PRÁCTICA 1

Aexp.hs

```haskell
module Aexp where

-- |-----------------------------------------------------------------------
-- | Exercise 1 - Abstract Syntax and Semantics of Aexp
-- |-----------------------------------------------------------------------
-- | Define the algebraic data type 'Aexp' for representing arithmetic
-- | expressions.

type VarId = String

data  Aexp  =  NumLit Integer
            | Var VarId
            | Add Aexp Aexp
            | Mul Aexp Aexp
            | Sub Aexp Aexp
            deriving Show

type Z = Integer
type State = VarId -> Z

-- | Define the function 'aval' that computes the value of an arithmetic
-- | expression in a given state.

aVal :: Aexp -> State -> Z
aVal (NumLit n) s = read n
aVal (Var a) s = s a
aVal (Add x y) s = aVal x s + aVal y s
aVal (Mul x y) s = aVal x s * aVal y s
aVal (Sub x y) s = aVal x s - aVal y s



-- |-----------------------------------------------------------------------
-- | Exercise 2 - Free variables of expressions
-- |-----------------------------------------------------------------------
-- | Define the function 'fvAexp' that computes the set of free variables
-- | occurring in an arithmetic expression. Ensure that each free variable
-- | occurs only once in the resulting list.

fvAexp :: Aexp -> [VarId]
fvAexp (NumLit n) = []
fvAexp (Var a) = [a]
fvAexp (Add x y) = fvAexp x ++ fvAexp y
fvAexp (Mul x y) = fvAexp x ++ fvAexp y
fvAexp (Sub x y) = fvAexp x ++ fvAexp y

-- |-----------------------------------------------------------------------
-- | Exercise 3 - Substitution of variables in expressions
-- |-----------------------------------------------------------------------
-- | Define the algebraic data type 'Subst' for representing substitutions.

data Subst = VarId :->: Aexp

-- | Define a function 'substAexp' that takes an arithmetic expression
-- | 'a' and a substitution 'y -> a0' and returns the substitution 'a [y -> a0]';
-- | i.e., replaces every occurrence of 'y' in 'a' by 'a0'.

substAexp :: Aexp -> Subst -> Aexp
substAexp (NumLit n) _ = NumLit n
substAexp (Var a) (y:->:a0) = if a==y then VarId a0 else a
substAexp (Add a1 a2) (y:->:a0) = Add (substAexp a1 (y:->:a0)) (substAexp a2 (y:->:a0))
substAexp (Mul a1 a2) (y:->:a0) = Mul (substAexp a1 (y:->:a0)) (substAexp a2 (y:->:a0))
substAexp (Sub a1 a2) (y:->:a0) = Sub (substAexp a1 (y:->:a0)) (substAexp a2 (y:->:a0))
```

```haskell
-- |----------------------------------------------------------------------
-- | Exercise 4 - Update of state
-- |----------------------------------------------------------------------
-- | Define the algebraic data type 'Update' for representing state updates.

data Update = VarId :=>: Z

-- | Define a function 'update' that takes a state 's' and an update 'x -> v'
-- | and returns the updated state 's [x -> v]'

update :: State -> Update -> State
update s (y:=>a) x = if x==y then a else s x

-- | Define a function 'updates' that takes a state 's' and a list of updates
-- | 'us' and returns the updated states resulting from applying the updates
-- | in 'us' from head to tail. For example:
-- |
-- |    updates s {x -> 1, y > 2, x -> 3}
-- |
-- | returns a state that binds 'x' to 3 (the most recent update for 'x').

updates :: State -> [Update] -> [State]
updates s (x:xs) = update s x ++ updates s xs

-- |----------------------------------------------------------------------
-- | Exercise 5 - Folding expressions
-- |----------------------------------------------------------------------
-- | Define a function 'foldAexp' to fold an arithmetic expression.

foldAexp :: (LitNum -> b) -> (VarId -> b) -> (b -> b -> b) -> (b -> b -> b) -> (b -> b -> b) -> Aexp -> b
foldAexp nl v add mul sub a = recAexp a
    where
        recAexp (NumLit n) = nl n
        recAexp (Var x) = v x
        recAexp (Add a1 a2) = add (recAexp a1) (recAexp a2)
        recAexp (Mul a1 a2) = mul (recAexp a1) (recAexp a2)
        recAexp (Sub a1 a2) = sub (recAexp a1) (recAexp a2)

-- | Use 'foldAexp' to define the functions 'aVal', 'fvAexp', and 'substAexp'.

