POPL HW 7

Due: April 17th, at the stroke of midnight
Spiciness: Two chilis outa four
Groups of two or less.
Full test coverage is required or 20% point deduction.
Explicit tests are required for any functions you have written or new expressions added to the language.

Part 1 — true, false, =, and if

Start with typed-lambda.rkt. The implementation already includes a bool type, but no expressions of bool type.

Add support for true, false, {= <Exp> <Exp>}, and {if <Exp> <Exp> <Exp>} expressions, where = produces a boolean given two numbers, and if requires a boolean expression for the test.

Examples:

```
  (test (interp (parse `{if true 4 5})
                mt-env)
        (numV 4))

  (test (interp (parse `{if false 4 5})
                mt-env)
        (numV 5))

  (test (interp (parse `{if {= 13 {if {= 1 {+ -1 2}}
                                      12
                                      13}}
                            4
                            5})
                mt-env)
        (numV 5))

  (test (typecheck (parse `{= 13 {if {= 1 {+ -1 2}}
                                     12
                                     13}})
                   mt-env)
        (boolT))

  (test (typecheck (parse `{if {= 1 {+ -1 2}}
                               {lambda {[x : num]} {+ x 1}}
                               {lambda {[y : num]} y}})
                   mt-env)
        ;; This result may need to be adjusted after part 3:
        (arrowT (numT) (numT)))

  (test/exn (typecheck (parse `{+ 1 {if true true false}})
                       mt-env)
            "no type")
```

Part 2 — Pairs

Implement {pair <Exp> <Exp>}, {fst <Exp>}, and {snd <Exp>} expressions, as well as {<Type> * <Type>} types, as shown in video 10.

Examples (some of which depend on a choice of constructor and may not apply directly to your implementation):

```
  (test (interp (parse `{pair 10 8})
                mt-env)
        ;; Your constructor might be different than pairV:
        (pairV (numV 10) (numV 8)))

  (test (interp (parse `{fst {pair 10 8}})
                mt-env)
        (numV 10))

  (test (interp (parse `{snd {pair 10 8}})
                mt-env)
        (numV 8))

  (test (interp (parse `{let {[p : (num * num) {pair 10 8}]}
                          {fst p}})
                mt-env)
        (numV 10))

  (test (typecheck (parse `{pair 10 8})
                   mt-env)
        ;; Your constructor might be different than crossT:
        (crossT (numT) (numT)))

  (test (typecheck (parse `{fst {pair 10 8}})
                   mt-env)
        (numT))

  (test (typecheck (parse `{+ 1 {snd {pair 10 8}}})
                   mt-env)
        (numT))

  (test (typecheck (parse `{lambda {[x : (num * bool)]}
                             {fst x}})
                   mt-env)
        ;; Your constructor might be different than crossT:
        (arrowT (crossT (numT) (boolT)) (numT)))

  (test (typecheck (parse `{{lambda {[x : (num * bool)]}
                              {fst x}}
                            {pair 1 false}})
                   mt-env)
        (numT))

  (test (typecheck (parse `{{lambda {[x : (num * bool)]}
                              {snd x}}
                            {pair 1 false}})
                   mt-env)
```

```
      (boolT))

  (test/exn (typecheck (parse `{fst 10})
                       mt-env)
            "no type")

  (test/exn (typecheck (parse `{+ 1 {fst {pair false 8}}})
                       mt-env)
            "no type")

  (test/exn (typecheck (parse `{lambda {[x : (num * bool)]}
                                 {if {fst x}
                                     1
                                     2}})
                       mt-env)
            "no type")
```

Part 3 — Functions that Accept Multiple Arguments, Yet Again

With pairs, functions can accept multiple arguments by accepting paired values, but we can also add direct support for multiple arguments.

Change the interpreter to allow multiple function arguments and multiple arguments at function calls. The grammar of the language is now as follows (not counting the let sugar, whose syntax can remain limited to a single binding):

```
  <Exp> = <Num>
        | true
        | false
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {= <Exp> <Exp>}
        | <Sym>
        | {if <Exp> <Exp> <Exp>}
        | {lambda {[<Sym> : <Type>]*} <Exp>}
        | {<Exp> <Exp>*}
        | {pair <Exp> <Exp>}
        | {fst <Exp>}
        | {snd <Exp>}

  <Type> = num
         | bool
         | (<Type> * <Type>)
         | (<Type>* -> <Type>)
```

Examples:

```
  (test (interp (parse `{{lambda {}
                           10}})
                mt-env)
        (numV 10))

  (test (interp (parse `{{lambda {[x : num] [y : num]} {+ x y}}
                         10
                         20})
                mt-env)
```

```
      (numV 30))


(test (typecheck (parse `{{lambda {[x : num] [y : bool]} y}
                          10
                          false})
                 mt-env)
      (boolT))

(test/exn (typecheck (parse `{{lambda {[x : num] [y : bool]} y}
                              false
                              10})
                    mt-env)
          "no type")
```