation of parameter instantiation. However, in other cases substitution also gives us a ground truth for what we expect as the program's behavior. The same will be true with polymorphism, as we will soon see [REF].

Observe that one virtue to the desugaring strategy is that it does not require our type checker to "know" about polymorphism. Rather, the core type language can continue to be monomorphic, and all the (rank-1) polymorphism is handled entirely through expansion. This offers a cheap strategy for adding polymorphism to a language, though—as C++ shows—it also introduces significant overheads.

Finally, though we have only focused on functions, the preceding discussions applies equally well to data structures.

Alternate Implementations

There are other implementation strategies that don't suffer from these problems. We won't go into them here, but the essence of at least some of them is the "caching" approach we sketched above. Because we can be certain that, for a given set of type parameters, we will always get the same typed body, we never need to instantiate a polymorphic function at the same type twice. This avoids the infinite loop. If we type-check the instantiated body once, we can avoid checking at other instantiations of the same type (because the body will not have changed). Furthermore, we do not need to retain the instantiated sources: once we have checked the expanded program, we can dispose of the expanded terms and retain just one copy at run-time. This avoids all the problems discussed in the pure desugaring strategy shown above, while retaining the benefits.

Actually, we are being a little too glib. One of the benefits of static types is that they enable us to pick more precise run-time representations. For instance, a static type can tell us whether we have a 32-bit or 64-bit number, or for that matter a 32-bit value or a 1-bit value (effectively, a boolean). A compiler can then generate specialized code for each representation, taking advantage of how the bits are laid out (for example, 32 booleans can be *packed* into a single 32-bit word). Thus, after type-checking at each used type, the polymorphic instantiator may keep track of all the special types at which a function or data structure was used, and provide this information to the compiler for code-generation. This will then result in several copies of the function, none of which are `eq?` with each other—but for good reason and, because their operations are truly different, rightly so.

Relational Parametricity

There's one last detail we must address regarding polymorphism.

We earlier said that a function like `cons` doesn't depend on the specific values of its arguments. This is also true of `map`, `filter`, and so on. When `map` and `filter` want to operate on individual elements, they take as a parameter another function which in turn is responsible for making decisions about how to treat the elements; `map` and `filter` themselves simply obey their parameter functions.

One way to "test" whether this is true is to substitute some different values in the argument list, and a correspondingly different parameter function. That is, imagine we have a relation between two sets of values; we convert the list elements according to the relation, and the parameter function as well. The question is, will the output from `map` and `filter` also be predictable by the relation? If, for some input, this was not true of the output of `map`, then it must be that `map` inspected the actual values and did something with that information. But in fact this won't happen for `map`, or indeed most of the standard polymorphic functions.

Functions that obey this relational rule are called *relationally parametric*. This is another very powerful property that types give us, because they tell us there is a strong limit on the kinds of operations such polymorphic functions can perform: essentially, that they can drop, duplicate, and rearrange elements, but not directly inspect and make decisions on them.

Read Wadler's *Theorems for Free!* and Reynolds's *Types, Abstraction and Parametric Polymorphism*.

At first this sounds very impressive (and it is!), but on inspection you might realize this doesn't square with your experience. In Java, for instance, a polymorphic method can still use `instanceof` to check which particular kind of value it obtained at run-time, and change its behavior accordingly. Such a method would indeed not be relationally parametric! Indeed, relational parametricity can equally be viewed as a statement of the weakness of the language: that it permits only a very limited set of operations. (You could still inspect the type—but not act upon what you learned, which makes the inspection pointless. Therefore, a run-time system that wants to simulate relational parametricity would have to remove operations like `instanceof` as well as various proxies to it: for instance, adding `1` to a value and catching exceptions would reveal whether the value is a number.) Nevertheless, it is a very elegant and surprising result, and shows the power of program reasoning possible with rich type systems.

On the Web, you will often find this property described as the inability of a function to inspect the argument—which is not quite right.

15.3.2 Type Inference

Writing polymorphic type instantiations everywhere can be an awfully frustrating process, as users of many versions of Java and C++ can attest. Imagine if in our programs, every single time we wrote `first` or `rest`, we had to also instantiate it at a type! The reason we have been able to avoid this fate is because our language implements *type inference*. This is what enables us to write the definition

```
(define (mapper f l)
  (cond
    [(empty? l) empty]
    [(cons? l) (cons (f (first l)) (mapper f (rest l)))]))
```

and have the programming environment *automatically* declare that

```
> mapper
- (('a -> 'b) (listof 'a) -> (listof 'b))
```
Not only is this the correct type, this is a very general type! The process of being able to derive such general types just from the program structure feels almost magical. Now let's look behind the curtain.

First, let's understand what type inference is doing. Some people mistakenly think of languages with inference as having no type declarations, with inference taking their place. This is confused at multiple levels. For one thing, even in languages with inference, programmers are free (and for documentation purposes, often encouraged—as you have been) to declare types. Furthemore, in the absence of such declarations, it is not quite clear what inference actually *means*.

Sometimes, inference is also undecidable and programmers have no choice but to declare some of the types. Finally, writing explicit annotations can greatly reduce indecipherable error messages.

Instead, it is better to think of the underlying language as being fully, explicitly typed—just like the polymorphic language we have just studied [REF]. We will simply say that we are free to leave the type annotations after the `:`'s blank, and assume some programming environment feature will fill them in for us. (And if we can go that far, we can drop the `:`'s and extra embellishments as well, and let them all be inserted automatically. Thus, inference becomes simply a user convenience for alleviating the burden of writing type annotations, but the language underneath is explicitly typed.

How do we even think about what inference does? Suppose we have an expression (or program) `e`, written in an explicitly typed language: i.e., `e` has type annotations everywhere they are required. Now suppose we erase all annotations in `e`, and use a procedure `infer` to deduce them back.

Do Now!

What property do we expect of `infer`?

We could demand many things of it. One might be that it produces precisely those annotations that `e` originally had. This is problematic for many reasons, not least that `e` might not even type-check, in which case how could `infer` possibly know what they were (and hence should be)? This might strike you as a pedantic trifle: after all, if `e` didn't type-check, how can erasing its annotations and filling them back in make it do so? Since neither program type-checks, who cares?

Do Now!

Is this reasoning correct?

Suppose `e` is

```
(lambda ([x : number]) : string x)
```

This procedure obviously fails to type-check. But if we erase the type annotations—obtaining

```
(lambda (x) x)
```

—we equally obviously obtain a typeable function! Therefore, a more reasonable demand might be that if the original `e` type-checks, then so must the version with annotations replaced by inferred types. This one-directional implication is useful in two ways:

1. It does not say what must happen if `e` fails to type-check, i.e., it does not preclude the type inference algorithm we have, which makes the faultily-typed identity function above typeable.

2. More importantly, it assures us that we lose nothing by employing type inference: no program that was previously typeable will now cease to be so. That means we can focus on using explicit annotations where we want to, but will not be forced to do so.

Of course, this only holds if inference is decidable.

We might also expect that both versions type to the same type, but that is not a given: the function

```
(lambda ([x : number]) : number x)
```

types to `(number -> number)`, whereas applying inference to it after erasing types yields a much more general type. Therefore, relating these types and giving a precise definition of type equality is not trivial, though we will briefly return to this issue later [REF].

With these preliminaries out of the way, we are now ready to delve into the mechanics of type inference. The most important thing to note is that our simple, recursive-descent type-checking algorithm [REF] will no longer work. That was possible because we already had annotations on all function boundaries, so we could descend into function bodies carrying information about those annotations in the type environment. Sans these annotations, it is not clear how to descend.

In fact, it is not clear that there is any particular direction that makes more sense than another. In a definition like `mapper` above, each fragment of code influences the other. For instance, applying `empty?`, `cons?`, `first`, and `rest` to `l` all point to its being a list. But a list of what? We can't tell from any of those operations. However, the fact that we apply `f` to each (or indeed, any) `first` element means the list members must be of a type that can be passed to `f`. Similarly, we know the output must be a list because of `cons` and `empty`. But what are its elements? They must be the return type of `f`. Finally, note something very subtle: when the argument list is empty, we return `empty`, not `l` (which we know is bound to `empty` at that point). Using the former leaves the type of the return free to be any kind of list at all (constrained only by what `f` returns); using the latter would force it to be the same type as the argument list.

All this information is in the function. But how do we extract it systematically and in an algorithm that terminates and enjoys the property we have stated above? We do this in two steps. First we *generate constraints*, based on program terms, on what the types must be. Then we *solve constraints* to identify inconsistencies and join together constraints spread across the function body. Each step is relatively simple, but the combination creates magic.

Constraint Generation

Our goal, ultimately, is to find a type to fill into every type annotation position. It will prove to be just as well to find a type for every *expression*. A moment's thought will show that this is likely necessary anyway: for instance, how can we determine the type to put on a function without knowing the type of its body? It is also sufficient, in

that if every expression has had its type calculated, this will include the ones that need annotations.

First, we must generate constraints to (later) solve. Constraint generation walks the program source, emitting appropriate constraints on each expression, and returns this set of constraints. It works by recursive descent mainly for simplicity; it really computes a *set* of constraints, so the order of traversal and generation really does not matter in principle—so we may as well pick recursive descent, which is easy—though for simplicity we will use a list to represent this set.

What are constraints? They are simply statements about the types of expressions. In addition, though the binding instances of variables are not expressions, we must calculate their types too (because a function requires both argument and return types). In general, what can we say about the type of an expression?

1. That it is related to the type of some identifier.
2. That it is related to the type of some other expression.
3. That it is a number.
4. That it is a function, whose domain and range types are presumably further constrained.

Thus, we define the following two datatypes:

```
(define-type Constraints
  [eqCon (lhs : Term) (rhs : Term)])

(define-type Term
  [tExp (e : ExprC)]
  [tVar (s : symbol)]
  [tNum]
  [tArrow (dom : Term) (rng : Term)])
```

Now we can define the process of generating constraints:
<***constr-gen***> **::=**

```
(define (cg [e : ExprC]) : (listof Constraints)
  (type-case ExprC e
    <constr-gen-numC-case>
    <constr-gen-idC-case>
    <constr-gen-plusC/multC-case>
    <constr-gen-appC-case>
    <constr-gen-lamC-case>))
```

When the expression is a number, all we can say is that we expect the type of the expression to be numeric:
<***constr-gen-numC-case***> **::=**

```
[numC (_) (list (eqCon (tExp e) (tNum)))]
```

This might sound trivial, but what we don't know is what other expectations are being made of this expression by those containing it. Thus, there is the possibility that some outer expression will contradict the assertion that this expression's type must be numeric, leading to a type error.

For an identifier, we simply say that the type of the expression is whatever we expect to be the type of that identifier:

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

```
[idC (s) (list (eqCon (tExp e) (tVar s)))]
```

If the context limits its type, then this expression's type will automatically be limited, and must then be consistent with what its context expects of it.

Addition gives us our first look at a contextual constraint. For an addition expression, we must first make sure we generate (and return) constraints in the two sub-expressions, which might be complex. That done, what do we expect? That each of the sub-expressions be of numeric type. (If the form of one of the sub-expressions demands a type that is not numeric, this will lead to a type error.) Finally, we assert that the entire expression's type is itself numeric.

`append3` is just a three-argument version of `append`.

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

```
[plusC (l r) (append3 (cg l)
                      (cg r)
                      (list (eqCon (tExp l) (tNum))
                            (eqCon (tExp r) (tNum))
                            (eqCon (tExp e) (tNum))))]
```

The case for `multC` is identical other than the variant name.

Now we get to the other two interesting cases, function declaration and application. In both cases, we must remember to generate and return constraints of the sub-expressions.

In a function definition, the type of the function is a function ("arrow") type, whose argument type is that of the formal parameter, and whose return type is that of the body:

<*constr-gen-lamC-case*> **::=**

```
[lamC (a b) (append (cg b)
                    (list (eqCon (tExp e) (tArrow (tVar a) (tExp b)))))]
```

Finally, we have applications. We cannot directly state a constraint on the type of the application. Rather, we can say that the function in the application position must consume arguments of the actual parameter expression's type, and return types of the application expression's type:

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

```
[appC (f a) (append3 (cg f)
                     (cg a)
                     (list (eqCon (tExp f) (tArrow (tExp a) (tExp e)))))]
```

And that's it! We have finished generating constraints; now we just have to solve them.

Constraint Solving Using Unification

The process used to solve constraints is known as *unification*. A unifier is given a set of equations. Each equation maps a variable to a term, whose datatype is above. Note one subtle point: we actually have *two* kinds of variables. Both `tvar` and `tExp` are "variables", the former evidently so but the latter equally so because we need to solve for the types of these expressions. (An alternate formulation would introduce fresh type variables for each expression, but we would still need a way to identify which ones correspond to which expression, which `eq?` on the expressions already does automatically. Also, this would generate far larger constraint sets, making visual inspection daunting.)

For our purposes, the goal of unification is generate a *substitution*, or mapping from variables to terms that do not contain any variables. This should sound familiar: we have a set of simultaneous equations in which each variable is used linearly; such equations are solved using *Gaussian elimination*. In that context, we know that we can end up with systems that are both under- and over-constrained. The same thing can happen here, as we will soon see.

The unification algorithm works iteratively over the set of constraints. Because each constraint equation has two terms and each term can be one of four kinds, there are essentially sixteen cases to consider. Fortunately, we can cover all sixteen with fewer actual code cases.

The algorithm begins with the set of all constraints, and the empty substitution. Each constraint is considered once and removed from the set, so in principle the termination argument should be utterly simple, but it will prove to be only slightly more tricky in reality. As constraints are disposed, the substitution set tends to grow. When all constraints have been disposed, unification returns the final substitution set.

For a given constraint, the unifier examines the left-hand-side of the equation. If it is a variable, it is now ripe for elimination. The unifier adds the variable's right-hand-side to the substitution and, to truly eliminate it, replaces all occurrences of the variable in the substitution with the this right-hand-side. In practice this needs to be implemented efficiently; for instance, using a mutational representation of these variables can avoid having to search-and-replace all occurrences. However, in a setting where we might need to backtrack (as we will, in the presence of unification [REF]), the mutational implementation has its own disadvantages.

