Project Assignment Report

All the data types and functions are included in a single file. The file is divided into parts for each of the functions and its helper functions. The structure was small enough to be designed in a single file and it's easy to follow.

Design Choices:

The program is contained in a single file, namely **P3_Jashanraj_Gosain.hs**

Functions and definitions are unified (in a single file structure) based on the duty/functionality of the program. All the functions have related functionality, therefore defined in a single file. Types, functions and type checking are interdependent and change in any function will advice updates in other functions and types. It is viable to maintain and update the program without having to change other files when placed in a single file. As each function will probably need some alteration if any of the other function is changed, it is better to keep the dependent functions in a single file structure. It will reduce the need to alter outside environments/files every time any function/type is changed which prevents foreign environment affecting the local environment and provide a robust system.

The file is divided into four parts: Imports, type definitions, functions, Main module.

P3_jashanraj_gosain.hs **imports module** Data.Map.Strict, System.Environment and System.IO.

Type and data declaration part defines VarId, Type, Expr and Env types and data.

- VarId is string type.
- data Type has three data constructors 'TInt', 'TBool', 'TArr Type Type' and it derives Eq, Ord, Read and Show classes.
- Expr has data constructors for CBool, Var, Plus, Minus, Equal, ITE, Abs, App, LetIn and it derives Eq, Ord, Read and Show classes.
- Env is a Map which takes VarId as key and Type value.

typingArith is a function with signature:: Maybe Type -> Maybe Type -> Maybe Type. It takes in **two parameters** of type **'Maybe Type'**. It checks if both the parameters have value (Just TInt) or not. If both parameters match the value **'(Just TInt)'**, it returns **'(Just TInt)'**, otherwise it returns **'Nothing'**.

typingEq is a function with signature:: Maybe Type -> Maybe Type -> Maybe Type. It takes in **two** parameters of type '**Maybe Type'**. It checks **if** both the **parameters** have **same values '(Just TInt)' or '(Just TBool)**'. If same, it **returns** the value as parameters **('Just TBool' if parameter value is 'Just TBool' or 'Just TInt' if the parameter value is 'TInt'**). If parameters have nonmatching values or values other than stated, it **returns 'Nothing'**.

typing is a function with signature:: Env -> Expr -> Maybe Type. It takes in an environment (it stores type values for different varId), Expr and **returns** a **value** of **type** '**Maybe Type'**. It evaluates the type of the return value of expression.

It checks for several patterns:

- If the expression is **CInt**, it returns '**Just TInt**'
- If the expression is **CBool** it returns '**Just TBool'**
- If the expression is (**Var var)**, it returns the type values with key 'var' in the environment provided in the parameters.
- If the expression is **(Plus e1 e2)**, it **recursively checks** for the **type** of **e1** and **e2** with the same environment. If the **type of e1 is not TInt,** it **returns** '**Nothing'**, **otherwise** it **checks** for the **type of e2**. If **both e1 and e2** have **type 'TInt'**, it **returns** '**Just TInt'**. All **other cases return** '**Nothing'**.
- If the expression is **(Minus e1 e2)**, it **recursively checks** for the **type** of **e1** and **e2** with the same environment. If the **type** of **e1** is **not TInt**, it **returns** '**Nothing'**, otherwise it **checks** for the **type** of **e2**. If **both e1 and e2 have type** '**TInt'**, it **returns 'Just TInt'**. All **other cases return** '**Nothing'**.
- If the expression is **(Equal e1 e2)**, it **recursively checks** for the **type** of **e1** and **e2** with the same environment. If the **type is 'TInt' or 'TBool'** for **both**, it **returns** '**Just TInt**' or '**Just TBool**' **respectively**. If **both e1** and **e2** have **different types**, it **returns** '**Nothing'**. Its implementation uses a case value block to first find the type value of e1 and then e2 for both the cases of 'TInt' or 'TBool'. If any of the e1 or e2 return 'Nothing' type, function returns 'Nothing'.
- If the expression is **(ITE c t e)**, it recursively checks for the type of condition. When the **type of** condition 'c' is **not TBool**, it **returns** '**Nothing'**. If the **type of** condition '**c**' is '**TBool'**, it **checks** if the **types** of firstreturn **(t**) or secondreturn **(e)**

is same. For **different types** for firstreturn (**t**) and secondreturn (**e**) it **returns 'Nothing'** and for **same type**, it **returns** '**Just type**' where **type is** the **returned type** of firstreturn (**t**) or secondreturn (**e**).

- If the expression is **(Abs v t1 e)**, function checks for the type of 'e' recursively but with a different environment. This environment contains the values from current environment 'env' plus the type 't1' for key 'v'. If the return type of 'e' is not 'Nothing', it returns '**Just (TArr t1 t2)**' where t2 is the returned type of 'e'. For all **other cases**, it **returns** '**Nothing'**.
- If the expression is **(LetIn var t e1 e2)**, function recursively calculates the type of 'e1' and 'e2' with the **environment** containing **Type value 't' for key 'var'** in both the calls. It compares the return value of the **recursive calls for 'e1' and 'e2' types**. If **same**, it **returns** that **value**. For all **other cases**, it **returns** '**Nothing'**.

readExpr has a type signature:: String -> Expr. It converts the expression from **string to Expr** value. It uses the **derivation of read in Haskell** to convert the value from **string to Expr object**.

typeCheck has a type signature:: Expr -> String. It takes in the expression as an Expr object and calls 'typing' with an empty environment to **calculate** the **type of given expression**. If the value **returned by** typing **function call** is **'(Just t)**', it **returns** '**show t**' (derived from Show class) which is a string value of expression type. For all **other cases** (case which returns 'Nothing') string value of "**Type Error**" is **returned**.

The last section consists of the **main module** of the file. This module takes in FileName as argument from the user and reads the indicated file data assuming each line is a FUN language expression. It then converts string lines to Expr object using readExpr function and prints the result of mapping typeCheck function over the list of Expr objects returned by readExpr function.