aVal' :: Aexp -> State -> Z
aVal' a s = foldAexp nVal s (+) (*) (-) a

fvAexp' :: untyped
fvAexp' = undefined

substAexp' :: Aexp -> Subst -> Aexp
substAexp' a (x:->:a0) = foldAexp NumLit subsVar Add Mul Sub a
    where
        subsVar y = if y==x then a0 else Var y
```

PRÁCTICA 2

NaturalSemantics.hs

```haskell
module NaturalSemantics where

import           Aexp
import           Bexp
import           State
import           While

-- representation of configurations for WHILE

data Config = Inter Stm State  -- <S, s>
            | Final State      -- s

-- representation of the execution judgement <S, s> -> s'

update :: Var -> Aexp -> State -> State
update x a s = (\v -> if v==x then aVal a s else s v)

nsStm :: Config -> Config

-- x := a

nsStm (Inter (Ass x a) s)      = Final s'
  where s' = update x a s

-- skip

nsStm (Inter Skip s)           = Final s

-- s1; s2

nsStm (Inter (Comp ss1 ss2) s) = Final s''
  where
    Final s' = nsStm (Inter ss1 s)
    Final s'' = nsStm (Inter ss2 s')

-- if b then s1 else s2

-- B[b]s = tt
nsStm (Inter (If b ss1 ss2) s)
  | bVal b s = Final s'
    where Final s' = nsStm (Inter ss1 s)

-- B[b]s = ff
nsStm (Inter (If b ss1 ss2) s)
  | not (bVal b s) = Final s'
    where Final s' = nsStm (Inter ss2 s)

-- while b do s

-- B[b]s = ff
nsStm (Inter (While b ss) s)
  | not (bVal b s) = Final s

-- B[b]s = tt
nsStm (Inter (While b ss) s)
  | bVal b s = Final s''
    where
      Final s' = nsStm (Inter ss s)
      Final s'' = nsStm (Inter (While b ss) s')

-- repeat S until b

nsStm (Inter (Repeat ss b) s)
  | bVal b s' = Final s'
    where
      Final s' = nsStm (Inter ss s)

nsStm (Inter (Repeat ss b) s)
  | not (bVal b s') = Final s''
    where
      Final s' = nsStm (Inter ss s)
      Final s'' = nsStm (Inter (Repeat ss b) s')
```

```haskell
-- for

nsStm (Inter (For x a1 a2 ss) s)
  | aVal a1 s <= aVal a2 s = Final s'''
    where
      Final s' = nsStm (Inter (Ass x a1) s)
      Final s'' = nsStm (Inter ss s')
      Final s''' = nsStm (Inter (For x (Add v1 (N "1")) v2 ss) s'')
      v1 = N ( show (aVal a1 s))
      v2 = N ( show (aVal a2 s))

nsStm (Inter (For x a1 a2 ss) s)
  | not (aVal a1 s <= aVal a2 s )  = Final s


-- semantic function for natural semantics
sNs :: Stm -> State -> State
sNs ss s = s'
  where Final s' = nsStm (Inter ss s)
```

Exercises02.hs

```haskell
module Exercises02 where

import           Aexp
import           Bexp
import           NaturalSemantics
import           State
import           While
import           WhileExamples
import           WhileParser

-- |----------------------------------------------------------------------
-- | Exercise 1
-- |----------------------------------------------------------------------
-- | The function 'sNs' returns the final state of the execution of a
-- | WHILE statement 'st' from a given initial state 's'. For example:

execFactorial :: State
execFactorial = sNs factorial factorialInit

-- | returns the final state:
-- |
-- |    s x = 1
-- |    s y = 6
-- |    s _ = 0
-- |
-- | Since a state is a function it cannot be printed thus you cannot
-- | add 'deriving Show' to the algebraic data type 'Config'.
-- | The goal of this exercise is to define a function to "show" a state
-- | thus you can inspect the final state yielded by the natural semantics
-- | of WHILE.

-- | Exercise 1.1
-- | Define a function 'showState' that given a state 's' and a list
-- | of variables 'vs' returns a list of strings showing the bindings
-- | of the variables mentioned in 'vs'. For example, for the state
-- | 's' above we get:
-- |
-- |    showState s ["x"] = ["x -> 1"]
-- |    showState s ["y"] = ["y -> 6"]
-- |    showState s ["x", "y"] = ["x -> 1", "y -> 6"]
-- |    showState s ["y", "z", "x"] = ["y -> 6", "z -> 0", "x -> 1"]

showState :: State -> [Var] -> [String]
showState s [] = []
showState s (x:xs) = ( x ++ "->" ++ show(s x)) : (showState s xs)


-- | Using the function 'sNs' to execute a WHILE program is handy but a bit awkward.
-- | The WHILE statement must be provided in abstract syntax and the initial
-- | state must be explicitly given and inspected.
