```
(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 $\sim$-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 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 an 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 brought "outisde";

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.

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.

See the Flapjax Web site.

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 desguaring. 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 it depends on have been, 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.