Do Now!

Did you notice the subtle error above?

The subtle error is this. We said that the unifier *eliminates* the variable by replacing all instances of it in the substitution. However, that assumes that the right-hand-side does not contain any instances of the same variable. Otherwise we have a circular definition, and it becomes impossible to perform this particular substitution. For this reason, unifiers include a *occurs check*: a check for whether the same variable occurs on both sides and, if it does, decline to unify.

Do Now!

Construct a term whose constraints would trigger the occurs check.

Do you remember ω?

Let us now consider the implementation of unification. It is traditional to denote the substitution by the Greek letter Θ.

```
(define-type-alias Subst (listof Substitution))
(define-type Substitution
  [sub [var : Term] [is : Term]])

(define (unify [cs : (listof Constraints)]) : Subst
  (unify/Θ cs empty))
```

Let's get the easy parts out of the way:

<***unify/Θ***> **::=**

```
(define (unify/Θ [cs : (listof Constraints)] [Θ : Subst]) : Subst
  (cond
    [(empty? cs) Θ]
    [(cons? cs)
     (let ([l (eqCon-lhs (first cs))]
           [r (eqCon-rhs (first cs))])
       (type-case Term l
         <unify/Θ-tVar-case>
         <unify/Θ-tExp-case>
         <unify/Θ-tNum-case>
         <unify/Θ-tArrow-case>))])))
```

Now we're ready for the heart of unification. We will depend on a function, `extend+replace`, with this signature: `(Term Term Subst -> Subst)`. We expect this to perform the occurs test and, if it fails (i.e., there is no circularity), extends the substituion and replaces all existing instances of the first term with the second in the substitution. Similarly, we will assume the existence of `lookup`: `(Term subst -> (optionof Term))`

Exercise

Define `extend+replace` and `lookup`.

If the left-hand of a constraint equation is a variable, we first look it up in the substitution. If it is present, we replace the current constraint with a new one; otherwise, we extend the substitution:

<***unify/Θ-tVar-case***> **::=**

```
[tVar (s) (type-case (optionof Term) (lookup l Θ)
            [some (bound)
                  (unify/Θ (cons (eqCon bound r)
                                 (rest cs))
                           Θ)]
```

```
            [none ()
                  (unify/Θ (rest cs)
                           (extend+replace l r Θ))])]
```

The same logic applies when it is an expression designator:
<***unify/Θ-tExp-case***> **::=**

```
[tExp (e) (type-case (optionof Term) (lookup l Θ)
            [some (bound)
                  (unify/Θ (cons (eqCon bound r)
                                 (rest cs))
                           Θ)]
            [none ()
                  (unify/Θ (rest cs)
                           (extend+replace l r Θ))])]
```

If it is a base type, such as a number, then we examine the right-hand side. There are four possibilities, for the four different kinds of terms:

- If it is a number, then we have an equation that claims that the type `num` is the same as the type `num`, which is patently true. We can therefore ignore this constraint—because it tells us nothing new—and move on to the remainder.

  You should, of course, question why such a constraint would have come about in the first place. Clearly, our constraint generator did not generate such constraints. However, a prior extension to the current substitution might have resulted in this situation. Indeed, in practice we will encounter several of these.

- If it is a function type, then we clearly have a type error, because numeric and function types are disjoint. Again, we would never have generated such a constraint directly, but it must have resulted from a prior substitution.

- It could have been one of the two variable kinds. However, we have carefully arranged our constraint generator to never put these on the right-hand-side. Furthermore, substitution will not introduce them on the right-hand-side, either. Therefore, these two cases cannot occur.

This results in the following code:
<***unify/Θ-tNum-case***> **::=**

```
[tNum () (type-case Term r
           [tNum () (unify/Θ (rest cs) Θ)]
           [else (error 'unify "number and something else")])]
```

Finally, we left with function types. Here the argument is almost exactly the same as for numeric types.
<***unify/Θ-tArrow-case***> **::=**

```
[tArrow (d r) (type-case Term r
                [tArrow (d2 r2)
                        (unify/Θ (cons (eqCon d d2)
                                       (cons (eqCon r r2)
                                             cs))
                                 Θ)]
                [else (error 'unify "arrow and something else")])]
```

Note that we do not always shrink the size of the constraint set, so a simple argument does not suffice for proving termination. Instead, we must make an argument based on the size of the constraint set, and on the size of the substitution (including the number of variables in it).

The algorithm above is very general in that it works for all sorts of type terms, not only numbers and functions. We have used numbers as a stand-in for all form of base types; functions, similarly, stand for all constructed types, such as `listof` and `vectorof`.

With this, we are done. Unification produces a substitution. We can now traverse the substitution and find the types of all the expressions in the program, then insert the type annotations accordingly. A theorem, which we will not prove here, dictates that the success of the above process implies that the program would have typed-checked, so we need not explicitly run the type-checker over this program.

Observe, however, that the nature of a type error has now changed dramatically. Previously, we had a recursive-descent algorithm that walked a expressions using a type environment. The bindings in the type environment were programmer-declared types, and could hence be taken as (intended) authoritative *specifications* of types. As a result, any mismatch was blamed on the expressions, and reporting type errors was simple (and easy to understand). Here, however, a type error is a *failure to notify*. The unification failure is based on events that occur at the confluence of two smart algorithms—constraint generation and unification—and hence are not necessarily comprehensible to the programmer. In particular, the equational nature of these constraints means that the location reported for the error, and the location of the "true" error, could be quite far apart. As a result, producing better error messages remains an active research area.

In practice the algorithm will maintain metadata on which program source terms were involved and probably on the history of unification, to be able to trace errors back to the source program.

Finally, remember that the constraints may not precisely dictate the type of all variables. If the system of equations is *over*-constrained, then we get clashes, resulting in type errors. If instead the system is *under*-constrained, that means we don't have enough information to make definitive statements about all expressions. For instance, in the expression `(lambda (x) x)` we do not have enough constraints to indicate what the type of `x`, and hence of the entire expression, must be. This is not an error; it simply means that `x` is free to be *any* type at all. In other words, its type is "the type of `x` `->` the type of `x`" with no other constraints. The types of these underconstrained identifiers are presented as type variables, so the above expression's type might be reported as `('a -> 'a)`.

The unification algorithm actually has a wonderful property: it automatically computes the *most general types* for an expression, also known as *principal types*. That is, any actual type the expression can have can be obtained by instantiating the inferred

type variables with actual types. This is a remarkable result: in another example of computers beating humans, it says that no human can generate a more general type than the above algorithm can!

Let-Polymorphism

Unfortunately, though these type variables are superficially similar to the polymorphism we had earlier [REF], they are not. Consider the following program:

```
(let ([id (lambda (x) x)])
  (if (id true)
      (id 5)
      (id 6)))
```

If we write it with explicit type annotations, it type-checks:

```
(if ((id boolean) true)
    ((id number) 5)
    ((id number) 6))
```

However, if we use type inference, it does not! That is because the `'a`'s in the type of `id` unify either with `boolean` or with `number`, depending on the order in which the constraints are processed. At that point `id` effectively becomes either a `(boolean -> boolean)` or `(number -> number)` function. At the use of `id` of the other type, then, we get a type error!

The reason for this is because the types we have inferred through unification are not actually *polymorphic*. This is important to remember: just because you type variables, you haven't seen polymorphism! The type variables could be unified at the next use, at which point you end up with a mere monomorphic function. Rather, true polymorphism only obtains when you have true *instantiation* of type variables.

In languages with true polymorphism, then, constraint generation and unification are not enough. Instead, languages like ML, Haskell, and even our typed programming language, implement something colloquially called *let-polymorphism*. In this strategy, when a term with type variables is bound in a lexical context, the type is automatically promoted to be a quantified one. At each use, the term is effectively automatically instantiated.

There are many implementation strategies that will accomplish this. The most naive (and unsatisfying) is to merely *copy the code* of the bound identifier; thus, each use of `id` above gets its own copy of `(lambda (x) x)`, so each gets its own type variables. The first might get the type `('a -> 'a)`, the second `('b -> 'b)`, the third `('c -> 'c)`, and so on. None of these type variables clash, so we get the effect of polymorphism. Obviously, this not only increases program size, it also does not work in the presence of recursion. However, it gives us insight into a better solution: instead of copying the code, why not just copy the *type*? Thus at each use, we create a renamed copy of the inferred type: `id`'s `('a -> 'a)` becomes `('b -> 'b)` at the first use, and so on, thus achieving the same effect as copying code but without its burdens. Because all these strategies effectively mimic copying code, however, they only work within a lexical context.

15.3.3 Union Types

Suppose we want to construct a list of zoo animals, of which there are many kinds: armadillos, boa constrictors, and so on. Currently, we are forced to create a new datatype:

"In Texas, there ain't nothing in the middle of the road but a yellow line and dead armadillos."—Jim Hightower

```
(define-type Animal
  [armadillo (alive? : boolean)]
  [boa (length : number)])
```

and make a list of these: `(listof Animal)`. The type `Animal` therefore represents a "union" of `armadillo` and `boa`, except the only way to construct such unions is to make a new type every time: if we want to represent the union of animals and plants, we need

```
(define-type LivingThings
  [animal (a : Animal)]
  [plant (p : Plant)])
```

so an actual animal is now one extra "level" deep. These datatypes are called *tagged unions* or *discriminated unions*, because we must introduce explicit tags (or *discriminators*), such as `animal` and `plant`, to tell them apart. In turn, a structure can only reside inside a datatype declaration; we have had to create datatypes with just one variant, such as

```
(define-type Constraints
  [eqCon (lhs : Term) (rhs : Term)])
```

to hold the datatype, and everywhere we've had to use the type `Constraints` because `eqCon` is not itself a type, only a variant that can be distinguished at run-time.

Either way, the point of a union type is to represent a disjunction, or "or". A value's type is one of the types in the union. A value usually belongs to only one of the types in the union, though this is a function of precisely how the union types are defined, whether there are rules for normalizing them, and so on.

Structures as Types

A natural reaction to this might be, why not lift this restriction? Why not allow each structure to exist on its own, and define a type to be a union of some collection of structures? After all, in languages ranging from C to Racket, programmers can define stand-alone structures without having to wrap them in some other type with a tag constructor! For instance, in raw Racket, we can write

```
(struct armadillo (alive?))
(struct boa (length))
```

and a comment that says

```
;; An Animal is either
;; - (armadillo <boolean>)
;; - (boa <number>)
```

but without enforced static types, the comparison is messy. However, we can more directly compare with *Typed Racket*, a typed form of Racket that is built into DrRacket. Here is the same typed code:

```
#lang typed/racket

(struct: armadillo ([alive? : Boolean]))
(struct: boa ([length : Real])) ;; feet
```

We can now define functions that consume values of type `boa` without any reference to `armadillos`:

```
;; http://en.wikipedia.org/wiki/Boa_constrictor#Size_and_weight
(define: (big-one? [b : boa]) : Boolean
  (> (boa-length b) 8))
```

In fact, if we apply this function to any other type, including an armadillo—`(big-one? (armadillo true))`—we get a *static* error. This is because armadillos are no more related to boas than numbers or strings are.

Of course, we can still define a union of these types:

```
(define-type Animal (U armadillo boa))
```

and functions over it:

```
(define: (safe-to-transport? [a : Animal]) : Boolean
  (cond
    [(boa? a) (not (big-one? a))]
    [(armadillo? a) (armadillo-alive? a)]))
```

Whereas before we had *one type with two variants*, now we have *three types*. It just so happens that two of the types form a union of convenience to define a third.

Untagged Unions

It might appear that we still need to have discriminative tags, but we don't. In languages with union types, the effect of the `optionof` type constructor is often obtained by combining the intended return type with a disjoint one representing failure or noneness. For instance, here is the moral equivalent of `(optionof number)`:

```
(define-type MaybeNumber (U Number Boolean))
```

For that matter, `Boolean` may itself be a union of `True` and `False`, as it is in Typed Racket, so a more accurate simulation of the option type would be:

```
(define-type MaybeNumber (U Number False))
```

More generally, we could define

```
(struct: none ())
(define-type (Maybeof T) (U T none))
```

which would work for all types, because `none` is a new, distinct type that cannot be confused for any other. This gives us the same benefit as the `optionof` type, except the value we want is not buried one level deep inside a `some` structure, but is rather available immediately. For instance, consider `member`, which has this Typed Racket type:

```
(All (a) (a (Listof a) -> (U False (Listof a))))
```

If the element is not found, `member` returns `false`. Otherwise, it returns the list starting from the element onward (i.e., the first element of the list will be the desired element):

```
> (member 2 (list 1 2 3))
'(2 3)
```

To convert this to use `Maybeof`, we can write

```
(define: (t) (in-list? [e : t] [l : (Listof t)]) : (Maybeof (Listof t))
  (let ([v (member e l)])
    (if v
        v
        (none))))
```

which, if the element is not found, returns the value `(none)`, but if it is found, still returns a list

```
> (in-list? 2 (list 1 2 3))
'(2 3)
```

so that there is no need to remove the list from a `some` wrapper.

Discriminating Untagged Unions

It's one thing to put values into unions; we have to also consider how to take them out, in a well-typed manner. In our ML-like type system, we use a stylized notation—`type-case` in our language, pattern-matching in ML—to identify and pull apart the pieces. In particular, when we write

```
(define (safe-to-transport? [a : Animal]) : boolean
  (type-case Animal a
    [armadillo (a?) a?]
    [boa (l) (not (big-one? l))]))
```

the type of `a` remains the same in the entire expression. The identifiers `a?` and `l` are bound to a boolean and numeric value, respectively, and `big-one?` must now be written to consume those types, not `armadillo` and `boa`. Put in different terms, we cannot have a function `big-one?` that consumes `boa`s, because there is no such type.

