Pamphlet 2, INF222, Spring 2023

2.1 Calculator with registers

The following shows the abstract syntax for a register based calculator on integers. Note how `CalcExprAST` has been extended with a case `Reg Register`, where `Register` is an enumeration of 10 distinct register names.

```haskell
-- | AST for register based integer calculator.
--
-- Author Magne Haveraaen
-- Since 2020-03-14

module Pam2RegisterAST where


-----------------------

-- | Expressions for a calculator with 10 registers.
-- The calculator supports literals and operations
-- Addition, multiplication, and subtraction/negation.
data CalcExprAST
  = Lit Integer
  | Add CalcExprAST CalcExprAST
  | Mult CalcExprAST CalcExprAST
  | Sub CalcExprAST CalcExprAST
  | Neg CalcExprAST
  | Reg Register
  deriving (Eq, Read, Show)

-- | Statement for setting a register
data CalcStmtAST
  = SetReg Register CalcExprAST
  deriving (Eq, Read, Show)

-- | Enumeration of the 10 registers.
data Register
  = Reg0
  | Reg1
  | Reg2
  | Reg3
  | Reg4
  | Reg5
  | Reg6
  | Reg7
  | Reg8
  | Reg9
  deriving (Eq, Read, Show)



-----------------------
```

```haskell
-- | A few ASTs for register based CalcExprAST.
calculatorRegisterAST1
  = Lit 4
calculatorRegisterAST2
  = Neg (Mult (Add (Lit 3) (Sub (Lit 7) (Lit 13))) (Lit 19))
calculatorRegisterAST3
  = Add (Reg Reg1) (Reg Reg4)
calculatorRegisterAST4
  = Reg Reg2

-- | A few ASTs for setting registers CalcStmtAST.
calculatorSetRegisterAST1
  = SetReg Reg4 calculatorRegisterAST1
calculatorSetRegisterAST2
  = SetReg Reg1 calculatorRegisterAST2
calculatorSetRegisterAST3
  = SetReg Reg2 calculatorRegisterAST3
calculatorSetRegisterAST4
  = SetReg Reg1 calculatorRegisterAST4

------------------------
```

The use of registers also introduces statements `CalcStmtAST` for setting values into registers.

The AST file ends with some example expressions and statements in the register calculator language.

2.2 Store

The introduction of registers induces the need for a store to keep track of the register values.

```haskell
-- | Semantics for register based integer calculator.
-- The values of the registers are stored in a Store.
--
-- Author Magne Haveraaen
-- Since 2020-03-14

module Pam2RegisterStore where

-- Use Haskell's array data structure
import Data.Array


------------------------

-- | A Store for a register calculator is an array with 10 integer elements.
-- The access functions getregister/setregister need to translate between register and array index.
type Store = Array Integer Integer

-- | Defines a store for 10 registers, all initialised to 0.
registerStore :: Store
registerStore = array (0,9) [(i,0) | i <-[0..9]]
```

```haskell
-- | Get the value stored for the given register.
getStore :: Store -> Integer -> Integer
getStore store ind =
  if 0 <= ind && ind < 10
  then store ! ind
  else error $ "Not a register index " ++ (show ind)

-- | Set the value stored for the given register.
setStore :: Integer -> Integer -> Store -> Store
setStore ind val store =
  if 0 <= ind && ind < 10
  then store // [(ind, val)]
  else error $ "Not a register index " ++ (show ind) ++ " for " ++ (show val)
```

The store above handles 10 distinct indices and stores integers. The store is initialised to contain only zeroes in `registerStore`. It also explicitly checks, in the functions `getStore` and `setStore`, that only integers `0..9` are used as indices.

The Store is implemented using the Haskell standard library `Array` data structure, see chapter 14 of `https://www.haskell.org/onlinereport/haskell2010/` for more details.

2.3 Task

The task is again to implement an interpreter, this time for the register calculator. The interpreter needs three functions:

- `evaluate :: CalcExprAST -> Store -> Integer`
  to evaluate a calculator expression given a store.
- `execute :: CalcStmtAST -> Store -> Store`
  to set the value of a calculator expression to a register in the store.
- `getRegisterIndex :: Register -> Integer`
  to map a register to an index in the store.

You should also write a unit test for the interpreter.

2.3.1 Main method

Below is a `main` method that will work nicely with the functions above.

It uses the library `System.Console.Haskeline` for IO. This gives a really professional line editor for entering calculator commands. The pattern used is called REPL (read-evaluate-print-loop), a standard pattern for interactive tools in Haskell. See `https://hackage.haskell.org/package/haskeline-0.7.4.0/docs/System-Console-Haskeline.html` for details.

The `main` method also uses the `readMaybe :: Read a => String -> Maybe a` function from `Text.Read` to parse the input string. If the parse (reading) fails, it will return `Nothing`, allowing the function to give an error message and continue reading input. Note how the call `readMaybe str` is explicitly typed to induce a parsing of `CalcStmtAST`.

```haskell
main = do
  putStrLn $ "-- Interactive register calulator --"
  runInputT defaultSettings (loop registerStore)
  where
  -- Parses and executes CalcStmtAST and prints what happens.
  -- The recursive call to loop must update the store.
  loop :: Store -> InputT IO ()
  loop state = do
    input <- getInputLine "¢ "
    case input of
      Nothing -> return ()
      Just "" ->
        do outputStrLn $ "Finished" ; return ()
      Just "show" ->
        do outputStrLn $ "state = " ++ (show state) ; loop state
      Just str -> do
        case readMaybe str :: Maybe CalcStmtAST of
          Nothing -> do
            outputStrLn $ "Not a statement: " ++ (show str)
            loop state
          Just stmt -> do
            let SetReg reg expr = stmt
            outputStrLn $ (show reg) ++ " = " ++ (show $ evaluate expr state)
            loop $ execute stmt state
```