CMPT 383 Comparative Programming Languages

Programming Assignment 3

This assignment is due by 11:59pm PT on Tuesday Mar 25, 2025. Please submit it to Canvas.

Requirements:

- This assignment must be your own work. No collaboration is permitted.
- You can learn the code on slides and start from it.
- You can only use library functions from the following modules: `Prelude`, `System.IO`, `System.Environment`, and `Data.Map.Strict`. Detailed information can be found on `https://hoogle.haskell.org`

Late policy:
Suppose you can get n (out of 100) points based on your code and report

- If you submit before the deadline, you can get all n points.
- If you submit between 11:59pm PT Mar 25 and 11:59pm PT Mar 26, you get $n - 10$ points.
- If you submit between 11:59pm PT Mar 26 and 11:59pm PT Mar 27, you get $n - 20$ points.
- If you submit after 11:59pm PT Mar 27, you get 0 points.

(100 points) Consider the following fragment of the FUN language

$$
\begin{array}{rcl}
e & ::= & c \mid b \mid x \mid \texttt{'('}\ e\ \texttt{')'} \\
& \mid & e\ \texttt{'+'}\ e \mid e\ \texttt{'-'}\ e \mid e\ \texttt{'=='}\ e \\
& \mid & \texttt{'if'}\ e\ \texttt{'then'}\ e\ \texttt{'else'}\ e \\
& \mid & \texttt{'lambda'}\ x\ \texttt{':'}\ t\ \texttt{'.'}\ e \\
& \mid & \texttt{'app'}\ e\ e \\
& \mid & \texttt{'let'}\ x\ \texttt{':'}\ t\ \texttt{'='}\ e\ \texttt{'in'}\ e \\
t & ::= & \texttt{'Int'} \mid \texttt{'Bool'} \mid \texttt{'('}\ t\ \texttt{')'} \\
& \mid & t\ \texttt{'->'}\ t
\end{array}
$$

$$c \in \mathbf{Int} \quad b \in \mathbf{Bool} \quad x \in \mathbf{Ident}$$

Here, e is the start symbol. c stands for an integer constant, b stands for a boolean constant, and x stands for an identifier (variable). The `->` operator is **right-associative**.

The typing rules of this language are defined as follows:

$$
\frac{\text{Int } c}{\Gamma \vdash c : Int}\ \text{(T-Int)} \qquad
\frac{\text{Bool } b}{\Gamma \vdash b : Bool}\ \text{(T-Bool)} \qquad
\frac{\Gamma \vdash e_1 : Int \quad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 + e_2 : Int}\ \text{(T-Plus)} \qquad
\frac{\Gamma \vdash e_1 : Int \quad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 - e_2 : Int}\ \text{(T-Minus)}
$$

$$
\frac{T \in \{Int, Bool\} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \texttt{==} e_2 : Bool}\ \text{(T-Eq)} \qquad
\frac{\Gamma \vdash e_1 : Bool \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T}\ \text{(T-ITE)} \qquad
\frac{\text{Ident } x \quad \Gamma(x) = T}{\Gamma \vdash x : T}\ \text{(T-Ident)}
$$

$$
\frac{\Gamma[x \triangleleft T_1] \vdash e : T_2}{\Gamma \vdash \texttt{lambda}\ x\texttt{:}T_1\texttt{.}e : T_1 \to T_2}\ \text{(T-Abs)} \qquad
\frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash \texttt{app}\ e_1\ e_2 : T_2}\ \text{(T-App)} \qquad
\frac{\Gamma[x \triangleleft T_1] \vdash e_1 : T_1 \quad \Gamma[x \triangleleft T_1] \vdash e_2 : T_2}{\Gamma \vdash \texttt{let}\ x\texttt{:}T_1\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : T_2}\ \text{(T-Let)}
$$

In this assignment, you need to write a type checker in Haskell to type check expressions in the FUN language. Specifically, given an expression e, the type checker computes the type of e if it is well-typed. If e is ill-typed, the type checker should output `Type Error`.

To avoid the complication of parsing, you can assume the type is represented as a value of the following `Type` data type in Haskell

```
data Type = TInt
          | TBool
          | TArr Type Type
          deriving (Eq, Ord, Read, Show)
```

where `TInt` represents the `Int` type, `TBool` represents the `Bool` type, and `TArr` represents the function type. For example, type `Int -> Bool -> Int` in FUN is represented as `TArr TInt (TArr TBool TInt)`.