In contrast, with union types, we do have the `boa` type. Therefore, we follow the principle that the act of asking predicates of a value *narrows the type*. For instance, in the `cond` case

```
[(boa? a) (not (big-one? a))]
```

though `a` begins as type `Animal`, after it passes the `boa?` test, the type checker is expected to narrow its type to just the `boa` branch, so that the application of `big-one?`

is well-typed. In turn, in the rest of the conditional its type is *not* `boa`—in this case, that leaves only one possibility, `armadillo`. This puts greater pressure on the type-checker's ability to test and recognize certain patterns—known as *if-splitting*—without which it would be impossible to program with union types; but it can always default to recognizing just those patterns that the ML-like system would have recognized, such as pattern-matching or `type-case`.

Retrofitting Types

It is unsurprising that Typed Racket uses union types. They are especially useful when *retrofitting* types onto existing programming languages whose programs were not defined with an ML-like type discipline in mind, such as in scripting languages. A common principle of such retrofitted types is to statically catch as many dynamic exceptions as possible. Of course, the checker must ultimately reject some programs, and if it rejects too many programs that would have run without an error, developers are unlikely to adopt it. Because these programs were written without type-checking in mind, the type checker may therefore need to go to heroic lengths to accept what are considered reasonable idioms in the language.

Unless it implements an interesting idea called *soft typing*, which rejects no programs but provides information about points where the program would not have been typeable.

Consider the following JavaScript function:

```
var slice = function (arr, start, stop) {
  var result = [];
  for (var i = 0; i <= stop - start; i++) {
    result[i] = arr[start + i];
  }
  return result;
}
```

It consumes an array and two indices, and produces the sub-array between those indices. For instance, `slice([5, 7, 11, 13], 0, 2)` produces `[5, 7, 11]`.

In JavaScript, however, developers are free to leave out any or all trailing arguments to a function. Every elided argument is given a special value, `undefined`, and it is up to the function to cope with this. For instance, a typical implementation of `splice` would let the user drop the third argument; the following definition

```
var slice = function (arr, start, stop) {
  if (typeof stop == "undefined")
    stop = arr.length - 1;
  var result = [];
  for (var i = 0; i <= stop - start; i++) {
    result[i] = arr[start + i];
  }
  return result;
}
```

automatically returns the subarray until the end of the array: thus, `slice([5, 7, 11, 13], 2)` returns `[11, 13]`.

In Typed JavaScript, a programmer can explicitly indicate a function's willingness to accept fewer arguments by giving a parameter the type `U Undefined`, giving it the type

Built at Brown by Arjun Guha and others. See our Web site.

```
∀ t : (Array[t] * Int * (Int U Undefined) -> Array[t])
```

In principle, this means there is a potential type error at the expression `stop - start`, because `stop` may not be a number. However, the assignment to `stop` sets it to a numeric type precisely when it was elided by the user. In other words, in all control paths, `stop` will eventually have a numeric type before the subtraction occurs, so this function is well-typed. Of course, this requires the type-checker to be able to reason about both control-flow (through the conditional) and state (through the assignment) to ensure that this function is well-typed; but Typed JavaScript can, and can thus bless functions such as this.

Design Choices

In languages with union types, it is common to have

- Stand-alone structure types (often represented using classes), rather than datatypes with variants.
- Ad hoc collections of structures to represent particular types.
- The use of sentinel values to represent failure.

To convert programs written in this style to an ML-like type discipline would be extremely onerous. Therefore, many retrofitted type systems adopt union types to ease the process of typing.

Of the three properties above, the first seems morally neutral, but the other two warrant more discussion. We will address them in reverse order.

- Let's tackle sentinels first. In many cases, sentinels ought to be replaced with exceptions, but in many languages, exceptions can be very costly. Thus, developers prefer to make a distinction between truly exceptional situations—that ought not occur—and situations that are expected in the normal course of operation. Checking whether an element is in a list and failing to find it is clearly in the latter category (if we already knew the element was or wasn't present, there would be no need to run this predicate). In the latter case, using sentinels is reasonable.

  However, we must square this with the observation that failure to check for exceptional sentinel values is a common source of error—and indeed, security flaws—in C programs. This is easy to reconcile. In C, the sentinel is of the *same type* (or at least, effectively the same type) as the regular return value, and furthermore, there are no run-time checks. Therefore, the sentinel can be used as a legitimate value without a type error. As a result, a sentinel of `0` can be treated as an address into which to allocate data, thus potentially crashing the system. In contrast, our sentinel is of a truly new type that cannot be used in any computation. We can easily reason about this by observing that no existing functions in our language consume values of type `none`.

- Setting aside the use of "ad hoc", which is pejorative, are different groupings of a set of structures a good idea? In fact, such groupings occur even in programs using an ML-like discipline, when programmers want to carve different subuniverses of a larger one. For instance, ML programmers use a type like

```
(define-type SExp
  [numSexp (n : number)]
  [strSexp (s : string)]
  [listSexp (l : (listof SExp))])
```

to represent s-expressions. If a function now operates on just some subset of these terms—say just numbers and lists of numbers—they must create a fresh type, and convert values between the two types even though their underlying representations are essentially identical. As another example, consider the set of CPS expressions. This is clearly a subset of all possible expressions, but if we were to create a fresh datatype for it, we would not be able to use any existing programs that process expressions—such as the interpreter.

In other words, union types appear to be a reasonable variation on the ML-style type system we have seen earlier. However, even within union types there are design variations, and these have consequences. For instance, can the type system create new unions, or are user-defined (and named) unions permitted? That is, can an expression like this

```
(if (phase-of-the-moon)
    10
    true)
```

be allowed to type (to `(U Number Boolean)`), or is it a type error to introduce unions that have not previously been named and explicitly identified? Typed Racket provides the former: it will construct truly ad hoc unions. This is arguably better for importing existing code into a typed setting, because it is more flexible. However, it is less clear whether this is a good design for writing new code, because unions the programmer did not intend can occur and there is no way to prevent them. This offers an unexplored corner in the design space of programming languages.

15.3.4 Nominal Versus Structural Systems

In our initial type-checker, two types were considered equivalent if they had the same structure. In fact, we offered no mechanism for naming types at all, so it is not clear what alternative we had.

Now consider Typed Racket. A developer can write

```
(define-type NB1 (U Number Boolean))
(define-type NB2 (U Number Boolean))
```

followed by

```
(define: v : NB1 5)
```

Suppose the developer also defines the function

```
(define: (f [x : NB2]) : NB2 x)
```

and tries to apply `f` to `v`, i.e., `(f v)`: should this application type or not?

There are two perfectly reasonable interpretations. One is to say that `v` was declared to be of type `NB1`, which is a different *name* than `NB2`, and hence should be considered a different *type*, so the above application should result in an error. Such a system is called *nominal*, because the name of a type is paramount for determining type equality.

In contrast, another interpretation is that because the *structure* of `NB1` and `NB2` are identical, there is no way for a developer to write a program that behaves differently on values of these two types, so these two types should be considered identical. Such a type system is called *structural*, and would successfully type the above expression. (Typed Racket follows a structural discipline, again to reduce the burden of importing existing untyped code, which—in Racket—is usually written with a structural interpretation in mind. In fact, Typed Racket not only types `(f v)`, it prints the result as having type `NB1`, despite the return type annotation on `f`!)

If you want to get especially careful, you would note that there is a difference between being considered the same and actually being the same. We won't go into this issue here, but consider the implication for a compiler writer choosing representations of values, especially in a language that allows run-time inspection of the static types of values.

The difference between nominal and structural typing is most commonly contentious in object-oriented languages, and we will return to this issue briefly later [REF]. However, the point of this section is to illustrate that these questions are not intrinsically about "objects". Any language that permits types to be named—as all must, for programmer sanity—must contend with this question: is naming merely a convenience, or are the choices of names intended to be meaningful? Choosing the former answer leads to structural typing, while choosing the latter leads down the nominal path.

15.3.5 Intersection Types

Since we've just explored union types, you must naturally wonder whether there are also *intersection* types. Indeed there are.

If a union type means that a value (of that type) belongs to one of the types in the union, an intersection type clearly means the value belongs to *all* the types in the intersection: a conjunction, or "and". This might seem strange: how can a value belong to more than one type?

As a concrete answer, consider *overloaded functions*. For instance, in some languages + operates on both numbers and strings; given two numbers it produces a number, and given two strings it produces a string. In such a language, what is the proper type for +? It is not `(number number -> number)` alone, because that would reject its use on strings. By the same reasoning, it is not `(string string -> string)` alone either. It is not even

```
(U (number number -> number)
   (string string -> string))
```

because + is not just one of these functions: it truly is both of them. We could ascribe the type

```
((number U string) (number U string) -> (number U string))
```

reflecting the fact that each argument, and the result, can be only one of these types, not both. Doing so, however, leads to a loss of precision.

Do Now!

In what way does this type lose precision?

Observe that with this type, the return type on *all* invocations is `(number U string)`. Thus, on every return we must distinguish beween numeric and string returns, or else we will get a type error. Thus, even though we know that if given two numeric arguments we will get a numeric result, this information is lost to the type system.

More subtly, this type permits each argument's type to be chosen independently of the other. Thus, according to this type, the invocation `(+ 3 "x")` is perfectly valid (and produces a value of type `(number U string)`). But of course the addition operation we have specified is not defined for these inputs at all!

Thus the proper type to ascribe this form of addition is

```
(∧ (number number -> number)
   (string string -> string))
```

where ∧ should be reminiscent of the conjunction operator in logic. This permits invocation with two numbers or two strings, but nothing else. An invocation with two numbers has a numeric result type; one with two strings has a string result type; and nothing else. This corresponds precisely to our intended behavior for overloading (sometimes also called *ad hoc polymorphism*). Observe that this only handles a finite number of overloaded cases.

15.3.6 Recursive Types

Now that we've seen union types, it pays to return to our original recursive datatype formulation. If we accept the variants as type constructors, can we write the recursive type as a union over these? For instance, returning to `BTnum`, shouldn't we be able to describe it as equivalent to

```
((BTmt) U (BTnd number BTnum BTnum))
```

thereby showing that `BTmt` is a zero-ary constructor, and `BTnd` takes three parameters? Except, what are the types of those three parameters? In the type we've written above, `BTnum` is either built into the type language (which is unsatisfactory) or unbound. Perhaps we mean

```
BTnum = ((BTmt) U (BTnd number BTnum BTnum))
```

Except now we have an equation that has no obvious solution (remember ω?).

This situation should be familiar from recursion in values [REF]. Then, we invented a recursive function constructor (and showed its implementation) to circumvent this problem. We similarly need a recursive *type* constructor. This is conventionally called μ (the Greek letter "mu"). With it, we would write the above type as

```
μ BTnum : ((BTmt) U (BTnd number BTnum BTnum))
```

μ is a binding construct; it binds `BTnum` to the entire type written after it, including the recursive binding of `BTnum` itself. In practice, of course, this entire recursive type is the one we wish to call `BTnum`:

```
BTnum = μ BTnum : ((BTmt) U (BTnd number BTnum BTnum))
```

Though this looks like a circular definition, notice that the name `BTnum` on the right does not depend on the one to the left of the equation: i.e., we could rewrite this as

```
BTnum = μ T : ((BTmt) U (BTnd number T T))
```

In other words, this definition of `BTnum` truly can be thought of as syntactic sugar and replaced everywhere in the program without fear of infinite regress.

At a semantic level, there are usually two very different ways of thinking about the meaning of types bound by μ: they can be interpreted as *isorecursive* or *equirecursive*. The distinction between these is, however, subtle and beyond the scope of this chapter. It suffices to note that a recursive type can be treated as equivalent to its unfolding. For instance, if we define a numeric list type as

This material is covered especially well in Pierce's book.

```
NumL = μ T : ((MtL) U (ConsL number T))
```

then

```
  μ T : ((MtL) U (ConsL number T))
= (MtL) U (ConsL number (μ T : ((MtL) U (ConsL number T))))
= (MtL) U (ConsL number (MtL))
        U (ConsL number (ConsL number (μ T : ((MtL) U (ConsL number T)))))
```

and so on (iso- and equi-recursiveness differ in precisely what the notion of equality is: definitional equality or isomorphism). At each step we simply replace the `T` parameter with the entire type. As with value recursion, this means we can "get another" `ConsL` constructor upon demand. Put differently, the *type* of a list can be written as the union of zero or arbitrarily many elements; this is the same as the *type* that consists of zero, one, or arbitrarily many elements; and so on. Any lists of numbers fits all (and precisely) these types.

Observe that even with this informal understanding of μ, we can now provide a type to ω, and hence to Ω.

Exercise

Ascribe types to ω and Ω.

15.3.7 Subtyping

Imagine we have a typical binary tree definition; for simplicity, we'll assume that all the values are numbers. We will write this in Typed Racket to illustrate a point:

```
#lang typed/racket

(define-struct: mt ())
(define-struct: nd ([v : Number] [l : BT] [r : BT]))
(define-type BT (U mt nd))
```

Now consider some concrete tree values:

```
> (mt)
- : mt
#<mt>
> (nd 5 (mt) (mt))
- : nd
#<nd>
```
Observe that each structure constructor makes a value of its own type, not a value of type `BT`. But consider the expression `(nd 5 (mt) (mt))`: the definition of `nd` declares that the sub-trees must be of type `BT`, and yet we are able to successfully give it values of type `mt`.

Obviously, it is not coincidental that we have defined `BT` in terms of `mt` and `nd`. However, it does indicate that when type-checking, we cannot simply be checking for function equality, at least not as we have so far. Instead, we must be checking that one type "fits into" the other. This notion of fitting is called *subtyping* (and the act of being fit, *subsumption*).

The essence of subtyping is to define a relation, usually denoted by `<:`, that relates pairs of types. We say `S <: T` if a value of type `S` can be given where a value of type `T` is expected: in other words, subtyping formalizes the notion of *substitutability* (i.e., anywhere a value of type `T` was expected, it can be replaced with—substituted by—a value of type `S`). When this holds, `S` is called the *subtype* and `T` the *supertype*. It is useful (and usually accurate) to take a subset interpretation: if the values of `S` are a subset of `T`, then an expression expecting `T` values will not be unpleasantly surprised to receive only `S` values.

