CS 496 – Typing Rules and Derivations
Exercise Booklet 3

Exercise 1

Provide typing derivations for the following expressions:

1. `if zero?(8) then 1 else 2`
2. `if zero?(8) then zero?(0) else zero?(1)`
3. `proc (x:int) { x-2 }`
4. `proc (x:int) { proc (y:bool) { if y then x else x-1 } }`
5. `let x=3 in let y = 4 in x-y`
6. `let two? = proc(x : int) { if zero?(x-2) then 0 else 1 } in (two? 3)`

Exercise 2

Recall that an expression `e` is *typable*, if there exists a type environment `tenv` and a type expression `t` such that the typing judgement `tenv` $\vdash$ `e :: t` is derivable. Argue that the expression `x x` (a variable applied to itself) is not typable.

Exercise 3

Give a typable term of each of the following types, justifying your result by showing a type derivation for that term.

1. `(bool->int)`
2. `((bool -> int) -> int)`
3. `(bool -> (bool -> bool))`
4. `((s -> t) -> (s -> t))`, for any types `s` and `t`.

Exercise 4

Show that the following term is typable:

```
letrec int double (x:int) = if zero?(x)
                            then 0
                            else (double (x-1)) + 2
in double
```

Exercise 5

What is the result of evaluating the following expressions in CHECKED?

```
> (check "
letrec int double (x:int) = if zero?(x)
                            then 0
                            else (double (x-1)) + 2
in (double 5)")
```

```
> (check "
letrec int double (x:int) = if zero?(x)
                            then 0
                            else (double (x-1)) + 2
in double")
```

```
> (check "
letrec bool double (x:int) = if zero?(x)
                            then 0
                            else -((double -(x,1)), -2)
in double")
```

```
> (check "
letrec bool double (x:int) = if zero?(x)
                            then 0
                            else 1
in double")
```

```
> (check "
letrec int double (x:bool) = if zero?(x)
                            then 0
                            else 1
in double")
```

Exercise 6

Suppose we add pairs to our language. This requires first adding *pair types*:

<Type> ::= int
<Type> ::= bool
<Type> ::= (<Type> -> <Type>)
<Type> ::= <<Type> * <Type>>

Our expressions are extended with a `pair(e1,e2)` construct to build new pairs and an `unpair (x,y)=e1 in e2` construct that given a an expression `e1` that evaluates to a pair, binds `x` and `y` to the first and second component of the pair in `e2`. Here are some examples of expressions in the extended language:

```
pair(3,4)

pair(pair(3,4),5)
```

```
4
pair(zero?(0),3)
6
pair(proc (x:int) { x-2 },4)
8
proc (z:<int*int>) { unpair (x,y)=z in x }
10
proc (z:<int*bool>) { unpair (x,y)=z in pair(y,x) }
```

You are asked to give typing rules for each of the two new constructs.