1. The aim of this assignment is to build a simulator of a programming language. The programming language contains variables (referred to by their name, given as a string), which can store an integer value:

```haskell
type Data = Integer

type Variable = String
```

The variable values are to be stored in a lookup table relating variable names and values, using the data synonym Store:

```haskell
type Store = [(Variable, Data)]
```

(a) Define the empty Store, which contains no information about variables.

(b) Define a function `getVal`, which takes a Store and a variable name, and returns its value. You may assume that the variable is defined in the Store.

```haskell
getVal :: Store -> Variable -> Data
```
(c) Define a function `setVar`, which takes a `Store`, a variable name and a
value, and returns an updated `Store` with the variable value updated.

```
setVal :: Store -> (Variable, Data) -> Store
```

2. Numeric expressions may appear in the programs we will simulate. They
will be expressed using the following datatype:

```
data Exp =
    Val Data
  | Var Variable
  | Exp :-: Exp
  | Exp :+: Exp
  | Exp :*: Exp
```

Note that `:+:` is interpreted as addition, `:-:` as subtraction and `:*:` as
multiplication.

(a) Define a function `getValExp`, which takes a `Store` and an expression,
and returns the value of the expression. You may assume that all
variables are defined in the `Store`.

```
getValExp :: Store -> Exp -> Data
```

(b) Hence or otherwise, define a function `setVarExp`, which takes a
`Store`, a variable name and an expression, and returns an updated
`Store` with the variable value updated.

```
setValExp :: Store -> (Variable, Exp) -> Store
```

3. Finally, we come to the program. The simple programming language,
which includes loops, conditionals and sequential composition will be ex-
pressed a datatype in Haskell.

```
data Prg =
    Skip
  | PrintS String
  | PrintE Exp

  | Declare Variable
  | Variable := Exp

  | Prg :> Prg
  | IfThenElse Exp (Prg, Prg)
  | While Exp Prg
```
The program `Skip` does nothing. The program `PrintS` ‘prints’ the given string to the output. Similarly, `PrintE` prints the value of the given expression to the output.

Declare declares a variable, assigning its initial value to 0. The `:=` operator performs variable assignment.

The `:>` operator is sequential composition, while `IfThenElse` and `While` are conditional and while-loops respectively. The condition in both cases resolves to true if the value of the expression is not zero.

A factorial program can be programmed in this small language as follows:

```plaintext
counter = "counter"
result = "result"

factorial n =
  Declare counter
  :> Declare result
  :> result := Val 1
  :> counter := Val n
  :> While (Var counter)
    ( result := (Var result) :*: (Var counter)
      :> counter := (Var counter) :-: (Val 1)
    )
  :> PrintS ("Factorial of " ++ show n ++ " is:")
  :> PrintE (Var result)
```

(a) The simulate function takes a Store and a program, and returns an updated store and list of output strings:

```plaintext
simulate :: Store -> Prg -> (Store, [String])
```

Using pattern matching, define `simulate` for `Skip`, `PrintS` and variable declarations.

(b) Add the cases for `PrintE` and variable assignment.

(c) Sequential composition can be defined by simulating the two instructions in sequence. Complete the following case:

```plaintext
simulate store (p1 :> p2) =
  let (store’, outs1) = simulate store p1
      (store’’, outs2) = simulate store’ p2
      in ...
```

(d) Define `simulate` for conditionals.

(e) We can ‘cheat’ by simulating while-loops in terms of conditionals and sequential composition:
simulate store (While condition program) =
simulate store
    (IfThenElse condition
        ( program :- While condition program
        , Skip
        )
    )

Using this definition, test your program with the factorial program.