Subtyping has a pervasive effect on the type system. We have to reexamine every kind of type and understand its interaction with subtyping. For base types, this is usually quite obvious: disjoint types like `number`, `string`, etc., are all unrelated to each other. (In languages where one base type is used to represent another—for instance, in some scripting languages numbers are merely strings written with a special syntax, and in other languages, booleans are merely numbers—there might be subtyping relationships even between base types, but these are not common.) However, we do have to consider how subtyping interacts with every single compound type constructor.

In fact, even our very diction about types has to change. Suppose we have an expression of type `T`. Normally, we would say that it produces values of type `T`. Now, we should be careful to say that it produces values of *up to* or *at most* `T`, because it may only produce values of a subtype of `T`. Thus every reference to a type should implicitly be cloaked in a reference to the potential for subtyping. To avoid pestering you I will refrain from doing this, but be wary that it is possible to make reasoning errors by not keeping this implicit interpretation in mind.

Unions

Let us see how unions interact with subtyping. Clearly, every sub-union is a subtype of the entire union. In our running example, clearly every `mt` value is also a `BT`; likewise for `nd`. Thus,

```
mt <: BT
nd <: BT
```

As a result, `(mt)` also has type `BT`, thus enabling the expression `(nd 5 (mt) (mt))` to itself type, and to have the type `nd`—and hence, also the type `BT`. In general,

```
S <: (S U T)
T <: (S U T)
```

(we write what seems to be the same rule twice just to make clear it doesn't matter which "side" of the union the subtype is on). This says that a value of `S` can be thought of as a value of `S U T`, because any expression of type `S U T` can indeed contain a value of type `S`.

Intersections

While we're at it, we should also briefly visit intersections. As you might imagine, intersections behave dually:

```
(S ∧ T) <: S
(S ∧ T) <: T
```

To convince yourself of this, take the subset interpretation: if a value is both `S` and `T`, then clearly it is either one of them.

Do Now!

Why are the following two *not* valid subsumptions?

1. `(S U T) <: S`
2. `T <: (S ∧ T)`

The first is not valid because a value of type `T` is a perfectly valid element of type `(S U T)`. For instance, a number is a member of type `(string U number)`. However, a number cannot be supplied where a value of type `string` is expected.

As for the second, in general, a value of type `T` is not also a value of type `S`. Any consumer of a `(S ∧ T)` value is expecting to be able to treat it as both a `T` and a `S`, and the latter is not justified. For instance, given our overloaded + from before, if `T` is `(number number -> number)`, then a function of this type will not know how to operate on strings.

Functions

We have seen one more constructor: functions. We must therefore determine the rules for subtyping when either type can be a function. Since we usually assume functions are disjoint from all other types, we therefore only need to consider when one function type is a subtype of another: i.e., when is

We have also seen parametric datatypes. In this edition, exploring subtyping for them is left as an exercise for the reader.

```
(S1 -> T1) <: (S2 -> T2)
```

? For convenience, let us call the type `(S1 -> T1)` as `f1`, and `(S2 -> T2)` as `f2`. The question then is, if an expression is expecting functions of the `f2` type, when can we safely give it functions with the `f1` type? It is easiest to think through this using the subset interpretation.

Consider a use of the `f2` type. It returns values of type `T2`. Thus, the context surrounding the function application is satisfied with values of type `T2`. Clearly, if

`T1` is the same as `T2`, the use of `f2` would continue to type; similarly, if `T1` consists of a subset of `T2` values, it would still be fine. The only problem is if `T1` has more values than `T2`, because the context would then encounter unexpected values that would result in undefined behavior. In other words, we need that `T1 <: T2`. Observe that the "direction" of containment is the same as that for the entire function type; this is called *covariance* (both vary in the same direction). This is perhaps precisely what you expected.

By the same token, you might expect covariance in the argument position as well: namely, that `S1 <: S2`. This would be predictable, and wrong. Let's see why.

An application of a function with `f2` type is providing parameter values of type `S2`. Suppose we instead substitute the function with one of type `f1`. If we had that `S1 <: S2`, that would mean that the new function accepts only values of type`S1`—a strictly smaller set. That means there may be some inputs—specifically those in `S2` that are not in `S1`—that the application is free to provide on which the substituted function is not defined, again resulting in undefined behavior. To avoid this, we have to make the subsumption go in the other direction: the substituting function should accept at least as many inputs as the one it replaces. Thus we need `S2 <: S1`, and say the function position is *contravariant*: it goes against the direction of subtyping.

Putting together these two observations, we obtain a subtyping rule for functions (and hence also methods):

```
(S2 <: S1) and (T1 <: T2) => (S1 -> T1) <: (S2 -> T2)
```

Implementing Subtyping

Of course, these rules assume that we have modified the type-checker to respect subtyping. The essence of subtyping is a rule that says, if an expression `e` is of type `S`, and `S <: T`, then `e` also has type `T`. While this sounds intuitive, it is also immediately problematic for two reasons:

- Until now all of our type rules have been syntax-driven, which is what enabled us to write a recursive-descent type-checker. Now, however, we have a rule that applies to *all* expressions, so we can no longer be sure when to apply it.
- There could be many levels of subtyping. As a result, it is no longer obvious when to "stop" subtyping. In particular, whereas before type-checking was able to calculate the type of an expression, now we have many possible types for each expression; if we return the "wrong" one, we might get a type error (due to that not being the type expected by the context) even though there exists some other type that was the one expected by the context.

What these two issues point to is that the description of subtyping we are giving here is fundamentally *declarative*: we are saying what must be true, but not showing how to turn it into an algorithm. For each actual type language, there is a less or more interesting problem in turning this into *algorithmic subtyping*: an actual algorithm that realizes a type-checker (ideally one that types exactly those programs that would have typed under the declarative regime, i.e., one that is both sound and complete).

15.3.8 Object Types

As we've mentioned earlier, types for objects are typically riven into two camps: nominal and structural. Nominal types are familiar to most programmers through Java, so I won't say much about them here. Structural types for objects dictate that an object's type is itself a structured object, consisting of names of fields and their types. For instance, an object with two methods, `add1` and `sub1` [REF], would have the type

```
{add1 : (number -> number), sub1 : (number -> number)}
```

(For future reference, let's call this type `addsub`.) Type-checking would then follow along predictable lines: for field access we would simply ensure the field exists and would use its declared type for the dereference expression; for method invocation we would have to ensure not only that the member exists but that it has a function type. So far, so straightforward.

Object types become complicated for many reasons:

- Self-reference. What is the type of `self`? It must the same type as the object itself, since any operation that can be applied to the object from the "outside" can also be applied to it from the "inside" using `self`. This means object types are recursive types.
- Access controls: private, public, and other restrictions. These lead to a distinction in the type of an object from "outside" and "inside".
- Inheritance. Not only do we have to give a type to the parent object(s), what is visible along inheritance paths may, again, differ from what is visible from the "outside".
- The interplay between multiple-inheritance and subtyping.
- The relationship between classes and interfaces in languages like Java, which has a run-time cost.
- Mutation.
- Casts.
- Snakes on a plane.

Whole books are therefore devoted to this topic. Abadi and Carelli's *A Theory of Objects* is important but now somewhat dated. Bruce's *Foundations of Object-Oriented Languages: Types and Semantics* is more modern, and also offers more gentle exposition. Pierce covers all the necessary theory beautifully.

and so on. Some of these problems simplify in the presence of nominal types, because given a type's name we can determine everything about its behavior (the type declarations effectively become a dictionary through which the object's description can be looked up on demand), which is one argument in favor of nominal typing.

A full exposition of these issues will take much more room than we have here. For now, we will limit ourselves to one interesting question. Remember that we said subtyping forces us to consider every type constructor? The structural typing of objects introduces one more: the object type constructor. We therefore have to understand its interaction with subtyping.

Note that Java's approach is not the only way to build a nominal type system. We have already argued that Java's class system needlessly restricts the expressive power of programmers [REF]; in turn, Java's nominal type system needlessly conflates types (which are interface descriptions) and implementations. It is, therefore, possible to have much better

Before we do, let's make sure we understand what an object type even means. Consider the type `addsub` above, which lists two methods. What objects can be given this type? Obviously, an object with just those two methods, with precisely those two types, is eligible. Equally obviously, an object with only one and not the other of those two methods, no matter what else it has, is not. But the phrase "no matter what else it has" is meant to be leading. What if an object represents an arithmetic package that also contains methods `+` and `*`, in addition to the above two (all of the appropriate type)? In that case we certainly have an object that can supply those two methods, so the arithmetic package certainly has type `addsub`. Its other methods are simply inaccessible using type `addsub`.

Let us write out the type of this package, in full, and call this type `as+*`:

```
{add1  : (number -> number),
 sub1  : (number -> number),
 +     : (number number -> number),
 *     : (number number -> number)}
```

What we have just argued is that an object of type `as+*` should also be allowed to claim the type `addsub`, which means it can be substituted in any context expecting a value of type `addsub`. In other words, we have just said that we want `as+* <: addsub`:

```
{add1  : (number -> number),           {add1 : (number -> number),
 sub1  : (number -> number),       <:   sub1 : (number -> number)}
 +     : (number number -> number),
 *     : (number number -> number)}
```

This may momentarily look confusing: we've said that subtyping follows set inclusion, so we would expect the smaller set on the left and the larger set on the right. Yet, it looks like we have a "larger type" (certainly in terms of character count) on the left and a "smaller type" on the right.

To understand why this is sound, it helps to develop the intuition that the "larger" the type, the fewer values it can have. Every object that has the four methods on the left clearly also has the two methods on the right. However, there are many objects that have the two methods on the right that fail to have all four on the left. If we think of a type as a constraint on acceptable value shapes, the "bigger" type imposes more constraints and hence admits fewer values. Thus, though the *types* may appear to be of the wrong sizes, everything is well because the sets of values they subscribe are of the expected sizes.

More generally, this says that by dropping fields from an object's type, we obtain a supertype. This is called *width subtyping*, because the subtype is "wider", and we move up the subtyping hierarchy by adjusting the object's "width". We see this even in the nominal world of Java: as we go up the inheritance chain a class has fewer and fewer methods and fields, until we reach `Object`, the supertype of all classes, which has the fewest. Thus for all class types `C` in Java, `C <: Object`.

As you might expect, there is another important form of subtyping, which is *within* a given member. This simply says that any particular member can be subsumed to a

Somewhat confusingly, the terms *narrowing* and *widening* are sometimes used, but with what some might consider the opposite meaning. To widen is to go from subtype to supertype, because it goes from a "narrower" (smaller) to a "wider" (bigger) set. These terms evolved

supertype in its corresponding position. For obvious reasons, this form is called *depth subtyping*.

Exercise

> Construct two examples of depth subtyping. In one, give the field itself an object type, and use width subtyping to subtype that field. In the other, give the field a function type.

Java has limited depth subtyping, preferring types to be *invariant* down the object hierarchy because this is a safe option for conventional mutation.

The combination of width and depth subtyping cover the most interesting cases of object subtyping. A type system that implemented only these two would, however, needlessly annoy programmers. Other convenient (and mathematically necessary) rules include the ability to permute names, reflexivity (every type is a subtype of itself, because it is more convenient to interpret the subtype relationship as $\subseteq$), and transitivity. Languages like Typed JavaScript employ all these features to provide maximum flexibility to programmers.

16 Checking Program Invariants Dynamically: Contracts

Contents

Type systems offer rich and valuable ways to represent program invariants. However, they also represent an important trade-off, because not all non-trivial properties of programs can be verified statically. Furthermore, even if we can devise a method to settle a certain property statically, the burdens of annotation and computational complexity may be too great. Thus, it is inevitable that some of the properties we care about must either be ignored or settled only at run-time. Here, we will discuss run-time enforcement.

This is a formal property, known as Rice's Theorem.

Virtually every programming language has some form of assertion mechanism that enables programmers to write properties that are richer than the language's static type system permits. In languages without static types, these properties might start with simple type-like assertions: whether a parameter is numeric, for instance. However, the language of assertions is often the entire programming language, so any predicate can be used as an assertion: for instance, an implementation of a cryptography package might want to ensure certain parameters pass a primality test, or a balanced binary search-tree might want to ensure that its subtrees are indeed balanced and preserve the search-tree ordering.

16.1 Contracts as Predicates

It is therefore easy to see how to implement simple contracts. A contract embodies a predicate. It consumes a value and applies the predicate to the value. If the value passes the predicate, the contract returns the value unmolested; if the value fails, the contract reports an error. Its only behaviors are to return the supplied value or to error: it should not change the value in any way. In short, on values that pass the predicate, the contact itself acts as the identity function.

In what follows we will use the language `#lang plai`, for two reasons. First, this better simulates programming in an untyped language. Second, for simplicity we will write type-like assertions as contracts, but in the typed language these will be flagged by the type-checker itself, not letting us see the run-time behavior. In effect, it is easier to "turn off" the type checker. However, contracts make perfect sense even in a typed world, because they enhance the set of invariants that a programmer can express.

We can encode this essence in the following function:

```
(define (make-contract pred?)
  (lambda (val)
    (if (pred? val) val (blame "violation"))))

(define (blame s) (error 'contract "~a" s))
```

Here's an example contract:

```
(define non-neg?-contract
  (make-contract
   (lambda (n) (and (number? n)
                    (>= n 0)))))
```

(In a typed language, the `number?` check would of course be unnecessary because it can be encoded—and statically checked!—in the type of the function using the contract.) Suppose we want to make sure we don't get imaginary numbers when computing square roots; we might write

```
(define (real-sqrt-1 x)
  (sqrt (non-neg?-contract x)))
```

In many languages assertions are written as statements rather than as expressions, so an alternate way to write this would be:

```
(define (real-sqrt-2 x)
  (begin
    (non-neg?-contract x)
    (sqrt x)))
```

(In some cases this form is clearer because it states crisply at the beginning of the function what is expected of the parameters. It also enables parameters to be checked just once. Indeed, in some languages the contract can be written in the function header itself, thereby improving the information given in the interface.) Now if `real-sqrt-1` or `real-sqrt-2` are applied to `4` they produce `2`, but if applied to `-1` they raise a contract violation error.

