COL 765	**October 10, 2021**
Assignment 3	
Instructor: Subodh Sharma	**Due: 23:59 hrs Oct 20, 2021**

Problem 1:

1. Extend the language from assignment 2 to now have support for functions. In particular, support for the following concrete syntax should be provided:

   - **Anonymous function**: `fn` $(x :: typ) \Rightarrow$ `exp end`
   - **Named function**: `fun` $\langle\text{name}\rangle(y :: typ) :: typ \Rightarrow$ `exp end`
   - **Application**: `(f A)`

   It is worth noticing that one cannot define a recursive function through an anonymous function. For this assignment, recursive functions can be implemented only through named functions.

   We provide some examples indicating concrete and abstract syntax. Note that you can have a slightly different abstract syntax so long as it captures the essential aspects of the concrete syntax.

Concrete syntax	Abstract syntax
`(f 3)`	`AppExp ( VarExp ("f"), NumExp (3))`
`fn` $(x$ `::int`$) \Rightarrow 1$	`Fn("x", INT, NumExp(1))`
`fun currF` $(x ::$ `int`$) ::$ `int` $\rightarrow$ `int` $\Rightarrow$ `fn` $(y ::$ `int`$) \Rightarrow$ `y+1`	`FunExp(VarExp ("curryF"), VarExp ("x"), INT, ARROW(INT,INT), Fn ("y", INT, BinExp(PLUS, VarExp ("y"), NumExp(1))))`

   We will provide you with skeletal `Alex` and `Happy` files that you will have to complete for this assignment. Note that you are free to not use our provided files and extend the recursive descent parser that you submitted as part of assignment 2.

2. For the second part of this assignment, implement a type checker for this extended language. Each expression is either *well-typed* (*i.e.*, associated with at most one type) or an exception must be raised. For instance, `if e1 then e2 else e3 fi` expression is well-typed when $e1$ is a boolean type and $e2$ and $e3$ must have the same types. In order to implement the type checker, you must implement a type grammar (that provides you with rules that must be searched and a derivation has to be obtained to conclude the type checking).
   HINT: You may have to define a type environment, and define the domain and range types for each operator

3. Finally, if the type checking goes through successfully, then perform the evaulation of an input program for the extended language using an evaluation strategy *call-by-value*. Note that for evaluation of an application, one will have to consider computing evironment closure (capturing the enclosing environment).
   HINT: An `eval` function should be provided for each expression type. Note that you may have to import the AST module for your evaluator.

Input and output format:

1. Only file inputs will be considered for this assignment.
2. The parser output should be the same as expected in assignment 2, *i.e.*, pre-order traversal of the AST.
3. The type-checker output should throw appropriate exceptions, if any. For instance, when looking up a variable `x` in the environment, if no type is assigned then the typechecker should throw an error along the line: `Var x w/o a type`. Consider another example of function application `(f 3)`. If the function argument for `f` doesn't match the `int` type of 3, then the typechecker should throw an error along the lines: `Application argument mismatch in f`.

Submission: A single zip file containing all the relevant haskell files should be submitted. The file must be named in the following format: `entry-number.zip`.
Optional Helper files: We will provide you with optional helper templates (such as Typing.hs, Evaluator.hs, Ast.hs, grammar.y, and lexer.x) that you may chooose to complete and use in your submission. The choice is completely yours and it is not mandated that the supplied templates must be used for the submission.
Test inputs: We shall also release a set of test programs which you can use to test your implementation for type checking as well as evaluation.