Pamphlet 4, INF222, Spring 2021

4.1 Calculator with variables

In this series we started out with a simple calculator, then added an enumeration of registers, before we changed the register type to the more open ended string type but kept the same names and number of registers. Now planning a complex computation with a limited amount of registers is demanding (in fact NP complete)[1]. So instead of using a fixed set of registers we can open up for an unbounded set of variables.

We get this flexibility by allowing the statements `CalcStmtAST` to declare new variables and assign to existing variables, and expressions `CalcExprAST` to use existing variables. This supports safety since programs are less vulnerable when variables are declared before use[2].

```haskell
-- | AST for variable based integer calculator.
--
-- Author Magne Haveraaen
-- Since 2020-03-19

module CalculatorVariableAST where



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

-- | Expressions for a calculator with variables.
-- 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
  | Var String
  deriving (Eq, Read, Show)

-- | Statement for setting and changing a variable
data CalcStmtAST
  = SetVar String CalcExprAST
  | AssVar String CalcExprAST
  deriving (Eq, Read, Show)



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

[1] For those especially interested: https://en.wikipedia.org/wiki/Register_allocation

[2] Forcing declaration before use reduces the chance for a misspelling in the code to introduce a new variable that is not used elsewhere. It is an old story that NASA lost one of their early space rockets due to a misprint in a Fortran program. The misprint introduced a new variable which was assigned to but not used anywhere.

Now forcing declaration of variables only helps if the chance of misspelling a variable name is detectable. This can be formally captured by *Hamming distance* on variable names. In practice it means that different variables should have easy to discern variable names. We have learned that variable names should be descriptive, but we should also keep the length of a variable name down so it is easy to recognise it and also the context where it appears. Like all things programming, finding good variable names is a mixture between an art and a skill.

```haskell
-- | A few ASTs for variable based CalcExprAST.
calculatorVariableAST1
  = Lit 4
calculatorVariableAST2
  = Neg (Mult (Add (Lit 3) (Sub (Lit 7) (Lit 13))) (Lit 19))
calculatorVariableAST3
  = Add (Var "Reg1") (Var "Reg4")
calculatorVariableAST4
  = Var "Reg2"

-- | A few CalcStmtASTs for setting and assigning variables.
calculatorSetVariableAST1
  = SetVar "Reg4" calculatorVariableAST1
calculatorSetVariableAST2
  = SetVar "Reg1" calculatorVariableAST2
calculatorSetVariableAST3
  = AssVar "Reg2" calculatorVariableAST3
calculatorSetVariableAST4
  = AssVar "Reg1" calculatorVariableAST4

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

The example variables in the example ASTs uses the old register names as variable names to reduce the difference with the previous code[3].

Keeping track of user defined variable names in addition to their values requires a state data structure rather than our previous 10 element Store.

4.2 Calculator State

The State data structure needs to keep track of two items:

- Which variables have been declared (Environment).
- What is the value associated with a variable (Store).

The link from the variable name to the store value is the store index, which are simple integer indices.

The state does not support scopes, since our source calculator programming language has no constructs for scoping.

The state supports the following functions.

- `newstate :: State`
  create a new state with empty variable environment and empty store.
- `getvalue :: String -> State -> Integer`
  `getvalue vname state`
  get the value associated to the variable `vname` in the `state`. If the variable name is not known, the function will crash with an error message.
- `addvariable :: String -> Integer -> State -> State`
  `addvariable vname value state`

[3] Is this really a good idea?

add a new variable `vname` with `value` to `state`. If the variable name is already known in the state, the function will crash with an error message.