16.2 Tags, Types, and Observations on Values

At this point we've reproduced the essence of assertion systems in most languages. What else is there to say? Let's suppose for a moment that our language is not statically typed. Then we will want to write assertions that reproduce at least traditional type-like invariants, if not more. `make-contract` above can capture all standard type-like properties such as checking for numbers, strings, and so on, assuming the appropriate predicates are either provided by the language or can be fashioned from the ones given. Or can it?

Recall that even our simplest type language had not just base types, like numbers, but also constructed types. While some of these, like lists and vectors, appear to not be very challenging, they are once we care about mutation, performance, and blame, which we discuss below. However, functions are immediately problematic.

As a working example, we will take the following function:

```
(define d/dx
  (lambda (f)
    (lambda (x)
      (/ (- (f (+ x 0.001))
            (f x))
         0.001))))
```

Statically, we would give this the type

```
((number -> number) -> (number -> number))
```

(it consumes a function, and produces its derivative—another function). Let us suppose we want to guard this with contracts.

The fundamental problem is that in most languages, we cannot directly express this as a predicate. Most language run-time systems store very limited information about the types of values—so limited that, relative to the types we have seen so far, we should use a different name to describe this information; traditionally they are called

tags. Sometimes tags coincide with what we might regard as types: for instance, a number will have a tag identifying it as a number (perhaps even a specific kind of number), a string will have a tag identifying it as a string, and so forth. Thus we can write predicates based on the values of these tags.

When we get to structured values, however, the situation is more complex. A vector would have a tag declaring it to be a vector, but not dictating what kinds of values its elements are (and they may not even all be of the same kind); however, a program can usually also obtain its size, and thus traverse it, to gather this information. (There is, however, more to be said about structured values below [REF].)

There have been a few efforts to preserve rich type information from the source program through lower levels of abstraction all the way down to assembly language, but these are research efforts.

Do Now!

Write a contract that checks that a list consists solely of even numbers.

Here it is:

```
(define list-of-even?-contract
  (make-contract
   (lambda (l)
     (and (list? l) (andmap number? l) (andmap even? l)))))
```

(Again, note that the first two questions need not be asked if we know, statically, that we have a list of numbers.) Similarly, an object might simply identify itself as an object, not providing additional information. But in languages that permit reflection on the object's structure, a contract can still gather the information it needs.

In every language, however, this becomes problematic when we encounter functions. We might think of a function as having a type for its domain and range, but to a run-time system, a function is just an opaque object with a function tag, and perhaps some very limited metadata (such as the function's arity). The run-time system can hardly even tell whether the function consumes and produces functions—as opposed to other kinds of values—much less whether it consumes and produces ones of `(number -> number)` type.

This problem is nicely embodied in the (misnamed) `typeof` operator in JavaScript. Given values of base types like numbers and strings, `typeof` returns a string to that effect (e.g., `"number"`). For objects, it returns `"object"`. Most importantly, for functions it returns `"function"`, with no additional information.

For this reason, perhaps `typeof` is a bad name for this operator. It should have been called `tagof` instead, leaving open the possibility that future static type systems for JavaScript could provide a true `typeof`.

To summarize, this means that at the point of being confronted with a function, a function contract can only check that it is, indeed, a function (and if it is not, that is clearly an error). It cannot check anything about the domain and range of the function. Should we give up?

16.3 Higher-Order Contracts

To determine what to do, it helps to recall what sort of guarantee contracts provide in the first place. In `real-sqrt-1` above, we demanded that the argument be non-negative. However, this is only checked if—and when—`real-sqrt-1` is actually used, and then only on the actual values that are passed to it. For instance, if the program contains this fragment

```
(lambda () (real-sqrt-1 -1))
```

but this thunk is never invoked, the programmer would never see this contract violation. In fact, it may be that the thunk is not invoked on this run of the program, but in a later run it will be; thus, the program has a lurking contract error. For this reason, it is usually preferable to express invariants through static types; but where we do use contracts, we understand that it is with the caveat that we will only be notified of errors when the program is suitably exercised.

This is a useful insight, because it offers a solution to our problem with functions. We check, immediately, that the purported function value truly is a function. However, instead of ignoring the domain and range contracts, we *defer* them. We check the domain contract when (and each time) the function is actually applied to a value, and we check the range contract when the function actually returns a value.

This is clearly a different pattern than `make-contract` followed. Thus, we should give `make-contract` a more descriptive name: it checks *immediate* contracts (i.e., those that can be checked in their entirety now).

In the Racket contract system, immediate contracts are called *flat*. This term is slightly misleading, since they can also protect data structures.

```
(define (immediate pred?)
  (lambda (val)
    (if (pred? val) val (blame val))))
```

In contrast, a function contract takes two *contracts* as arguments—representing checks to be made on the domain and range—and returns a predicate. This is the predicate to apply on values purporting to satisfy that contract. First, this checks that the given value actually is a function: this part is still immediate. Then, we create a *surrogate* procedure that applies the "residual" contracts—to check the domain and range—but otherwise behaves the same as the original function.

This creation of a surrogate represents a departure from the traditional assertion mechanism, which simply checks values and then leaves them alone. Instead, for functions we must use the created surrogate if we want contract checking. In general, therefore, it is useful to have a wrapper that consumes a contract and value, and creates a guarded version of that value:

```
(define (guard ctc val) (ctc val))
```

As a very simple example, let us suppose we want to wrap the `add1` function in numeric contracts (with `function`, the constructor of function contracts, to be defined momentarily):

```
(define a1 (guard (function (immediate number?)
                            (immediate number?))
                  add1))
```

We want `a1` to be bound to essentially the following code:

```
(define a1
  (lambda (x)
    (num?-con (add1 (num?-con x)))))
```

Here, the `(lambda (x) ...)` is the surrogate; it applies two numeric contracts around the invocation of `add1`. Recall that contracts must behave like the identity function in the absence of violations, so this procedure has precisely the same behavior as `add1` on non-violating uses.

To achieve this, we use the following definition of `function`. Remember that we have to also ensure that the given value is truly a function (as `add1` above indeed is, and can be checked immediately, which is why the check has disappeared by the time we bind the surrogate to `a1`):

For simplicity we assume single-argument functions here, but the extension to multiple arity is straightforward. Indeed, more complex contracts can even check for relationships *between* the arguments.

```
(define (function dom rng)
  (lambda (val)
    (if (procedure? val)
        (lambda (x) (rng (val (dom x))))
        (blame val))))
```

To understand how this works, let us substitute arguments. To keep the resulting code readable, we will first construct the `number?` contract checker and give it a name:

```
  (define num?-con (immediate number?))
= (define num?-con
    (lambda (val)
      (if (number? val) val (blame val))))
```

Now let's return to the definition of `a1`. First we apply `guard`:

```
(define a1
  ((function num?-con num?-con)
   add1))
```

Now we apply the function contract constructor:

```
(define a1
  ((lambda (val)
     (if (procedure? val)
         (lambda (x) (num?-con (val (num?-con x))))
         (blame val)))
   add1))
```

Applying the left-left-lambda gives:

```
(define a1
  (if (procedure? add1)
      (lambda (x) (num?-con (add1 (num?-con x))))
      (blame add1)))
```

Notice that this immediately checks that the guarded value is indeed a function. Thus we get

```
(define a1
  (lambda (x)
    (num?-con (add1 (num?-con x)))))
```

which is precisely the surrogate we desired, with the behavior of `add1` on non-violating executions.

Do Now!

How many ways are there to violate the above contract for `add1`?

There are three ways, corresponding to the three contract constructors:

1. the value wrapped might not be a function;
2. the wrapped value might be a function that is applied to a non-numeric value; or,
3. the wrapped value might be a function that consumes numbers but produces values of non-numeric type.

Exercise

Write examples that perform each of these three violations, and observe the behavior of the contract system. Can you improve the error messages to better distinguish these cases?

The same wrapping technique works for `d/dx` as well:

```
(define d/dx
  (guard (function (function (immediate number?) (immediate number?))
                   (function (immediate number?) (immediate number?)))
         (lambda (f)
           (lambda (x)
             (/ (- (f (+ x 0.001))
                   (f x))
                0.001)))))
```

Exercise

There are seven ways to violate this contract, corresponding to each of the seven contract constructors. Violate each of them by passing arguments or modifying code, as needed. Can you improve error reporting to correctly identify each kind of violation?

Notice that the nested function contract defers the checking of the immediate contracts for two applications, rather than one. This is what we should expect, because immediate contracts only report problems with actual values, so they cannot report anything until applied to actual values. However, this does mean that the notion of "violation"' is subtle: the function value passed to `d/dx` may in fact truly be in violation of the contract, but this violation will not be *observed* until numeric values are passed or returned.

16.4 Syntactic Convenience

Earlier we saw two styles of using flat contracts, as embodied in `real-sqrt-1` and `real-sqrt-2`. Both styles have disadvantages. The latter, which is reminiscent of traditional assertion systems, simply does not work for higher-order values, because it is the wrapped value that must be used in the computation. (Not surprisingly, traditional assertion systems only handle immediate contracts, so they fail to notice this subtlety.) The style in the former, where we wrap each use with a contract, works in theory but suffers from three downsides:

1. The developer may forget to wrap some uses.
2. The contract is checked once per use, which is wasteful when there is more than one use.
3. The program comingles contract checking with its functional behavior, reducing readability.

Fortunately, a judicious use of syntactic sugar can solve this problem in common cases. For instance, suppose we want to make it easy to attach contracts to function parameters, so a developer could write

```
(define/contract (real-sqrt (x :: (immediate positive?)))
  (sqrt x))
```

with the intent of guarding `x` with `positive?`, but performing the check only once, on function invocation. This should translate to, say,

```
(define (real-sqrt new-x)
  (let ([x (guard (immediate positive?) new-x)])
    (sqrt x)))
```

That is, the macro generates a fresh name for each identifier, then associates the name given by the user to the wrapped version of the value supplied to that fresh name. The following macro implements exactly this:

```
(define-syntax (define/contract stx)
  (syntax-case stx (::)
    [(_ (f (id :: c) ...) b)
     (with-syntax ([(new-id ...) (generate-temporaries #'(id ...))])
       #'(define f
           (lambda (new-id ...)
             (let ([id (guard c new-id)]
                   ...)
               b))))]))
```

With conveniences like this, designers of contract languages can improve the readability, efficiency, and robustness of contract use.

16.5 Extending to Compound Data Structures

As we have already discussed, it appears easy to extend contracts to structured datatypes such as lists, vectors, and user-defined recursive datatypes. This only requires that the appropriate set of run-time observations be available. This will usually be the case, up to the resolution of types in the language. For instance, as we have discussed [REF], a language with datatypes does not require *type* predicates but will still offer predicates to distinguish the *variants*; this is case where type-level "contract" checking is best (and perhaps must) be left to the static type system, while the contacts assert more refined structural properties.

However, this strategy can run into significant performance problems. For instance, suppose we built a balanced binary search-tree to perform asymptotic logarithmic time (in the size of the tree) insertions and lookups. Now say we have wrapped this tree in a suitable contract. Sadly, the mere act of checking the contract visits the entire tree, thereby taking linear time! Ideally, therefore, we would prefer a strategy whereby the contract was already checked—incrementally—at the time of construction, and does not need to be checked again at the time of lookup.

Worse, both balancing and search-tree ordering are recursive properties. In principle, therefore, they attach to every sub-tree, and so should be applied on every recursive call. During insertion, which is a recursive procedure, the contract would be checked on every visited sub-tree. In a tree of size t, the contract predicate applies to a sub-tree of $\frac{t}{2}$ elements, then to a sub-sub-tree of $\frac{t}{4}$ elements, and so on, resulting—in the worst case—in visiting a total of $\frac{t}{2} + \frac{t}{4} + \cdots + \frac{t}{t}$ elements...making our intended logarithmic-time insertion process take linear time.

In both cases, there is ready mitigation available in many cases. Each value needs to be associated (either intrinsically, or by storage in a hash table) with the set of contracts it has already passed. Then, when a contract is ready to apply, it first checks whether the value has already been checked and, if it has, does not check again. This is essentially a form of memoization of contract checking and can thus reduce the algorithmic complexity of checking. Again, like memoization, this works best when the values are immutable. If the values can mutate and the contracts perform arbitrary computations, it may not be sound to perform this optimization.

There is a subtler way in which we might examine the issue of data structures. As an example, consider the contract we wrote earlier to check that all values in a numeric list are even. Suppose we have wrapped a list in this contract, but are interested only in the first element of the list. Naturally, we are paying the cost of checking all the values in the list, which may take a very long time. More importantly, however, a user might argue that reporting a violation about the second element of the list is itself a violation of our expectation about contract-checking, since we did not actually use that element.

This suggests deferring checking even for some values that could be checked immediately. For instance, the entire list could be turned into a wrapped value containing a deferred check, and each value is checked only when it is visited. This strategy might be attractive, but it is not trivial to code, and especially runs into problems in the presence of *aliasing*: if two different identifiers are referring to the same list, one with a contract guard and the other without, we have to ensure both of them function as expected (which usually means we cannot store any mutable state in the list itself).

16.6 More on Contracts and Observations

A general problem for any contract implementation—which is exacerbated by complex data—is a curious one. Earlier, we complained that it was difficult to check function contracts because we have insufficient power to observe: all we can check is that a value is a function, and no more. In real languages, the problem for data structures is actually the opposite: we have too much ability to observe. For instance, if we implement a strategy of deferring checking of a list, we quite possibly need to use a structure to hold the actual list, and modify `first` and `rest` to get their values through this structure (after checking contracts). However, a procedure like `list?` might now return `false` rather than `true` because structures are not lists; therefore, `list?` needs to be re-bound to a procedure that also returns `true` on structures that represent these special deferred-contract lists. But the contract system author needs to also remember to tackle `cons?`, `pair?`, and goodness knows how many other procedures that all perform observations.

