CS558 Programming Languages – Fall 2023 – Suggested Study Question Solutions for Lecture 7b

1. (a)

Here is the AST, with one node on each numbered line (arbitrarily numbered breadth-first).

```
1:        let f
         /     \
        /       \
2:   fun x       \
3:    |         fun y
4:    x           |
5:              fun z
                  |
6:               if
                / | \
7:             y  |  \
8:                z   \
9:                     @
                      / \
10:                  f   \
11:                       3
```

From this tree, we generate the following constraints:

```
Node #    Rule       Constraint
1         let        t2 = tf and t1 = t3
2         fun        t2 = tx -> t4
3         fun        t3 = ty -> t5
4         var        t4 = tx
5         fun        t5 = tz -> t6
6         if         t7 = Bool and t6 = t8 = t9
7         var        t7 = ty
8         var        t8 = tz
9         app        t10 = t11 -> t9
10        var        t10 = tf
11        int        t11 = Int
```

We can solve this by inspection:

First, using the identities for `t2`, `t4`, `t7`,`t8`, `t10`, `t11` we can substitute for these variables, leading to the following modified constraints:

```
2'      tf = tx -> tx
6'      ty = Bool and t6 = tz = t9
9'      tf = Int -> t9
```

Using `t1 = t3`, we can substitute for `t3` to get the modified contraint

```
3'      t1 = ty -> t5
```

(Choosing whether to get rid of `t1` or `t3` is fairly arbitrary, but we ultimately want to know the root expression type `t1`, so we keep that.)

Similarly, from `t6 = tz = t9`, we can substitute for `t6` and `t9` (again fairly arbitrary, but we ultimately want to know `tz`), getting

```
5'       t5 = tz -> tz
9''      tf = Int -> tz
```

Equating the two expressions (2' and 9") for `tf`, we get that

```
tx -> tx = Int -> tz
```

which implies that `tx = tz = Int`.

Summarizing, we have

```
t1 = Bool -> (Int -> Int)
tf = Int -> Int
tx = Int
ty = Bool
tz = Int
```

(b)

The AST:

```
1:    fun f
        |
2:    fun g
        |
3:    fun x
        |
4:      @
       / \
5:   f    \
6:          @
           / \
7:       g    \
8:             x
```

The generated constraints:

```
Node #    Rule    Constraint
-----------------------------
1         fun     t1 = tf -> t2
2         fun     t2 = tg -> t3
3         fun     t3 = tx -> t4
4         app     t5 = t6 -> t4
```

```
5          var      t5 = tf
6          app      t7 = t8 -> t6
7          var      t7 = tg
8          var      t8 = tx
```

Solution by inspection:

Using the identities from nodes 5, 6, and 7, we can rewrite the contraints for nodes 4 and 6 as

```
4'        tf = t6 -> t4
6'        tg = tx -> t6
```

There are no other contraints on `tx`, `t4` and `t6`, so the overall type must be parametric in (i.e. polymorphic over) these types. The type of the overall expression is

```
t1 = tf -> t2              (by 1)
   = tf -> (tg -> t3)      (by 2)
   = tf -> (tg -> (tx -> t4))   (by 3)
   = (t6 -> t4) -> ((tx -> t6) -> (tx -> t4))   (by 4' and 6')
```

Or, using more suggesting names for the polymorphic types, and the convention that `->` associates to to the right:

```
t1 = (tb -> tc) -> (ta -> tb) -> (ta -> tc)
tx = ta
tf = tb -> tc
tg = ta -> tb
```

which makes sense for a general-purpose “compose” function.