PL: Lecture #19

Tuesday, November 12th

Implementing Laziness (in plain Racket)

> 👉 ***PLAI §8***

Generally, we know how lazy evaluation works when we use the substitution model. We even know that if we have:

```
{bind {{x y}}
  {bind {{y 2}}
    {+ x y}}}
```

then the result should be an error because we cannot substitute the binding of `x` into the body expression because it will capture the `y` — changing the binding structure. As an indication, the original expression contains a free reference to `y`, which is exactly why we shouldn't substitute it. But what about:

```
{bind {{x {+ 4 5}}}
  {bind {{y {+ x x}}}
    {bind {{z y}}
      {bind {{x 4}}
        z}}}}
```

Evaluating this eagerly returns 18, we therefore expect any other evaluation (eager or lazy, using substitutions or environments) to return 18 too, because any of these options should not change the meaning of numbers, of addition, *or* of the scoping rules. (And we know that no matter what evaluation strategy we choose, if we get to a value (no infinite loop or exception) then it'll always be the same value.) For example, try using lazy evaluation with substitutions:

```
{bind {{x {+ 4 5}}}
  {bind {{y {+ x x}}}
    {bind {{z y}}
      {bind {{x 4}}
        z}}}}
-->
{bind {{y {+ {+ 4 5} {+ 4 5}}}}
  {bind {{z y}}
    {bind {{x 4}}
      z}}}
-->
{bind {{z {+ {+ 4 5} {+ 4 5}}}}
  {bind {{x 4}}
    z}}
-->
{bind {{x 4}}
  {+ {+ 4 5} {+ 4 5}}}
-->
{+ {+ 4 5} {+ 4 5}}
-->
{+ 9 9}
-->
18
```

And what about lazy evaluation using environments:

```
{bind {{x {+ 4 5}}}
  {bind {{y {+ x x}}}
    {bind {{z y}}
      {bind {{x 4}}
        z}}}}          []
-->
{bind {{y {+ x x}}}
  {bind {{z y}}
    {bind {{x 4}}
      z}}}             [x:={+ 4 5}]
-->
{bind {{z y}}
  {bind {{x 4}}
    z}}                [x:={+ 4 5}, y:={+ x x}]
-->
{bind {{x 4}}
  z}                   [x:={+ 4 5}, y:={+ x x}, z:=y]
-->
z                      [x:=4, y:={+ x x}, z:=y]
-->
y                      [x:=4, y:={+ x x}, z:=y]
-->
{+ x x}                [x:=4, y:={+ x x}, z:=y]
-->
{+ 4 4}                [x:=4, y:={+ x x}, z:=y]
-->
8                      [x:=4, y:={+ x x}, z:=y]
```

We have a problem! This problem should be familiar now, it is very similar to the problem that led us down the mistaken path of dynamic scoping when we tried to have first-class functions. In both cases, substitution always worked, and it looks like in both cases the problem is that we don't remember the environment of an expression: in the case of functions, it is the environment at the time of creating the closure that we want to capture and use when we go back later to evaluate the body of the function. Here we have a similar situation, except that we don't need a function to defer computation: *most* expressions get evaluated at some time in the future, so every time we defer such a computation we need to remember the lexical environment of the expression.

This is the major point that will make things work again: every expression creates something like a closure — an object that closes over an expression and an environment at the (lexical) place where that expression was used, and when we actually want to evaluate it later, we need to do it in the right lexical context. So it is like a closure except it doesn't need to be applied, and there are no arguments. In fact it is also a form of a closure — instead of closing over a function body and an environment, it closes over any expression and an environment. (As we shall see, lazy evaluation is tightly related to using nullary functions: *thunks*.)

Sloth: A Lazy Evaluator

So we implement this by creating such closure values for all expressions that are not evaluated right now. We begin with the Toy language, and rename it to "Sloth". We then add one more case to the data type of values which implements the new kind of expression closures, which contains the expression and its environment:

```
(define-type VAL
  [RktV  Any]
  [FunV  (Listof Symbol) SLOTH ENV]
```

```
  [ExprV                SLOTH ENV] ;*** new: expression and scope
  [PrimV ((Listof VAL) -> VAL)])
```

(Intuition#1: `ExprV` is a delayed evaluation and therefore it has the two values that are ultimately passed to `eval`. Intuition#2: laziness can be implemented with thunks, so we hold the same information as a `FunV` does, only there's no need for the argument names.)