- `changevalue :: String -> Integer -> State -> State`
  `changevalue vname value state`
  change the value of a known variable `vname` to `value` in `state`. If the variable name is unknown in the state, the function will crash with an error message.

```haskell
-- | Semantics for variable based integer calculator.
-- It uses State to keep track of variables and their values.
-- This is separated into a environment which keeps track of variables, and
-- a store which keeps track of their (changing) values.
--
-- Author Magne Haveraaen
-- Since 2020-03-19

module CalculatorState where

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


------------------
-- | A state is an environment of variable-store index associations, and
-- a store which at each store index keeps a value (for that variable).
type State = (Environment, Store)

-- | A new state is an empty environment with an empty store.
newstate :: State
newstate = (emptyenvironment, emptystore)

-- | Gets the value linked to the variable in the state.
getvalue :: String -> State -> Integer
getvalue vname (env, store) =
  case lookup vname env of
    Just loc -> getstore store loc
    Nothing -> error $ "Variable␣" ++ vname ++ "␣not␣found␣in␣state␣" ++ (show env)

-- | Add a new variable with value to the state.
addvariable :: String -> Integer -> State -> State
addvariable vname value (env, store) = (newenv, newstore)
  where
    (newhigh, newstore) = enlargestore store value
    newenv = addenv vname newhigh env

-- | Changes the value associated with a known variable.
changevalue :: String -> Integer -> State -> State
changevalue vname value (env, store) = (env, newstore) where
  newstore = case lookup vname env of
    Just loc -> setstore loc value store
    Nothing -> error $ "Variable␣" ++ vname ++ "␣not␣found␣in␣state␣" ++ (show env)
```

```
--------------------
-- | An Environemnt for a calculator with variables.
-- It stores an association list of distinct variable names and their store index.
-- As such, it can be searched by the Haskell standard function
-- lookup :: Eq a => a -> [(a, b)] -> Maybe b
type Environment = [( String , Integer )]

-- | Defines an empty environment
emptyenvironment :: Environment
emptyenvironment = []

-- | Add a new variable (and a store index) to the environment.
addenv :: String -> Integer -> Environment -> Environment
addenv vname ind env =
  case lookup vname env of
    Just loc -> error $ "New␣variable␣" ++ (show (vname,ind))
             ++ "␣already␣ registered ␣in␣" ++ (show env)
    Nothing -> (vname,ind):env


--------------------
-- | A Store for a calculator is an array where the number of indices
-- corresponds to the number of distinct variables.
type Store = Array Integer Integer

-- | Defines an empty store
emptystore :: Store
emptystore = array (0,-1) []

-- | Get the value stored for the given index.
getstore :: Store -> Integer -> Integer
getstore store ind =
  if low <= ind && ind <= high
  then store ! ind
  else error $ "Not␣a␣store␣index␣" ++ (show ind)
  where (low, high) = bounds store

-- | Set the value stored at the given index.
setstore :: Integer -> Integer -> Store -> Store
setstore ind val store =
  if low <= ind && ind <= high
  then store // [( ind , val )]
  else error $ "Not␣a␣store␣index␣" ++ (show ind) ++ "␣for␣" ++ (show val)
  where (low, high) = bounds store

-- | Get next store index and increase store size with one and set value at new location.
enlargestore :: Store -> Integer -> (Integer , Store )
enlargestore store value = (newhigh, newstore )
  where
    (low , high ) = bounds store
    newhigh = high + 1
    storelist = assocs store
    newstore = array (low ,newhigh) ( storelist ++[(newhigh, value ) ])
```

```
--------------------
-- | Unit tests for State.
 unittestCalculatorState   = do
   print  $ "--␣ unittestCalculatorState "
   -- putStrLn $ "Empty state = " ++ (show newstate)
   let  state1  = addvariable  "v1"  1  newstate
   let  state2  = addvariable  "v2"  4  state1
   let  state3  = addvariable  "v3"  9  state2
   let  state4  = changevalue  "v2"  25  state3
   -- putStrLn $ "Value of v1 == " ++ (show $ getvalue "v1" state4)
   -- putStrLn $ "Value of v2 == " ++ (show $ getvalue "v2" state4)
   -- putStrLn $ "Value of v3 == " ++ (show $ getvalue "v3" state4)
   -- putStrLn $ "State3 = " ++ (show state3)
  putStrLn $
     if  (1  == (getvalue  "v1"  state4 )  )
    && (25 == (getvalue  "v2"  state4 )  )
    && (9 == (getvalue  "v3"  state4 )  )
    then  "Unit␣ tests ␣hold"
     else  " Tests␣ failed "
```

As you can see from the source code above, the functions on State call functions on the Environment and the Store . Normally we do not need to relate directly to the environment or store or their support functions.

The Environment uses a simple association list of variable name–store indices. The functions are emptyenvironment for creating an empty environment, Haskell's standard function lookup :: Eq a =>a -> [(a, b)] -> Maybe b for searching an association list, and addenv for adding a variable to the environment (error stop if already registered).

The Store uses the same array structure as before, but rather than setting aside 10 indices it will now grow as new variables are added. The functions are emptystore for creating an empty store, getstore for finding a value at a given index (error stop if index is out of range), setstore for setting a new value at a given index (error stop if index is out of range), and enlargestore for adding a new index and setting its value, returning the new index and the enlarged store.

At the bottom of the CalculatorState module you can see a unit test for the state API.

4.3 Task

The task is to implement an interpreter for the variable based calculator. Other than handling variable declaration, assignment (statements) and look up (expression), this interpreter will be very similar to the register based interpreters.

Also upgrade the unit tests and interactive calculator to deal with explicit variables.

An important difference from Pamphlet 3 is that the AST examples for running the interpreter, both for CalcExprAST and CalcStmtAST , need to declare variables before they can be used. Previously all registers were initalised to zero. Now we cannot access a variable that has not been declared.