Variable IDs are assumed to be strings:

```
type VarId = String
```

An expression of the FUN language is represented as a value of the following `Expr` type

```
data Expr = CInt Int
          | CBool Bool
          | Var VarId
          | Plus Expr Expr
          | Minus Expr Expr
          | Equal Expr Expr
          | ITE Expr Expr Expr
          | Abs VarId Type Expr
          | App Expr Expr
          | LetIn VarId Type Expr Expr
          deriving (Eq, Ord, Read, Show)
```

As indicated by the names of data constructors, `CInt` denotes an integer constant, `CBool` denotes a boolean constant, `Var` denotes a variable (identifier), `Plus` denotes the `+` operator, `Minus` denotes the `-` operator, `Equal` denotes the `==` operator, `ITE` denotes the if-then-else expression, `Abs` denotes the function abstraction (lambda), `App` denotes the function application, and `LetIn` denotes the let-in expression. There is no data constructor for parenthesized expressions. For example,

- `x1` is represented as `Var "x1"`
- `1 + 2` is represented as `Plus (CInt 1) (CInt 2)`
- `if True then 1 else 2` is represented as `ITE (CBool True) (CInt 1) (CInt 2)`
- `lambda x:Int.x` is represented as `Abs "x" TInt (Var "x")`
- `app (lambda x:Int.x) 1` is represented as `App (Abs "x" TInt (Var "x")) (CInt 1)`
- `let x:Int = 1 in x` is represented as `LetIn "x" TInt (CInt 1) (Var "x")`

Note that you need to use exactly the same definition of `Type`, `VarId`, `Expr`, and their deriving clauses as written in this document. Otherwise, you will lose points because potential grading scripts may not work as expected.

Detailed Steps

1. Use `Map` from `Data.Map.Strict` to define the `Env` type for the typing environment, i.e., finish the following declaration

```
type Env = ...
```

2. Write an auxiliary function `typingArith :: Maybe Type -> Maybe Type -> Maybe Type` that

- returns `Just TInt` if both arguments are `Just TInt`
- returns `Nothing` otherwise

3. Write an auxiliary function `typingEq :: Maybe Type -> Maybe Type -> Maybe Type` that
   - returns `Just TBool` if both arguments are `Just TInt`
   - returns `Just TBool` if both argumnets are `Just TBool`
   - returns `Nothing` otherwise
4. Write a function `typing :: Env -> Expr -> Maybe Type` that takes a typing environment and a FUN expression as input and produces as output the type of that expression based on the typing rules. If there is a type error, it returns `Nothing`. Note that you can use the auxiliary function `typingArith` and `typingEq` in this function.
5. Write a **simple** function `readExpr :: String -> Expr` that can read a value of `Expr` type from the corresponding string, such as `"CInt 1"`.
6. Write a function `typeCheck :: Expr -> String` that takes an expression as input and produces a string as output representing the type checking result. Specifically,
   - If the expression is well-typed and a value `v` of type `Type` is obtained, generate the output using `show v`.
   - If the expression is ill-typed, output string "`Type Error`".
7. Write a `main` to handle IO and put everything together.

The program must be in a form that GHC can compile. It needs to take one command-line argument denoting the path to the expression file. Each line of the file contains a string representing a FUN expression, and the program needs to print the result of `typeCheck` on each expression to the console.

Sample Input and Output

Suppose we have a file called `exprs.txt` that contains the following six lines:

```
Var "x1"
Plus (CInt 1) (CInt 2)
ITE (CBool True) (CInt 1) (CInt 2)
Abs "x" TInt (Var "x")
App (Abs "x" TInt (Var "x")) (CInt 1)
LetIn "x" TInt (CInt 1) (Var "x")
```

After compiling, we can run the executable and get

```
$ ./P3_SFUID exprs.txt
Type Error
TInt
TInt
TArr TInt TInt
TInt
TInt
```

Deliverable

A zip file called `P3_SFUID.zip` that contains at least the followings:

- A file called `P3_SFUID.hs` that contains the source code of your Haskell program. You can have multiple source files if you want, but you need to make sure `ghc P3_SFUID.hs` can compile.
- A report called `P3_SFUID.pdf` that explains the design choices, features, issues (if any), and anything else that you want to explain about your program.