Where should we use the new `ExprV`? — At any place where we want to be lazy and defer evaluating an expression for later. The two places in the interpreter where we want to delay evaluation are the named expressions in a bind form and the argument expressions in a function application. Both of these cases use the helper `eval*` function to do their evaluations, for example:

```
[(Bind names exprs bound-body)
 (eval bound-body (extend names (map eval* exprs) env))]
```

To delay these evaluations, we need to change `eval*` so it returns an expression closure instead of actually doing the evaluation — change:

```
(: eval* : SLOTH -> VAL)
(define (eval* expr) (eval expr env))
```

to:

```
(: eval* : SLOTH -> VAL)
(define (eval* expr) (ExprV expr env))
```

Note how simple this change is — instead of an `eval` function call, we create a value that contains the parts that would have been used in the `eval` function call. This value serves as a promise to do this evaluation (the `eval` call) later, if needed. (This is exactly why a Lazy Racket would make this a lazy evaluator: in it, *all* function calls are promises.)

Side note: this can be used in any case when you're using an eager language, and you want to delay some function call — all you need to do is replace (using a C-ish syntax)

```
int foo(int x, str y) {
  ...do some work...
}
```

with

```
// rename `foo':
int real_foo(int x, str y) {
  ...same work...
}

// `foo' is a delayed constructor, instead of a plain function
struct delayed_foo {
  int x;
  str y;
}
delayed_foo foo(int x, str y) {
  return new delayed_foo(x, y);
}
```

now all calls to `foo` return a `delayed_foo` instance instead of an integer. Whenever we want to force the delayed promise, we can use this function:

```
int force_foo(delayed_foo promise) {
  return real_foo(promise.x, promise.y);
}
```

You might even want to make sure that each such promise is evaluated exactly once — this is simple to achieve by adding a cache field to the struct:

```
int real_foo(int x, str y) {
  ...same work...
}

struct delayed_foo {
  int  x;
  str  y;
  bool is_computed;
  int  result;
}
delayed_foo foo(int x, str y) {
  return new delayed_foo(x, y, false, 0);
}

int force_foo(delayed_foo promise) {
  if (!promise.is_computed) {
    promise.result = real_foo(promise.x, promise.y);
    promise.is_computed = true;
  }
  return promise.result;
}
```

As we will see shortly, this corresponds to switching from a call-by-name lazy language to a call-by-need one.

Back to our Sloth interpreter — given the `eval*` change, we expect that `eval`-uating:

```
{bind {{x 1}} x}
```

will return:

```
(ExprV (Num 1) ...the-global-environment...)
```

and the same goes for `eval`-uating

```
{{fun {x} x} 1}
```

Similarly, evaluating

```
{bind {{x {+ 1 2}}} x}
```

should return

```
(ExprV (Call (Id +) (Num 1) (Num 2)) ...the-global-environment...)
```

But what about evaluating an expression like this one:

```
{bind {{x 2}}
  {+ x x}}
```

?

Using what we have so far, we will get to evaluate the body, which is a (Call …) expression, but when we evaluate the arguments for this function call, we will get `ExprV` values — so we will not be able to perform the addition. Instead, we will get an error from the function that `racket-func->prim-val` creates, due to the value being an `ExprV` instead of a `RktV`.

What we really want is to actually add two *values*, not promises. So maybe distinguish the two applications — treat `PrimV` differently from `FunV` closures?

```
(: eval* : SLOTH -> VAL)
(define (eval* expr) (ExprV expr env))
(: real-eval* : SLOTH -> VAL)
(define (real-eval* expr) (eval expr env))
(cases expr
  ...
  [(Call fun-expr arg-exprs)
   (define fval (eval fun-expr env))
   ;; move: (define arg-vals (map eval* arg-exprs))
   (cases fval
     [(PrimV proc) (proc (map real-eval* arg-exprs))] ; change
     [(FunV names body fun-env)
      (eval body (extend names (map eval* arg-exprs) fun-env))]
     ...)]
  ...)
```