In general, one observation is essentially impossible to "fix": `eq?`. Normally, we have the property that every value is `eq?` to itself, even for functions. However, the wrapped value of a function is a new procedure that not only *isn't* `eq?` to itself but probably *shouldn't* be, because its behavior truly is different (though only on contract violations, and only after enough values have been supplied to observe the violation). However, this means that a program cannot surreptitiously guard itself, because the act of guarding can be observed. As a result, a malicious module can sometimes detect whether it is being passed guarded values, behaving normally when it is and abnormally only when it is not!

16.7 Contracts and Mutation

We should rightly be concerned about the interaction between contracts and mutation, and even more so when we have contracts that are either inherently deferred or have been implemented in a deferred fashion. There are two things to be concerned about. One is storing a contracted value in mutable state. The other is writing a contract *for* mutable state.

When we store a contracted value, the strategy of wrapping ensures that contract checking works gracefully. At each stage, a contract checks as much as it can with the value at hand, and creates a wrapped value embodying the residual check. Thus, even if this wrapped value is stored in mutable state and retrieved for use later, it still contains these checks, and they will be performed when the value is eventually used.

The other issue is writing contracts for mutable data, such as boxes and vectors. In this case we probably have to create a wrapper for the entire datatype that records the intended contract. Then, when a value inside the datatype is replaced with a new one, the operation that performs the update—such as `set-box!`—needs to retrieve the intended contract from the wrapper, apply it to the value, and store the wrapped value. Therefore, this requires changing the behavior of the data structure mutation operators to be sensitive to contracted values. However, mutation does not change the point at which violations are caught: right away for immediate contracts, upon (in)appropriate use for deferred ones.

16.8 Combining Contracts

Now that we've discussed combinators for all the basic datatypes, it's natural to discuss combining contracts. Just as we saw unions [REF] and intersections [REF] for types, we should be considering unions and intersections (respectively, "or"s and "and"s), ; for that matter, we might also consider negation. However, contracts are only superficially like types, so we have to consider these questions in their own light for contracts rather than try to map the meanings we have learned from types to the sphere of contracts.

As always, the immediate case is straightforward. Union contracts combine with disjunction—indeed, being predicates, their results can literally be combined with `or`—and intersection contracts with conjunction. We apply the predicates in turn, with short-circuiting, and either generate an error or return the contracted value. Intersection contracts combine with conjunction (`and`). And negation contracts are simply the original immediate contract applied and the decision negated (with `not`).

Contract combination is much harder in the deferred, higher-order case. For instance, consider the negation of a function contract from numbers to numbers. What exactly does it mean to negate it? Does it mean the function should *not* accept numbers? Or that if it does, it should not produce them? Or both? And in particular, how do we enforce such a contract? How, for instance, do we check that a function does not accept numbers—are we expecting that when given a number, it produces an error? But now consider the identity function wrapped with such a contract; since it clearly does not result in an error when given a number (or indeed any other value), does that mean we should wait until it produces a value, and if it does produce a number, reject it? But worst of all, note that this means we will be running functions on domains on which they are *not* defined: a sure recipe for destroying program invariants, polluting the heap, or crashing the program.

Intersection contracts require values to pass all the sub-contracts. This means re-wrapping the higher-order value in something that checks all the domain sub-contracts as well as all the range sub-contracts. Failing to meet even one sub-contract means the value has failed the entire intersction.

Union contracts are more subtle, because failing to meet any one sub-contract is not grounds for rejection. Rather, it simply means that that one sub-contract is no longer a candidate contract representing the wrapped value; the other sub-contracts might still be candidates, and only when no others are left must be reject the value. This means the implementation of union contracts must maintain memory of which sub-contracts have and have not yet passed—memory, in this case, being a sophisticated term for the use of mutation. As each sub-contract fails, it is removed from the list of candidates, while all the remaining ones continue to applied. When no candidates remain, the contract system must report a violation. The error report would presumably provide the actual values that eliminated each part of each sub-contract (keeping in mind that these may be nested multiple functions deep).

In a multi-threaded language like Racket, this also requires locks to avoid race conditions.

The implemented versions of contract constructors and combinators in Racket place restrictions on the acceptable forms of sub-contracts. These enable implementations that are both efficient and yield useful error messages. Furthermore, the more extreme situations discussed above rarely occur in practice—though now you know how to

implement them if you need to.

16.9 Blame

Let's now return to the issue of reporting contract violations. By this I don't mean what string to print, but the much more important question of *what* to report, which as we are about to see is really a semantic consideration.

To illustrate the problem recall our definition of `d/dx` above, and assume we were running it without any contract checking. Suppose now that we apply this function to the entirely inappropriate `string-append` (which neither consumes nor produces numbers). This simply produces a value:

```
> (define d/dx-sa (d/dx string-append))
```

(Observe that this would succeed even if contract checking were on, because the immediate portion of the function contract recognizes `string-append` to be a function.) Now suppose we apply `d/dx-sa` to a number, as we ought to be able to do:

```
> (d/dx-sa 10)
string-append: contract violation
  expected: string?
  given: 10.001
```

Notice that the error report is deep inside the body of `d/dx`. On the one hand, this is entirely legitimate: that is where the improper application of `string-append` occurred. However, the *fault* is not that of `d/dx` at all—rather, it is the fault of whatever body of code supplied `string-append` as a purportedly legitimate function from numbers to numbers. Except, however, the code that did so has long since fled the scene; it is no longer on the stack, and is hence outside the ambit of traditional error-reporting mechanisms.

This problem is not a peculiarity of `d/dx`; in fact, it routinely occurs in large systems. This is because systems, especially with graphical, network, and other external interfaces, make heavy use of *callbacks*: functions (or methods) that register interest in some entity and are invoked to signal some status or value. (Here, `d/dx` is the moral equivalent of the graphics layer, and `string-append` is the callback that has been supplied to (and stored by) it.) Eventually, the system layer invokes the callback. If this results in an error, it is the fault of *neither* the system layer—which was given a callback of purportedly the right contract—*nor* of the callback itself, which presumably has legitimate uses but was improperly supplied to the function. Rather, *the fault is of the entity that introduced these two entities*. However, at this point the call stack contains only the callback (on top) and the system (below it)—and the only guilty party is no longer present. These kinds of errors can therefore be extremely difficult to debug.

The solution is to extend the contract system to incorporate a notion of *blame*. The idea is to effectively record the introduction that resulted in a pair of components coming together, so that if a contract violation occurs between them, we can ascribe the failure to the expression that did the introduction. Observe that this is only really interesting in the context of functions, but for consistency we will extend blame to immediate contracts as well in a natural way.

For a function, notice that there are two possible points of failure: either it was *given* the wrong kind of value (the pre-condition), or it *produced* the wrong kind of

value (the post-condition). It is important to distinguish these two cases because in the former case we should blame the environment—in particular, the actual parameter expression—whereas in the latter case (assuming the parameter has passed muster) we should blame the function itself. (The natural extension to immediate values is that we can only blame the value itself for not satisfying the contract, which is akin to the "post-condition".)

For contracts, we will introduce the terms *positive* and *negative* position. For a first-order function, the negative position is the pre-condition and the positive one the post-condition. Therefore, this might appear to be needless extra terminology. As we will soon see, however, these terms have a more general meaning.

We will now generalize the parameters consumed by contracts. Previously, immediate contracts consumed a predicate and function contracts consumed domain and range contracts. This will still be the case. However, what they each return will be a function of two arguments: labels for the positive and negative positions. (These labels can be drawn from any reasonable datatype: abstract syntax nodes, buffer offsets, or other descriptions. For simplicity, we will use strings.) Thus function contracts will close over the labels of these program positions, to later blame the provider of an invalid function.

The `guard` function is now responsible for passing through the labels of the contract application locations:

```
(define (guard ctc val pos neg) ((ctc pos neg) val))
```

and let us also have `blame` display the appropriate label (which we will pass to it from the contract implementations):

```
(define (blame s) (error 'contract s))
```

Suppose we are guarding the use of `add1`, as before. What are useful names for the positive and negative positions? The positive position is post-condition: i.e., any failure here must be blamed on the body of `add1`. The negative position is the pre-condition: i.e., any failure here must be blamed on the parameter to `add1`. Thus:

```
(define a1 (guard (function (immediate number?)
                            (immediate number?))
                  add1
                  "add1 body"
                  "add1 input"))
```

Had we provided a non-function to guard, we would expect an error at the "post-condition" location: this is not really a failure of the post-condition, but surely the parameter cannot be blamed if the application failed to be a function. (However, this shows that we are really stretching the term "post-condition", and the terms "positive" provides a useful alternative.) Because we trust the implementation of `add1` to only produce numbers, we would expect it is impossible to fail the post-condition. However, we would expect an expression like `(a1 "x")` to trigger a pre-condition error, presumably signaling a contract error at the location `"add1 input"`. In contrast, had we guarded a function that violates the post-condition, such as this,

```
(define bad-a1 (guard (function (immediate number?)
                                (immediate number?))
                      number->string
                      "bad-add1 body"
                      "bad-add1 input"))
```

we would expect blame to be ascribed to `"bad-add1 body"`.

Let us now see how to implement these contract constructors. For immediate contracts, we have seen that blame should be ascribed to the positive position:

```
(define (immediate pred?)
  (lambda (pos neg)
    (lambda (val)
      (if (pred? val) val (blame pos)))))
```

For functions, we might be tempted to write

```
(define (function dom rng)
  (lambda (pos neg)
    (lambda (val)
      (if (procedure? val)
          (lambda (x) (dom (val (rng x))))
          (blame pos)))))
```

but this fails to work in a very fundamental way: it violates the expected signature on contracts. That is because all contracts now expect to be given the labels of positive and negative positions, which means `dom` and `rng` cannot be used as above. (As another hint, we are using `pos` but not `neg` anywhere in the body, even though we have seen examples where we expect the position bound to `neg` to be blamed.) Instead, clearly, we somehow instantiate the domain and range contracts using `pos` and `neg`, so that they "know" and "remember" where a potentially violating function was applied.

The most obvious reaction would be to instantiate these contract constructors with the same values of `dom` and `rng`:

```
(define (function dom rng)
  (lambda (pos neg)
    (let ([dom-c (dom pos neg)]
          [rng-c (rng pos neg)])
      (lambda (val)
        (if (procedure? val)
            (lambda (x) (rng-c (val (dom-c x))))
            (blame pos))))))
```

Now all the signatures match up, and we can run our contracts. But when we do so, the answers are a little strange. For instance, on our simplest contract violation example, we get

```
> (a1 "x")
contract: add1 body
```

Huh? Maybe we should expand out the code of `a1` to see what happened.

```
   (a1 "x")
 = (guard (function (immediate number?)
                    (immediate number?))
          add1
          "add1 body"
          "add1 input")
 = (((function (immediate number?) (immediate number?))
     "add1 body" "add1 input")
    add1)
 = (let ([dom-c ((immediate number?) "add1 body" "add1 input")]
         [rng-c ((immediate number?) "add1 body" "add1 input")])
     (lambda (x) (rng-c (add1 (dom-c x)))))
 = (let ([dom-c (lambda (val)
                  (if (number? val) val (blame "add1 body")))]
         [rng-c (lambda (val)
                  (if (number? val) val (blame "add1 body")))])
     (lambda (x) (rng-c (add1 (dom-c x)))))
```

Poor `add1`: it never stood a chance! The only blame label left is `"add1 body"`, so it was the only thing that could ever be blamed.

We will return to this problem in a moment, but observe how in the above code, there are no real traces of the function contract left. All we have are immediate contracts, ready to blame actual values if and when they occur. This is perfectly consistent with what we said earlier [REF] about being able to observe only immediate values. Of course, this is only true for first-order functions; when we get to higher-order functions, this will no longer be true.

What went wrong? Notice that only the contract bound to `rng-c` ought to be blaming the body of `add1`. In contrast, the contract bound to `dom-c` ought to be blaming the input to `add1`. It's almost as if, in the domain position of a function contract, the positive and negative labels should be...swapped.

If we consider the contract-guarded `d/dx`, we see that this is indeed the case. The key insight is that, when applying a function taken as a parameter, the "outside" becomes the "inside" and vice versa. That is, the body of `d/dx`—which was in positive position—is now the caller of the function to differentiate, putting that function's body in positive position and the caller—the body of `d/dx`—in negative position. Thus, on the domain side of the contract, every nested function contract causes positive and negative positions to swap.

On the range side, there is no need to swap. Consider again `d/dx`. The function it returns represents the derivative, so it should be given a number (representing the point at which to calculate the derivative) and it should return a number (the derivative at that point). The negative position of this function is indeed the client who uses the derivative function—the pre-condition—and the positive position is indeed the body of `d/dx` itself—the post-condition—since it is responsible for generating the derivative.

As a result, we obtain an updated, and correct, definition for the function constructor:

```
(define (function dom rng)
  (lambda (pos neg)
    (let ([dom-c (dom neg pos)]
          [rng-c (rng pos neg)])
      (lambda (val)
        (if (procedure? val)
            (lambda (x) (rng-c (val (dom-c x))))
            (blame pos))))))
```

Exercise

> Apply this to our earlier example and confirm that we get the expected blame. Also expand the code manually to see why this happens.

Suppose, further, we define `d/dx` with the labels `"d/dx body"` for its positive position and `"d/dx input"` for its negative. Say we supply the function `number->string`, which patently does not compute derivatives, and apply the result to `10`:

```
((d/dx (guard (function (immediate number?)
                        (immediate string?))
               number->string
               "n->s body"
               "n->s input"))
 10)
```

This correctly indicates that the blame should be ascribed to the expression that fed `number->string` as a supposed numeric function to `d/dx`—not to `d/dx` itself.

Exercise

> Hand-evaluate `d/dx`, apply it to *all* the relevant violation examples, and confirm that the resulting blame is accurate. What happens if you supply `d/dx` with `string->number` with a function contract indicating it maps strings to numbers? What if you supply the same function with no contract at all?

