CS 4110 – Programming Languages and Logics
Homework #7

Instructions This assignment may be completed alone or with a partner. You and your partner should submit a single solution on Gradescope. Please do not offer or accept any other assistance on this assignment. Apart from the automatic 48-hour extension, late submissions will not be accepted.

Submit these files on Gradescope: your completed `check.ml`, an archive called `tests.zip` containing your 10 tests and your description `tests.txt`, and a text file `debriefing.txt`. Because we grade anonymously, please do not include your name or NetID in your submission.

Exercise 1. This assignment is about implementing a type checker. We have written an interpreter for a version of the simply-typed λ-calculus with a few extensions:

- Integers and addition. The language uses the name `Int` for integers. This is a well-typed program, for example:

```
(lambda f : Int -> Int. (lambda x : Int. f (f (x + x))))
(lambda x : Int. x + 5) 16
```

- Booleans and **`if`**/**`then`**/**`else`** expressions. We also have the integer comparison operators <, >, and = and the standard Boolean operators **`and`**, **`or`**, and **`not`**. This program is well-typed for example, and evaluates to **`true`**:

```
(lambda x : Int. lambda y : Int.
  if x > y then x = 4 and y = 2 else false
) 4 2
```

- **`let`** expressions, which make things way more readable. This is also a valid program:

```
let double = lambda x : Int. x + x in
let reapply = lambda f : Int -> Int. lambda x : Int. f (f x) in
let quadruple = reapply double in
let hexadecatuple = reapply quadruple in
hexadecatuple 5
```

- Arbitrary-length tuples, which generalize pairs. This language uses a different syntax for projection from the language we formalized in class: use $e.n$ to get the nth element, **starting at zero**, of the tuple e. So instead of writing `#1 (4, 2)`, write `(4, 2).0`. You can write the type of a tuple as `T0 * T1 * ...` where each Tn is type. Here's a complete example program:

```
let map = lambda p : (Int -> Int) * (Int * Int). (p.0 p.1.0, p.0 p.1.1) in
let numbers = 2, 1 in
map ((lambda x : Int. x + x), numbers)
```

- Inductively-defined lists. The lists should be homogeneous, that is, each element must have the same type. The type of a list containing values of type τ is `List` τ.

  There are two constructors for lists. The `::`, or "cons," binary operator works like the same operator in OCaml. The **`empty`** `:` τ expression makes an empty list; to keep type checking straightforward for this assignment, it needs to be annotated with the element type. This expression has type (`List Int`):

```
empty : Int
```

The destructor for lists is a custom pattern matching expression. The ternary **match** expression takes in a list, an expression to evaluate if the list is empty, and an expression to evaluate if the list is non-empty. The expression for the non-empty case must evaluate to an abstraction that takes a two-element tuple as an argument, where the first element is the head of the list and the second element is the rest of the list.

This well-typed example program evaluates to 5:

```
let l = 5 :: (empty : Int) in
match l with_empty 0 with_head_rest (lambda p: (Int * List Int). p.0)
```

- A **fix** operator, which behaves like a fixed-point combinator in the untyped λ-calculus. Here's a well-typed implementation of a recursive factorial function:

```
let factp = (lambda f : Int -> Int.
    lambda n : Int. if n = 0 then 1 else n * (f (n - 1))
) in
let fact = fix factp in
fact 4
```

The interpreter can raise exceptions for poorly-behaved programs. Your job is to design and implement a sound type system for this language: that is, any program that "passes" your type checker will be guaranteed not to throw an exception and, if it eventually terminates, the resulting value will be of the appropriate type.

As usual, this archive contains the starter code, including the AST, lexer, parser, and interpreter. See the README for details about the setup. The 'examples' directory contains the short programs above; they are all well typed, so your type checker should accept all of them. You need to edit `check.ml` and submit along with your tests (see below) and debriefing.

Exercise 2. Develop a suite of 10 or more distinct test programs. Write 5 programs that are well-typed and 5 that are not. Because your type system is sound, all the well-typed programs should execute successfully and terminate with a value. (Please do not submit any non-terminating programs.) The ill-typed programs will either raise an exception in the interpreter or expose a source of incompleteness in your type system.

Include a short description of your tests in a file `tests.txt`. For each ill-typed test, say whether the program will throw an exception (it's a *true negative* for your type system) or it demonstrates incompleteness (it's a *false negative*).

Please submit a single archive `tests.zip` containing the test files named `good01.stlc` through `good05.stlc` and `bad01.stlc` through `bad05.stlc` as well as your `tests.txt`.

Debriefing

(a) How many hours did you spend on this assignment?
(b) Would you rate it as easy, moderate, or difficult?
(c) Did everyone in your study group participate?
(d) How deeply do you feel you understand the material it covers (0%–100%)?
(e) If you have any other comments, we would like to hear them! Please write them here or post a private note on Ed.