This still doesn't work — the problem is that the function now gets a bunch of values, where some of these can still be `ExprV`s because the evaluation itself can return such values… Another way to see this problem is to consider the code for evaluating an `If` conditional expression:

```
[(If cond-expr then-expr else-expr)
 (eval* (if (cases (real-eval* cond-expr)
              [(RktV v) v] ; Racket value => use as boolean
              [else #t])   ; other values are always true
            then-expr
            else-expr))]
```

…we need to take care of a possible `ExprV` here. What should we do? The obvious solution is to use `eval` if we get an `ExprV` value:

```
[(If cond-expr then-expr else-expr)
 (eval* (if (cases (real-eval* cond-expr)
              [(RktV v) v] ; Racket value => use as boolean
              [(ExprV expr env) (eval expr env)] ; force a promise
              [else #t])   ; other values are always true
            then-expr
            else-expr))]
```

Note how this translates back the data structure that represents a delayed `eval` promise back into a real `eval` call…

Going back to our code for `Call`, there is a problem with it — the

```
(define (real-eval* expr) (eval expr env))
```

will indeed evaluate the expression instead of lazily deferring this to the future, but this evaluation might itself return such lazy values. So we need to inspect the resulting value again, forcing the promise if needed:

```
(define (real-eval* expr)
  (let ([val (eval expr env)])
    (cases val
      [(ExprV expr env) (eval expr env)]
      [else val])))
```

But we *still* have a problem — programs can get an arbitrarily long nested chains of `ExprV`s that get forced to other `ExprV`s.

```
{bind {{x true}}
  {bind {{y x}}
```

```
{bind {{z y}}
  {if z
    {foo}
    {bar}}}}}
```

What we really need is to write a loop that keeps forcing promises over and over until it gets a proper non-`ExprV` value.

```
(: strict : VAL -> VAL)
;; forces a (possibly nested) ExprV promise,
;; returns a VAL that is not an ExprV
(define (strict val)
  (cases val
    [(ExprV expr env) (strict (eval expr env))] ; loop back
    [else val]))
```

Note that it's close to `real-eval*`, but there's no need to mix it with `eval`. The recursive call is important: we can never be sure that `eval` didn't return an `ExprV` promise, so we have to keep looping until we get a "real" value.

Now we can change the evaluation of function calls to something more manageable:

```
[(Call fun-expr arg-exprs)
 (define fval (strict (eval* fun-expr)))        ;*** strict!
 (define arg-vals (map eval* arg-exprs))
 (cases fval
   [(PrimV proc) (proc (map strict arg-vals))]  ;*** strict!
   [(FunV names body fun-env)
    (eval body (extend names arg-vals fun-env))]
   [else (error 'eval "function call with a non-function: ~s"
                fval)])]
```

The code is fairly similar to what we had previously — the only difference is that we wrap a `strict` call where a proper value is needed — the function value itself, and arguments to primitive functions.

The `If` case is similar (note that it doesn't matter if `strict` is used with the result of `eval` or `eval*` (which returns an `ExprV`)):

```
[(If cond-expr then-expr else-expr)
 (eval* (if (cases (strict (eval* cond-expr))
              [(RktV v) v] ; Racket value => use as boolean
              [else #t])   ; other values are always true
            then-expr
            else-expr))]
```

Note that, like before, we always return `#t` for non-`RktV` values — this is because we know that the value there is never an `ExprV`. All we need now to get a working evaluator, is one more strictness point: the outermost point that starts our evaluation — `run` — needs to use `strict` to get a proper result value.

```
(: run : String -> Any)
;; evaluate a SLOTH program contained in a string
(define (run str)
  (let ([result (strict (eval (parse str) global-environment))])
    (cases result
      [(RktV v) v]
      [else (error 'run "evaluation returned a bad value: ~s"
                   result)])))
```

With this, all of the tests that we took from the Toy evaluator run successfully. To make sure that the interpreter is lazy, we can add a test that will fail if the language is strict:

```
;; Test laziness
(test (run "{{fun {x} 1} {/ 9 0}}") => 1)
(test (run "{{fun {x} 1} {{fun {x} {x x}} {fun {x} {x x}}}}") => 1)
(test (run "{bind {{x {{fun {x} {x x}} {fun {x} {x x}}}}} 1}") => 1)
```

[In fact, we can continue and replace all `eval` calls with `ExprV`, leaving only the one call in `strict`. This doesn't make any difference, because the resulting promises will eventually be forced by `strict` anyway.]