17 Alternate Application Semantics

Long ago [REF], we considered the question of what to substitute when performing application. Now we are ready to consider some alternatives. At the time, we suggested just one alternative; in fact there are many more. To understand this, see whether you can answer this question:

Which of these is the same?

- `(f x (current-seconds))`
- `(f x (current-seconds))`
- `(f x (current-seconds))`

- `(f x (current-seconds))`

What we're about to find is that this fragment of syntax can have wildly different run-time behaviors. For instance, there is the distinction we have already mentioned: variation in when `(current-seconds)` is evaluated. There is variation in *how many times* it is evaluated (and hence `f` is run). There is even variation even in whether values for `x` flow strictly from the caller to the callee, or can even flow in the opposite direction!

17.1 Lazy Application

Let's start by considering when parameters are reduced to values. That is, do we substitute formal parameters with the *value* of the actual parameter, or with the actual parameter *expression* itself? If we define

```
(define (sq x) (* x x))
```

and invoke it as

```
(sq (+ 2 3))
```

does that reduce to

```
(* 5 5)
```

or to

```
(* (+ 2 3) (+ 2 3))
```

? The former is called *eager* application, while the latter is *lazy*. Of course we don't want to return to defining interpreters by substitution, but it is always useful to think of substitution as a design principle.

Some people also use the term *strict* for the former. A more arcane terminology is *applicative-order evaluation* for the former and *normal-order evaluation* for the latter. Or, *call-by-value* for the former and *call-by-name* or *call-by-need* for the latter. The last two terms—by-name versus by-need—actually represent a technical distinction we will see below. This concludes our name-dump.

17.1.1 A Lazy Application Example

The lazy alternative has a distinguished history (for instance, this is what the true λ-calculus uses), but returned to the fore from programming experiments considering what might happen if certain operators did not evaluate the arguments at application but only when the value was needed. For instance, consider the definition

```
(define ones (cons 1 ones))
```

In ordinary Racket, this is clearly ill-defined: `ones` has not yet been defined (on the left) when we try to evaluate it (on the right), so this results in an error. If, however, we do not try to evaluate it until we actually need it, by that time the definition is well-formed. Because each `rest` obtains another `ones`, this produces an infinite list.

We've glossed over a lot that needs explaining. Does the `ones` in the `rest` position of the `cons` evaluate to a *copy* of that expression, or to the result of the very same

expression itself? In other words, have we simply created an infinitely unfolding list, or have we created an actually *cyclic* one?

This depends in good part on whether or not our language has mutation. If it does, then perhaps we can modify each of the cells of the resulting list, which means we can observe the difference between the two implementations above: in the unfolded version mutating one `first` will not affect another, but in the cyclic one, changing one will affect them all. Therefore, in a language with mutation, we might argue that this should represent a lazy unfolding, but not an actual cyclic datum.

Keep this discussion in mind. We cannot resolve it right now; rather, let us examine lazy evaluation a little more, then return to this question [REF].

17.1.2 What Are Values?

If we return to our core higher-order function interpreter [REF], we recall that we have two kinds of values: numbers and closures. If we want to support lazy evaluation instead, we need to ask what happens at function application. What exactly are we passing?

This seems obvious enough: in a lazy application semantics, we need to pass *expressions*. But a moment's thought shows that this can be problematic. Expressions contain identifier names, and we don't want them to be accidentally bound.

And now these truly will be *identifiers*, not *variables*, as we will see [REF].

For instance, suppose we have

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

and apply it as follows:

```
((f 3) (+ x 4))
```

Do Now!

What should this produce?

Clearly, we should get an error reporting `x` as not being bound.

Now let's trace it. The first application creates a closure where `x` is bound to `3`. If we now bind `y` to `(+ x 4)`, this results in the expression `(+ x (+ x 4))` in an environment where `x` is bound. As a result we get the answer `10`, not an error.

Do Now!

Have we made a subtle assumption above?

Yes we have: we've assumed that `+` evaluates arguments and returns numeric answers. Perhaps `+` also behaves lazily; we will study this issue in a moment. Nevertheless, the central point remains: if we are not careful, this erroneous expression will produce some kind of valid answer, not an error.

In case you think this is entirely a problem with erroneous programs, and can hence be treated specially (e.g., first scan the program source for free identifiers), here is another use of the same `f`:

```
(let ([x 5])
  ((f 3) x))
```

Do Now!

What should this produce?

We would expect this to produce the result of `(+ 3 5)` (probably `8`). However, if we substitute `x` inside the arithmetic expression, we would get `(+ 3 3)` instead.

This latter example holds the key to our solution. In the latter example, the problem ostensibly arises only when we use environments; if instead we use substitution, `x` in the application is substituted as soon as we encounter the `let`, and the result is what we expect. In fact, note that the same argument holds earlier: if we had used substitution, the very occurrence of `x` would have signaled an error. In short, we have to make sure our environment-based implementation matches what substitution would have done. Doesn't that sound familiar!

In other words, the solution is to bundle the argument expression *with its environment*: i.e., create a closure. This closure has no parameters, so it is effectively a *thunk*. We could use existing functions to represent these thunks, but our instinct should tell us that it is better to use different data representations for logically different purposes: `closV` for user-created closures, and something else for internally-created ones. Indeed, as we will see, it will have been wise to keep them separate because there is one place where it is critical we can tell them apart.

Indeed, this demonstrates that functions have two uses: to substitute names with values, and also to defer substitution. `let` is the former without the latter; thunks are the latter without the former. We have already established that the former is valuable in its own right; this section shows that the same is true of the latter.

To conclude this discussion, here is our new set of values:

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

The first two variants are exactly the same; the third is new, and as we discussed, is effectively a parameter-less procedure, as its type suggests.

17.1.3 What Causes Evaluation?

Let us now return to discussing arithmetic expressions. On evaluating `(+ 1 2)`, a lazy application interpreter could return any number of things, including `(suspendV (+ 1 2) mt-env)`. In this way suspended computation could cascade on suspended computation, and in the limiting case every program would return immediately with an "answer": the thunk representing the suspension of its computation.

It is legitimate to write `mt-env` here because even if the `(+ 1 2)` expression was written in a non-empty environment, it has no free identifiers, so it doesn't need any of the environment's bindings.

Clearly, *something* must force a suspension to be lifted. (Lifting a suspension means, of course, evaluating its body in the stored environment.) Those expression positions that undo suspensions are called *strictness points*. The most obvious strictness point is the interactive environment's printer, because a user clearly would not use such an environment if they did not wish to see answers. We will embody the act of lifting suspension in the procedure `strict`:

```
(define (strict [v : Value]) : Value
  (type-case Value v
    [numV (n) v]
    [closV (a b e) v]
    [suspendV (b e) (strict (interp b e))]))
```

where the returned `Value` is guaranteed to not be a `suspendV`. We can imagine the printer as wrapping `strict` around the result of evaluating the program, to obtain a value to print.

Do Now!

> What impact would using closures to represent suspended computation have had?

The definition of `strict` above depends crucially on being able to distinguish deferred computations—which are internally-constructed closures—from user-defined closures. Had we conflated the two, then we would have to guess what to do with zero-argument closures. If we fail to further process them, we might incorrectly get an error (e.g., + might get a thunk rather than the numeric value residing inside it). If we do process it further, we might accidentally force a user-defined thunk prematurely. In short, we need a flag on thunks telling us whether they are internal or user-defined. For clarity, our interpreter uses a separate variant.

Let us now return to the interaction between `strict` and the interpreter. Unfortunately, as we have defined things, this will cause an infinite loop. The act of trying to interpret an addition creates a suspension, which `strict` tries to undo by forcing the interpreter to interpret an addition, which.... Clearly, therefore, we cannot have every expression simply suspend its computation; instead, we will limit suspension to applications. This suffices to give us the rich power of laziness, without making the language absurd.

17.1.4 An Interpreter

As usual, we will define the interpreter in cases.

<***lazy-interp***> **::=**

```
(define (interp [expr : ExprC] [env : Env]) : Value
  (type-case ExprC expr
    <lazy-numC-case>
    <lazy-idC-case>
    <lazy-plusC/multC-case>
    <lazy-appC-case>
    <lazy-lamC-case>))
```

Numbers are easy: they are already values, so there is no point needlessly suspending them:

<***lazy-numC-case***> **::=**

```
[numC (n) (numV n)]
```

Closures, similarly, remain the same:
<lazy-lamC-case> **::=**

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

Identifiers should just return whatever they are bound to:
<lazy-idC-case> **::=**

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

The arguments of arithmetic expressions are usually defined as strictness points, because otherwise we would simply have to implement the actual arithmetic elsewhere:
<lazy-plusC/multC-case> **::=**

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

Finally, we have application. Here, instead of evaluating the argument position, we suspend it. The function position has to be a strictness point, however, otherwise we wouldn't know what function to apply and hence how to continue the computation:
<lazy-appC-case> **::=**

```
[appC (f a) (local ([define f-value (strict (interp f env))])
              (interp (closV-body f-value)
                      (extend-env (bind (closV-arg f-value)
                                        (suspendV a env))
                                  (closV-env f-value))))]
```

And that's it! By adding a new kind of answer, inserting a few calls to `strict`, and replacing `interp` with `suspendV` in the argument position of application, we have turned our eager application interpreter into one with lazy application. Yet this small change has such enormous impact on the programs we write! For a more thorough examination of this impact, study Haskell or the `#lang lazy` language in Racket.

Exercise

If we instead replace the identifier case with `(strict (lookup n env))` (i.e., wrapped `strict` around the result of looking up an identifier), what impact would it have on the language? Consider richer languages with data structures, etc.

Exercise

Construct programs that produce different results in a lazy evaluation than an eager evaluation (i.e., the same program text with different answers in the two cases). Try to make the differences interesting, i.e., beyond whether one returns a `suspendV` while the other doesn't. For instance, does one terminate or produce an error while the other one doesn't?

Exercise

Instrument both interpreters to count the number of steps they take to return answers. For programs that produce the same answer under both evaluation strategies, does one strategy always take more steps than the other?

17.1.5 Laziness and Mutation

One of the virtues of lazy evaluation is that it defers execution. Usually this is a good thing: it enables us to build infinite data structures and avoids computation until necessary. Unfortunately, it also changes when computations occur, and in particular, changes the order of when computations evaluate relative to each other, depending on what strictness points are encountered when. As a result, programmers greatly lose predictability of ordering. This is of course a problem when expressions perform mutation operations, because now it becomes extremely difficult to predict what value a program will compute (relative to the eager version).

As a result, the core of every lazy language is free of mutation. In Haskell, mutation and other state operations are introduced through a variety of mechanisms such as *monads* and *arrows* that ultimately introduce the ability to (strictly) sequentialize code; this sequentiality is essential to being able to predict the order of execution and thus the result of operations. If programs are structured well the number of these dependencies should be small; furthermore, the Haskell type system attempts to reflect these operations in the types themselves, so programmers can more easily reason about their effects.

17.1.6 Caching Computation

Now that we've concluded that lazy computation has to have no mutations, we observe a pleasant consequence (dare we say, side-effect?): given a fixed environment, an expression always produces the same answer. As a result, the run-time system can cache the value of an expression when it is first forced to an answer by strictness, and return this cached value on subsequent attempts to compute it. Of course, this caching—which is a form of *memoization*—is only sound when the expression returns the same value every time, which we have assumed. In fact, the compiler and run-time system can aggressively hunt for uses of the same expression in different parts of the program and, if the relevant parts of their environment are the same, conflate their evaluation. The strategy of evaluating the suspended computation every time it is needed is called *call-by-name*; that of caching its result, *call-by-need*.

17.2 Reactive Application

Now consider an expression like `(current-seconds)`. When we evaluate it, it returns a single number representing the current time. For instance,

```
> (current-seconds)
1353030630
```

However, even as we stare at this value, it is already out-of-date! It represents the time when the function application occurred, but does not stay current.

17.2.1 Motivating Example: A Timer

Suppose we were trying to implement a timer that measures elapsed time. Ideally, we would like to write a program such as this:

```
(let ([start (current-seconds)])
  (- (current-seconds)
     start))
```

In JavaScript, we might write:

```
d = new Date();
start = d.getTime();
current = d.getTime();
elapsed = current - start;
```

On most machines this Racket expression, or the value of `elapsed` in JavaScript, will evaluate to `0` or some other very small number. This is because these programs represent *one* measure of the elapsed time: that at the second invocation of the procedure that gets the current time. This gives us an instanteous time split, but not an actual timer.

In most languages, to build an actual timer, we would have to create an instance of some sort of timer object, and install a callback. Every time the clock ticks, the timer object—representing the operating system—invokes the callback. The callback is then responsible for updating values in the rest of the system, and hopefully doing so globally and consistently. However, it cannot do so by returning values, because it would return to the operating system, which is agnostic to and does not care about our application; therefore, the callback is forced to perform its action through mutation. In JavaScript, for instance:

```
var timerID = null;
var elapsedTime = 0;

function doEverySecond() {
  elapsedTime += 1;
  document.getElementById('curTime').innerHTML = elapsedTime; }
function startTimer() {
  timerId = setInterval(doEverySecond, 1000); }
```

assuming we have an HTML page with an id named `curTime`, and that the `onload` or other callback invokes `startTimer`.

One alternative to this spaghetti code is for the application program to repeatedly poll the operating system for the current time. However:

- Calling too frequently wastes resources, while calling too infrequently results in incorrect answers. However, to call at just the right resolution, we would need a timer signal in the first place!
- While it may be possible to create such a polling loop for regular events such as timers, it is impossible to do so accurately for unpredictable behaviors such as user input (whose frequency cannot, in general, be predicted).

- On top of all this, writing this loop pollutes the program's structure and forces the developer to sustain this extra burden.

The callback-based solution, however, demonstrates an *inversion of control*. Instead of the application program calling the operating system, the operating system has now been charged with calling (into) the application program. The reactive behavior that should have been deeply nested, inside the display expression, has instead been brought to the top-level, and its value drives the other computations. The fundamental cause for this is that the world is in control, not the program, so external stimuli determine when and how the program should next run, not intrinsic program expressions.

