CS 5510 Homework 3

Due: Wednesday, September 16th, 2015 11:59pm

Part 1 — Booleans

Start with the interpreter with function values, and extend the implementation to support boolean literals, an equality test, and a conditional form:

```
<Expr> = ....
       | true
       | false
       | {= <Expr> <Expr>}
       | {if <Expr> <Expr> <Expr>}
```

The `=` operator should only work on number values, and `if` should ony work when the value of the first subexpression is a boolean. The `if` form should evaluate its second subexpression only when the first subexpression's value is true, and it should evaluate its third subexpression only when the first subexpression's value is false. True and false are always spelled `true` and `false`, not `#t` or `#f`.

Note that you not only need to extend `ExprC` with new kinds of expressions, you will also need to add booleans to `Value`.

For example,

```
true
```

should produce a true value, while

```
{if true {+ 1 2} 5}
```

should produce 3, and

```
{if 1 2 3}
```

should report a "not a boolean" error.

As usual, update `parse` to support the extended language.

More examples:

```
(test (interp (parse '{if {= 2 {+ 1 1}} 7 8})
              mt-env)
      (interp (parse '7)
              mt-env))
(test (interp (parse '{if false {+ 1 {lambda {x} x}} 9})
              mt-env)
      (interp (parse '9)
              mt-env))
(test (interp (parse '{if true 10 {+ 1 {lambda {x} x}}})
              mt-env)
      (interp (parse '10)
              mt-env))
(test/exn (interp (parse '{if 1 2 3})
                  mt-env)
          "not a boolean")
```

Part 2 — Thunks

A *thunk* is like a function of zero arguments, whose purpose is to delay a computation. Extend your interpreter with a `delay` form that creates a thunk, and a `force` form that causes a thunk's expression to be evaluated:

```
<Expr> = ....
       | {delay <Expr>}
       | {force <Expr>}
```

A thunk is a new kind of value, like a number, function, or boolean.

For example,

```
{delay {+ 1 {lambda {x} x}}}
```

produces a thunk value without complaining that a function is not a number, while

```
{force {delay {+ 1 {lambda {x} x}}}}
```

triggers a "not a number" error. As another example,

```
 {let {[ok {delay {+ 1 2}}]}
   {let {[bad {delay {+ 1 false}}]}
     {force ok}}}
```

produces 3, while

```
 {let {[ok {delay {+ 1 2}}]}
   {let {[bad {delay {+ 1 false}}]}
     {force bad}}}
```

triggers a "not a number" error.

More examples:

```
(test/exn (interp (parse '{force 1})
                  mt-env)
          "not a thunk")
(test (interp (parse '{force {if {= 8 8} {delay 7} {delay 9}}})
              mt-env)
      (interp (parse '7)
              mt-env))
(test (interp (parse '{let {[d {let {[y 8]}
                                 {delay {+ y 7}}}]}
                        {let {[y 9]}
                          {force d}}})
              mt-env)
      (interp (parse '15)
              mt-env))
```

Note: In plain Racket, `delay` creates a *promise*. A promise is like a thunk, but when a promise is `force`d multiple times, the result from the first time is remembered and returned all the other times, so that the expression in a promise is evaluated at most once. We're not concerned with that facet of `delay` and `force`, for now.

Last update: Tuesday, September 8th, 2015

mflatt@cs.utah.edu