Getting more from Sloth

As we've seen, using `strict` in places where we need an actual value rather than a delayed promise is enough to get a working lazy evaluator. Our current implementation assumes that all primitive functions need strict values, therefore the argument values are all passed through the `strict` function — but this is not always the case. Specifically, if we have constructor functions, then we don't need (and usually don't want) to force the promises. This is basically what allows us to use infinite lists in Lazy Racket: the fact that `list` and `cons` do not require forcing their arguments.

To allow some primitive functions to consume strict values and some to leave them as is, we're going to change `racket-func->prim-val` and add a flag that indicates whether the primitive function is strict or not. Obviously, we also need to move the `strict` call around arguments to a primitive function application into the `racket-func->prim-val` generated function — which simplifies the `Call` case in `eval` (we go from (proc (map strict arg-vals)) back to (proc arg-vals)). The new code for `racket-func->prim-val` and its helper is:

```
(: unwrap-rktv : VAL -> Any)
;; helper for `racket-func->prim-val': strict and unwrap a RktV
;; wrapper in preparation to be sent to the primitive function
(define (unwrap-rktv x)
  (let ([s (strict x)])
    (cases s
      [(RktV v) v]
      [else (error 'racket-func "bad input: ~s" s)])))

(: racket-func->prim-val : Function Boolean -> VAL)
;; converts a racket function to a primitive evaluator function ...
(define (racket-func->prim-val racket-func strict?)
  (define list-func (make-untyped-list-function racket-func))
  (PrimV (lambda (args)
           (let ([args (if strict?
                           (map unwrap-rktv args)
                           args)])  ;*** use values as is!
             (RktV (list-func args))))))
```

We now need to annotate the primitives in the global environment, as well as add a few constructors:

```
;; The global environment has a few primitives:
(: global-environment : ENV)
(define global-environment
  (FrameEnv (list (list '+ (racket-func->prim-val + #t))
                  (list '- (racket-func->prim-val - #t))
                  (list '* (racket-func->prim-val * #t))
                  (list '/ (racket-func->prim-val / #t))
                  (list '< (racket-func->prim-val < #t))
                  (list '> (racket-func->prim-val > #t))
                  (list '= (racket-func->prim-val = #t))
                  ;; note flags:
                  (list 'cons  (racket-func->prim-val cons  #f))
                  (list 'list  (racket-func->prim-val list  #f))
```

```
                (list 'first (racket-func->prim-val car   #t)) ;**
                (list 'rest  (racket-func->prim-val cdr   #t)) ;**
                (list 'null? (racket-func->prim-val null? #t))
                ;; values
                (list 'true  (RktV #t))
                (list 'false (RktV #f))
                (list 'null  (RktV null)))
          (EmptyEnv)))
```

Note that this last change raises a subtle type issue: we're actually abusing the Racket `list` and `cons` constructors to hold Sloth values. One way in which this becomes a problem is the current assumption that a primitive function always returns a Racket value (it is always wrapped in a `RktV`) — but this is no longer the case for `first` and `rest`: when we use

```
{cons 1 null}
```

in Sloth, the resulting value will be

```
(RktV (cons (ExprV (Num 1) ...) (ExprV (Id null) ...)))
```

This leads to two problems: first, if we use Racket's `first` and `rest`, they will complain (throw a runtime error) since the input value is not a *proper* list (it's a pair that has a non-list value in its tail). To resolve that, we use the more primitive `car` and `cdr` functions to implement Sloth's `first` and `rest`.

