Homework 5 – Recursive Functions

This is an individual assignment. You need to provide full test coverage for the submitted work. This now includes explicit tests on all functions written.

Start with `lambda+if0.rkt`, which doesn't already include recursive binding and doesn't include `*` for multiplication.

Part 1 — Syntactic Sugar for Recursive Bindings

Extend the `parse` function so that it supports a `letrec` form for recursive function bindings.

```
<Exp> = ...
      | {letrec {[<Symbol> <Exp>]} <Exp>}
```

You should not change the `interp` function at all.

The September 26 lecture slides spell out how to extend the parser to make `letrec` work, especially at the end of part 4. You may find the following definition useful:

```
(define mk-rec-fun
  `{lambda {body-proc}
     {let {[fX {lambda {fX}
                 {let {[f {lambda {x}
                            {{fX fX} x}}]}
                   {body-proc f}}}]}
      {fX fX}}})
```

The above definition makes sense only if you can keep track of different languages and how they interact. The `mk-rec-fun` definition above is a Plait definition. The value of `mk-rec-fun` is a representation of the concrete syntax of a Curly expression. If you pass `mk-rec-fun` to `parse`, you get a Plait value that is an `interp`retable representation of a Curly expression.

Example:

```
  (test (interp (parse `{letrec {[f {lambda {n}
                                      {if0 n
                                           0
                                           {+ {f {+ n -1}} -
1}}}]}
                          {f 10}})
                 mt-env)
          (numV -10))
```

Part 2 — Implementing a Two-Argument Function in Curly

Define the Plait constant `plus` as a representation of the concrete syntax of a Curly expression such that

```
   (interp (parse (list->s-exp (list (list->s-exp (list plus
`n)) `m))) mt-env)
```

produces the same value as

```
   (interp (parse (list->s-exp (list `+ `n `m))) mt-env)
```

for any Plait number *n* and *m*.

In other words, you add a Plait definition

```
   (define plus `{lambda ....})
```

to the interepreter program, replacing the `....` with somethig that creates the desired Curly function.

You should not change the `interp` or `parse` function for this part.

Part 3 — Implementing a Recursive Function in the Curly

Define the Plait constant `times` such that

```
   (interp (parse (list->s-exp (list (list->s-exp (list times
`n)) `m))) mt-env)
```

produces the same value as `(numV (* n m))` for any non-negative Plait integers *n* and *m*.

You should not change the `interp` or `parse` function for this part.