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 desguaring-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 desguaring 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:

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

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.

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 desguars 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.

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

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.

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 desguaring 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`. Can we implement this easily?

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 anthropomorphism.

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*.

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.

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.

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

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

`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.

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

Exercise