The second problem happens when we try and grab the first value of this

```
{first {cons 1 null}}
```

we will eventually get back the `ExprV` and wrap it in a `RktV`:

```
(RktV (ExprV (Num 1) ...))
```

and finally `run` will strip off the `RktV` and return the `ExprV`. A solution to this is to make our `first` and `rest` functions return a value *without* wrapping it in a `RktV` — we can identify this situation by the fact that the returned value is already a VAL instead of some other Racket value. We can identify such values with the `VAL?` predicate that gets defined by our `define-type`, implemented by a new `wrap-in-val` helper:

```
(: unwrap-rktv : VAL -> Any)
;; helper for `racket-func->prim-val': strict and unwrap a RktV
;; wrapper in preparation to be sent to the primitive function
(define (unwrap-rktv x)
  (let ([s (strict x)])
    (cases s
      [(RktV v) v]
      [else (error 'racket-func "bad input: ~s" s)])))

(: wrap-in-val : Any -> VAL)
;; helper that ensures a VAL output using RktV wrapper when needed,
;; but leaving as is otherwise
(define (wrap-in-val x)
  (if (VAL? x) x (RktV x)))

(: racket-func->prim-val : Function Boolean -> VAL)
;; converts a racket function to a primitive evaluator function ...
(define (racket-func->prim-val racket-func strict?)
  (define list-func (make-untyped-list-function racket-func))
  (PrimV (lambda (args)
           (let ([args (if strict? (map unwrap-rktv args) args)])
             (wrap-in-val (list-func args))))))
```

Note that we don't need to worry about the result being an `ExprV` — that will eventually be taken care of by `strict`.