17.2.2 Callback Types are Four-Letter Words

The characteristic signature (so to speak) of this pattern is manifest in the types. Because the operating system is agnostic to the program's values, the callback usually has no return type at all, or it is a generic status indicator, not an application-specific value. Therefore, in typed languages, the type is usually some *four-letter word*. For instance, here is a fragment of a GUI library in Java:

```
interface ChangeListener extends EventListener {
  void stateChanged(ChangeEvent e) { ... } }

interface ActionListener extends EventListener {
  void actionPerformed(ActionEvent e) { ... } }

interface MouseListener extends EventListener {
  void mouseClicked(MouseEvent e) { ... }
  void mouseEntered(MouseEvent e) { ... } }
```

And here's one in OCaml:

```
mainLoop : unit -> unit
closeTk : unit -> unit

destroy : 'a Widget.widget -> unit
update : unit -> unit

pack : ... -> 'd Widget.widget list -> unit
grid : ... -> 'b Widget.widget list -> unit
```

In Haskell, the four letters have an extra space in them:

```
select :: Selecting w => Event w (IO ())
mouse :: Reactive w => Event w (EventMouse -> IO ())
keyboard :: Reactive w => Event w (EventKey -> IO ())
resize :: Reactive w => Event w (IO ())
focus :: Reactive w => Event w (Bool -> IO ())
activate :: Reactive w => Event w (Bool -> IO ())
```

and so on. In all these cases, the presence of a "void"-like type clearly indicates that the functions do not return any interesting value, so their only purpose must be to mutate the store or have some other side-effect. This also means that no rich means of

composition—such as the nesting of expressions—is possible: the only composition operator for void-typed statements is sequencing. Thus the types reveal that we will be forced away from being able to write nested expressions.

Readers will, of course, be familiar with this problem from our earlier discussion of Web programming. This problem occurs on the server due to statelessness [REF], and also on the client due to single-threading [REF]. On the server, at least, we were able to use continuations to address this problem. However, continuations are not available in all languages, and implementing them can be onerous. Furthermore, it can be tricky to set up just the right continuation to pass as a callback. Instead, we will explore an alternate solution.

17.2.3 The Alternative: Reactive Languages

Consider the FrTime (pronounced "Father Time") language in DrRacket. If we run this expression at the interactions window, we still get `0` or some other very small non-negative number:

In DrRacket v5.3, you must select the language from the Language menu; writing `#lang frtime` will not provide the interesting interactions window behavior.

```
(let ([start (current-seconds)])
  (- (current-seconds)
     start))
```

In fact, we can try several other expressions and see that FrTime seems to have exactly like traditional Racket.

However, it also binds a few additional identifiers. For instance, it provides a value bound to `seconds`. If we type this into the interaction prompt, we get something very interesting! First we see `1353030630`, then a second later `1353030631`, another second later `1353030632`, and so on. This kind of value is called a *behavior*: a value that changes over time. Except we haven't written any callbacks or other code to keep it current.

A behavior can be used in computations. For instance, we can write `(- seconds seconds)`, and this always evaluates to `0`. Here are some more expressions to try at the interaction prompt:

```
(add1 seconds)
(modulo seconds 10)
(build-list (modulo seconds 10) identity)
(build-list (add1 (modulo seconds 10)) identity)
```

As you can see, being a behavior is "sticky": if any sub-expression is a behavior, so is its enclosing expression.

Thanks to this evaluation model, every time `seconds` updates the entire application happens afresh: as a result, even though we have written seemingly simple expressions without any explicit loop-like control, the program still "loops". In other words, having explored an application semantics where arguments are evaluated once, then another where they may be evaluated zero times, now we have one where they are evaluated as many times as necessary, and the entire corresponding function with them. As a consequence, reactive values that are "inside" an expression no longer need to be brought

"outside"; rather, they can reside nested inside expressions, giving programmers a more natural means of expression. This style of evaluation is called *dataflow* or *functional reactive* programming.

Historically, *dataflow* has tended to refer to languages with first-order functions, whereas *functional reactive* languages support higher-order functions too. See the Flapjax Web site.

FrTime implements what we call *transparent reactivity*, whereby the programmer can inject a reactive behavior anywhere in a program's evaluation without needing to make any syntactic changes to its context. This has the virtue of making it easy to inject reactivity into existing programs, but it can make the evaluation and cost model more complex for programmers. In other languages, programmers can instead explicitly introduce behavior through appropriate primitives, trading convenience for greater predictability. FrTime's sister language, Flapjax, an extension of JavaScript, provides both modes.

17.2.4 Implementing Transparent Reactivity

To make an existing language implement transparent reactivity, we have to (naturally) alter the semantics of function application. We will do this in two steps. First we will rewrite reactive function applications into a more complex form, then we will show how this more complex form enables reactive updates.

Dataflow Graph Construction

The essence of making an application reactive is simple to explain through desugaring. Assume we have defined a new constructor `behavior`. The constructor takes a thunk that represents what computation to perform every time an argument updates, and all the values that the expression depends on. The value it produces stores the current value of the behavior. Then an expression like `(f x y)` turns into

```
(if (or (behavior? x) (behavior? y))
    (behavior (λ () (f (current-value x) (current-value y))) x y)
    (f x y))
```

where we assume, given a non-behavior constant, `current-value` behaves as the identity function.

Let us look at two examples of using the above definition. Consider the trivial case where neither parameter is a behavior, e.g., `(+ 3 4)`. This desugars to

```
(if (or (behavior? 3) (behavior? 4))
    (behavior (λ () (+ (current-value 3) (current-value 4))) 3 4)
    (+ 3 4))
```

Since both `3` and `4` are numbers, not behaviors, this reduces to `(+ 3 4)`, which is precisely what we would like. This reflects an important principle: when no behaviors are present, programs behave exactly as they did in the non-reactive version of the language.

If we compute `(+ 1 seconds)`, this expands to

```
(if (or (behavior? 1) (behavior? seconds))
    (behavior (λ () (+ (current-value 1) (current-value seconds))) 1 seconds)
    (+ 1 seconds))
```

Because `seconds` is a behavior, this reduces to

```
(behavior (λ () (+ (current-value 1) (current-value seconds))) 1 seconds)
```

Any expression that depends on this now sees its argument also become a behavior, making the property "sticky" as we argued before.

Exercise

In what way, if any, did the above desugaring depend on eager evaluation?

Dataflow Graph Update

Of course, simply constructing behavior values is not enough. The key additional information is in the extra arguments to `behavior`. The language filters out those arguments that are themselves behaviors (e.g., `seconds`, above) and registers this new behavior as one of that depends on those existing ones. This registration process creates a graph of behavior expression dependencies, known as a *dataflow graph* (since it reflects the paths along which data need to flow).

If the program did not evaluate to any behaviors, then evaluation simply produces an answer, and there are no graphs created. If, however, there are behavior dependencies, then evaluation produces not a traditional answer but a behavior value, with dependencies already recorded. (In practice, it is useful to also track which primitive behaviors are actually necessary, to avoid unnecessarily evaluating primitives that no other behavior in the program refers to.) In short, *program execution generates a dataflow graph*. Thus, we do not need a special, new evaluator for the language; we instead embed the graph-construction semantics in traditional evaluation.

Now a dataflow propagation algorithm begins to execute. Every time a primitive behavior changes, the algorithm applies its stored thunk, obtains its new value, stores it, and then signals each behavior dependent on it. For instance, if `seconds` updates, it notifies the `(+ 1 seconds)` expression's behavior. The latter behavior now evaluates its thunk, `(λ () (+ (current-value 1) (current-value seconds)))`. This adds `1` to the newest value of `seconds`, making that the new value of this behavior—just as we would expect.

Evaluation Order

The discussion above presents too simple a view of graph update. Consider the following program:

```
(> (add1 seconds)
   seconds)
```

This program has one primitive behavior, `seconds`, and constructs two more: one for `(add1 seconds)` and one more for the entire expression.

We would expect this expression to always be true. However, when `seconds` updates, depending on the order in which it handles updates, it might update the whole expression before it does `(add1 seconds)`. Suppose the old value of `seconds` was `100`, so the new one is `101`. However, the node for `(add1 seconds)` is still storing its old value (because it has not yet been updated), so it holds `(add1 100)` or `101`. That means the `>` compares `101` with `1`, which is false, making this expression return

a value that should simply never have ensued from its static description. This situation is called a *glitch*.

There is an easy solution to avoiding glitches, which the above example illustrates (and that a theorem can show is sufficient). This is to *topologically sort* the nodes. Then, every node is only processed after those it depends on have updated, so there is no danger of seeing outdated or inconsistent values.

The problem becomes more difficult in the presence of cycles in the graph. In those cases, we need special recursion operators that can take an initial value for the cyclic behavior. This makes it possible to break the cyclic dependency, reducing evaluation to the process that has already been defined.

There is much more to say about the evaluation of dataflow languages, such as the treatment of conditionals and a dual notion to behaviors that is discrete and stream-like. I hope you will read the literature on reactive languages to learn more about these topics.

Exercise

> Earlier we picked on a Haskell library. To be fair, however, the reactive solution we have shown was enunciated in Haskell, whose lazy evaluation makes this form of evaluation relatively easy to support.
>
> Implement reactive evaluation using laziness.

17.3 Backtracking Application

Another reason an application can occur multiple times is because it is part of a *search tree*. The language's application semantics attempts to satisfy a search; if it succeeds it returns information about the success, but if it fails, it retries applications in the hope of succeeding. This, of course, assumes that the program has been written in terms of choices that can be tried until the search succeeds. Thus the central operation in a language with a backtracking application semantics is *disjunction* ("or"). For a variety of reasons, such languages also include *conjunction* ("and"), not least because negation can be problematic so the usual rules of Boolean algebra don't apply.

17.3.1 Searching for Satisfaction

It is easiest to describe the problem of backtracking search in terms of simple binary-valued goals. With boolean variables, finding satisfying assignments even for propositional formulas is computationally challenging from a performance perspective, and extremely important in a variety of real-world problems. We will, however, consider a restricted version of this problem, where we given boolean constants, not variables, so all we need to determine is truth of the formula. This is only mildly interesting, but it helps set up the actually interesting general case.

Look for the numerous uses for "SAT solvers".

Suppose, therefore, that we are given a formula with conjunction, disjunction, and constants representing truth and falsity. Our goal is to determine whether the formula itself evaluates to truth or falsity. We want to minimize computation, so that when we discover an answer—either one—we want to return it as quickly as possible to a context that depends on it. For instance, if we are evaluating a conjunction and discover that a

For this special case we might as well just draw up a truth table, but that won't work in the general case.

term is false, we would like the entire term to be false right away—the familiar notion of *short-circuit* evaluation of conditionals. However, we want this to generalize to the call-stack as well: if a sub-expression discovers a truth or falsehood, and it matters to the enclosing expression, then it should be reported "up the stack" quickly, as well.

In general, then, every computation should be parameterized by two receptacles: one to which report truth of the current term (if it is discovered) and the other to report falsity (if it is discovered). To avoid complications with pending function calls, etc., we will furthermore expect that the values supplied for these two parameters are both *continuations*, so that the value returns to the right context as quickly as possible, instead of being scrutinized by intermediate levels of evaluation that don't care about the result.

These continuations do not currently have any interesting values to communicate: which continuation is invoked supplies all the information there is (one bit, representing truth or falsehood). Because by default continuations expect one argument, we will supply a symbol that indicates what we already know.

The easiest values to see are truth and falsity itself. Recall that all expressions consume two continuations, called *success* and *failure* continuations, and invoke one if they have a definitive value. The value for truth therefore invokes the success continuation and that for falsity the failure one:

```
(define (truth t1 t2) (t1 'yes))
(define (falsity t1 t2) (t2 'no))
```

Let us now examine disjunction. For simplicity, we will assume a two-armed version of it. Like all computations, the disjunction of two backtracking searches must consume a success and a failure continuation.

<***try-or-bt***> **::=**

```
(define (try-or t1 t2)
  (lambda (success failure)
    <try-or-bt-body>))
```

It is simplest, conceptually, to think of setting up two local continuations, call them `pass` and `fail` to supply to the evaluation of `t1`. If `t1` (or, recursively, one of its descendents) succeeds, control will return to the context of the creation of `pass`; on failure, it will return to `fail`.

If it returns to `pass`, we now know that the first sub-expression has already succeeded. But because this is all that matters for the disjunction to hold, we can now return control to the continuation `success`. Thus any invocation of `pass` should immediately trigger `success`.

In contrast, suppose `t1` fails. Then we should try `t2`. Thus `fail` should be defined in a sequence that next tries `t2`, on the expectation that had `t1` succeeded, control would simply not return this way. Now when trying `t2`, there is no need to worry about `pass` and `fail`: after the failure of `t1` the success and failure of the entire disjunction are the same as those of `t2` (a form of tail position), so its success and failure continuations are the same as that of the whole expression. As a result, we obtain:

<***try-or-bt-body***> **::=**

```
(success (let/cc pass
           (begin
             (let/cc fail
               (t1 pass fail))
             (t2 success failure))))
```

Thus if `t1` succeeds, control returns to the context of creating `pass`, i.e., it invokes `success`. If `t1` succeeds control returns to the continuation of creating `fail`, which is the next statement in the sequencing, which is `t2`.

By symmetric reasoning, we get the dual program for `try-and`:

```
(define (try-and t1 t2)
  (lambda (success failure)
    (failure (let/cc fail
               (begin
                 (let/cc pass
                   (t1 pass fail))
                 (t2 success failure))))))
```

To convert these continuation-fed responses to simple, testable values, we can write a convenient little wrapper:

```
(define (run t)
  (let/cc escape
    (t (lambda (v) (escape 'yes))
       (lambda (v) (escape 'no)))))
```

and with it construct test cases, from

```
(test (run (try-or falsity falsity)) 'no)
```

to

```
(test (run (try-or (try-and (try-or falsity truth) (try-or truth falsity))
                   (try-and truth (try-and falsity truth)))) 'yes)
```