The Sloth Implementation

The complete Sloth code follows. It can be used to do the same fun things we did with Lazy Racket.

```
#lang pl

;;; ---------------------------------------------------------------
;;; Syntax

#| The BNF:
   <SLOTH> ::= <num>
             | <id>
             | { bind {{ <id> <SLOTH> } ... } <SLOTH> }
             | { fun { <id> ... } <SLOTH> }
             | { if <SLOTH> <SLOTH> <SLOTH> }
             | { <SLOTH> <SLOTH> ... }
|#

;; A matching abstract syntax tree datatype:
(define-type SLOTH
  [Num  Number]
  [Id   Symbol]
  [Bind (Listof Symbol) (Listof SLOTH) SLOTH]
  [Fun  (Listof Symbol) SLOTH]
  [Call SLOTH (Listof SLOTH)]
  [If   SLOTH SLOTH SLOTH])

(: unique-list? : (Listof Any) -> Boolean)
;; Tests whether a list is unique, guards Bind and Fun values.
(define (unique-list? xs)
  (or (null? xs)
      (and (not (member (first xs) (rest xs)))
           (unique-list? (rest xs)))))

(: parse-sexpr : Sexpr -> SLOTH)
;; parses s-expressions into SLOTHs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'bind more)
     (match sexpr
       [(list 'bind (list (list (symbol: names) (sexpr: nameds))
                          ...)
          body)
        (if (unique-list? names)
          (Bind names (map parse-sexpr nameds) (parse-sexpr body))
          (error 'parse-sexpr "duplicate `bind' names: ~s" names))]
       [else (error 'parse-sexpr "bad `bind' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: names) ...) body)
        (if (unique-list? names)
          (Fun names (parse-sexpr body))
```

```
          (error 'parse-sexpr "duplicate `fun' names: ~s" names))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(cons 'if more)
     (match sexpr
       [(list 'if cond then else)
        (If (parse-sexpr cond)
            (parse-sexpr then)
            (parse-sexpr else))]
       [else (error 'parse-sexpr "bad `if' syntax in ~s" sexpr)])]
    [(list fun args ...) ; other lists are applications
     (Call (parse-sexpr fun)
           (map parse-sexpr args))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> SLOTH)
;; Parses a string containing an SLOTH expression to a SLOTH AST.
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;;; ---------------------------------------------------------------
;;; Values and environments

(define-type ENV
  [EmptyEnv]
  [FrameEnv FRAME ENV])

;; a frame is an association list of names and values.
(define-type FRAME = (Listof (List Symbol VAL)))

(define-type VAL
  [RktV  Any]
  [FunV  (Listof Symbol) SLOTH ENV]
  [ExprV SLOTH ENV]
  [PrimV ((Listof VAL) -> VAL)])

(: extend : (Listof Symbol) (Listof VAL) ENV -> ENV)
;; extends an environment with a new frame.
(define (extend names values env)
  (if (= (length names) (length values))
    (FrameEnv (map (lambda ([name : Symbol] [val : VAL])
                     (list name val))
                   names values)
              env)
    (error 'extend "arity mismatch for names: ~s" names)))

(: lookup : Symbol ENV -> VAL)
;; lookup a symbol in an environment, frame by frame,
;; return its value or throw an error if it isn't bound
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(FrameEnv frame rest)
     (let ([cell (assq name frame)])
       (if cell
         (second cell)
         (lookup name rest)))]))

(: unwrap-rktv : VAL -> Any)
```

```
;; helper for `racket-func->prim-val': strict and unwrap a RktV
;; wrapper in preparation to be sent to the primitive function
(define (unwrap-rktv x)
  (let ([s (strict x)])
    (cases s
      [(RktV v) v]
      [else (error 'racket-func "bad input: ~s" s)])))

(: wrap-in-val : Any -> VAL)
;; helper that ensures a VAL output using RktV wrapper when needed,
;; but leaving as is otherwise
(define (wrap-in-val x)
  (if (VAL? x) x (RktV x)))

(: racket-func->prim-val : Function Boolean -> VAL)
;; converts a racket function to a primitive evaluator function
;; which is a PrimV holding a ((Listof VAL) -> VAL) function.
;; (the resulting function will use the list function as is,
;; and it is the list function's responsibility to throw an error
;; if it's given a bad number of arguments or bad input types.)
(define (racket-func->prim-val racket-func strict?)
  (define list-func (make-untyped-list-function racket-func))
  (PrimV (lambda (args)
           (let ([args (if strict? (map unwrap-rktv args) args)])
             (wrap-in-val (list-func args))))))

;; The global environment has a few primitives:
(: global-environment : ENV)
(define global-environment
  (FrameEnv (list (list '+ (racket-func->prim-val + #t))
                  (list '- (racket-func->prim-val - #t))
                  (list '* (racket-func->prim-val * #t))
                  (list '/ (racket-func->prim-val / #t))
                  (list '< (racket-func->prim-val < #t))
                  (list '> (racket-func->prim-val > #t))
                  (list '= (racket-func->prim-val = #t))
                  ;; note flags:
                  (list 'cons  (racket-func->prim-val cons  #f))
                  (list 'list  (racket-func->prim-val list  #f))
                  (list 'first (racket-func->prim-val car   #t))
                  (list 'rest  (racket-func->prim-val cdr   #t))
                  (list 'null? (racket-func->prim-val null? #t))
                  ;; values
                  (list 'true  (RktV #t))
                  (list 'false (RktV #f))
                  (list 'null  (RktV null)))
            (EmptyEnv)))

;;; ---------------------------------------------------------------
;;; Evaluation

(: strict : VAL -> VAL)
;; forces a (possibly nested) ExprV promise, returns a VAL that is
;; not an ExprV
(define (strict val)
  (cases val
    [(ExprV expr env) (strict (eval expr env))]
    [else val]))
```

```
(: eval : SLOTH ENV -> VAL)
;; evaluates SLOTH expressions
(define (eval expr env)
  ;; convenient helper
  (: eval* : SLOTH -> VAL)
  (define (eval* expr) (ExprV expr env))
  (cases expr
    [(Num n)   (RktV n)]
    [(Id name) (lookup name env)]
    [(Bind names exprs bound-body)
     (eval bound-body (extend names (map eval* exprs) env))]
    [(Fun names bound-body)
     (FunV names bound-body env)]
    [(Call fun-expr arg-exprs)
     (define fval (strict (eval* fun-expr)))
     (define arg-vals (map eval* arg-exprs))
     (cases fval
       [(PrimV proc) (proc arg-vals)]
       [(FunV names body fun-env)
        (eval body (extend names arg-vals fun-env))]
       [else (error 'eval "function call with a non-function: ~s"
                    fval)])]
    [(If cond-expr then-expr else-expr)
     (eval* (if (cases (strict (eval* cond-expr))
                  [(RktV v) v] ; Racket value => use as boolean
                  [else #t])   ; other values are always true
                then-expr
                else-expr))]))

(: run : String -> Any)
;; evaluate a SLOTH program contained in a string
(define (run str)
  (let ([result (strict (eval (parse str) global-environment))])
    (cases result
      [(RktV v) v]
      [else (error 'run "evaluation returned a bad value: ~s"
                   result)])))

;;; ---------------------------------------------------------------
;;; Tests

(test (run "{{fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{bind {{add3 {fun {x} {+ x 3}}}} {add3 1}}")
      => 4)
(test (run "{bind {{add3 {fun {x} {+ x 3}}}
                  {add1 {fun {x} {+ x 1}}}}
                {bind {{x 3}} {add1 {add3 x}}}}")
      => 7)
(test (run "{bind {{identity {fun {x} x}}
                  {foo {fun {x} {+ x 1}}}}
                {{identity foo} 123}}")
      => 124)
(test (run "{bind {{x 3}}
                {bind {{f {fun {y} {+ x y}}}}
                  {bind {{x 5}}
                    {f 4}}}}")
```

```
      => 7)
(test (run "{{{fun {x} {x 1}}
              {fun {x} {fun {y} {+ x y}}}}
             123}")
      => 124)

;; More tests for complete coverage
(test (run "{bind x 5 x}")      =error> "bad `bind' syntax")
(test (run "{fun x x}")         =error> "bad `fun' syntax")
(test (run "{if x}")            =error> "bad `if' syntax")
(test (run "{}")                =error> "bad syntax")
(test (run "{bind {{x 5} {x 5}} x}") =error> "duplicate*bind*names")
(test (run "{fun {x x} x}")     =error> "duplicate*fun*names")
(test (run "{+ x 1}")           =error> "no binding for")
(test (run "{+ 1 {fun {x} x}}") =error> "bad input")
(test (run "{+ 1 {fun {x} x}}") =error> "bad input")
(test (run "{1 2}")             =error> "with a non-function")
(test (run "{{fun {x} x}}")     =error> "arity mismatch")
(test (run "{if {< 4 5} 6 7}")  => 6)
(test (run "{if {< 5 4} 6 7}")  => 7)
(test (run "{if + 6 7}")        => 6)
(test (run "{fun {x} x}")       =error> "returned a bad value")

;; Test laziness
(test (run "{{fun {x} 1} {/ 9 0}}") => 1)
(test (run "{{fun {x} 1} {{fun {x} {x x}} {fun {x} {x x}}}}") => 1)
(test (run "{bind {{x {{fun {x} {x x}} {fun {x} {x x}}}}} 1}") => 1)

;; Test lazy constructors
(test (run "{bind {{l {list 1 {/ 9 0} 3}}}
              {+ {first l} {first {rest {rest l}}}}}")
      => 4)

;;; ---------------------------------------------------------------
```

Shouldn’t there be more `ExprV` promises?

You might notice that there are some apparently missing promises. For example, consider our evaluation of `Bind` forms:

```
[(Bind names exprs bound-body)
 (eval bound-body (extend names (map eval* exprs) env))]
```

The named expressions are turned into expression promises via `eval*`, but shouldn’t we change the first `eval` (the one that evaluates the body) into a promise too? This is a confusing point, and the bottom line is that there is no need to create a promise there. The main idea is that the `eval` function is actually called from contexts that actually *need* to be evaluated. One example is when we force a promise via `strict`, and another one is when `run` calls `eval`. Note that in both of these cases, we actuallly need a (forced) value, so creating a promise in there doesn’t make any difference.

To see this differently, consider how `bind` might be used *within* the language. The first case is when `bind` is the topmost expression, or part of a `bind` “spine”:

```
{bind {{x ...}}
  {bind {{y ...}}
    ...}}
```

In these cases we evaluate the `bind` expression when we need to return a result for the whole run, so adding an `ExprV` is not going to make a difference. The second case is when `bind` is used in an

expression line a function argument:

```
{foo {bind {{x ...}} ...}}
```

Here there is also no point in adding an `ExprV` to the `Bind` case, since the evaluation of the whole argument (the `Bind` value) will be wrapped in an `ExprV`, so it is already delayed. (And when it get forced, we will need to do the `bind` evaluation anyway, so again, it adds no value.)

A generalization of this is that when we actually call `eval` (either directly or via `strict`), there is never any point in making the result that it returns a promise.

(And if you'll follow this carefully and look at all of the `eval` calls, you will see that this means that *neither* of the `eval*`s in the `If` case are needed!)