-- |
-- | The 'run' function allows to execute a WHILE program stored in a file
-- | in concrete syntax and reports the final value of the variables mentioned
-- | in the program header. For example:
-- |
-- |    > run "Examples/Factorial.w"
-- |    Program Factorial finalized.
-- |    Final State: ["x->0","y->24"]

-- | Run the WHILE program stored in filename and show final values of variables
run :: FilePath -> IO()
run filename =
  do
     (programName, vars, stm) <- parser filename
     let Final s = nsStm (Inter stm (const 0))
     putStrLn $ "Program " ++ programName ++ " finalized."
     putStr "Final State: "
     print $ showState s vars

-- | Exercise 1.2
-- | Use the function 'run' to execute the WHILE programs 'Factorial.w' and 'Divide.w'
-- | in the directory 'Examples' to check your implementation of the Natural Semantics.
-- | Write a few more WHILE programs. For example, write a WHILE program
-- | "Power.w" to compute x^y.
```

```
-- |----------------------------------------------------------------------
-- | Exercise 2
-- |----------------------------------------------------------------------
-- | The WHILE language can be extended with a 'repeat S until b' statement.
-- | The file Examples/FactorialRepeat.w contains a simple program to
-- | compute the factorial with a 'repeat until' loop.

-- | Exercise 2.1
-- | Define the natural semantics of this new statement. You are not allowed
-- | to rely on the 'while b do S' statement.

{- Formal definition of 'repeat S until b'


                         < S, s > -> s'
  [repeat tt] -------------------------------------------    if bVal[b]s' = tt
                    < repeat S b , s > -> s'


               < S, s > -> s'   < repeat S b , s' > -> s''
  [repeat ff] -----------------------------------------------    if bVal[b]s' = ff
                     < repeat S b , s > -> s''

-}

-- | Exercise 2.2
-- | Extend the definition of 'nsStm' in module NaturalSemantics.hs
-- | to include the 'repeat S until b' statement.


-- | Exercise 2.3
-- | Write a couple of WHILE programs that use the 'repeat' statement.
-- | Use 'run' to test your programs.


-- |----------------------------------------------------------------------
-- | Exercise 3
-- |----------------------------------------------------------------------
-- | The WHILE language can be extended with a 'for x:= a1 to a2 do S'
-- | statement.
-- | The file Examples/FactorialFor.w contains a simple program to compute
-- | the factorial with a 'for' loop.
-- | The file Examples/ForTests.w contains a more contrived example illustrating
-- | some subtle points of the semantics of the for loop.

-- | Exercise 3.1
-- | Define the natural semantics of this new statement. You are not allowed
-- | to rely on the 'while b do S' or the 'repeat S until b' statements.

{- Formal definition of 'for x:= a1 to a2 do S'

          < x := a1 , S > -> s'  < S , s' > -> s''  < for x (v1 + 1) v2 S, s'' > -> s'''
[for tt] ------------------------------------------------------------------------       if aVal[a1]s <=
aVal[a2]s = tt
                       < for x a1 a2 S , s > -> s'''

          where
            v1 = N^-1[ aVal[a1]s ]
            v2 = N^-1[ aVal[a2]s ]


[for ff] -----------------------------------------------------------  if aVal[x]s <= aVal[a2]s = ff
                       < for x a1 a2 S , s > -> s

-}

-- | Exercise 3.2
-- | Extend  the definition 'nsStm' in module NaturalSemantics.hs
-- | to include the 'for x:= a1 to a2 do S' statement.

-- | Exercise 3.3
-- | Write a couple of WHILE programs that use the 'for' statement.
-- | Use 'run' to test your programs.
```

```haskell
-- |----------------------------------------------------------------------
-- | Exercise 4
-- |----------------------------------------------------------------------

-- | Define the semantics of arithmetic expressions (Aexp) by means of
-- | natural semantics. To that end, define an algebraic datatype 'ConfigAexp'
-- | to represent the configurations, and a function 'nsAexp' to represent
-- | the evaluation judgement.

-- representation of configurations for Aexp

data ConfigAExp = Redex Aexp State
                | Value Z

-- representation of the evaluation judgement <a, s> -> z

nsAexp :: ConfigAExp -> ConfigAExp
nsAexp (Redex (N n) s) = Value (read n)
nsAexp (Redex (V v) s) = Value (s v)
nsAexp (Redex (Add a1 a2) s) = Value (aVal a1 s + aVal a2 s)
nsAexp (Redex (Mult a1 a2) s) = Value (aVal a1 s * aVal a2 s)
nsAexp (Redex (Sub a1 a2) s) = Value (aVal a1 s - aVal a2 s)

-- | Test your function with a number of expressions and states.
s0 :: State
s0 "x" = 5
s0 "y" = 3
s0 _ = 0

expr1 :: Aexp
expr1 = Mult (Add (V "x") (N "3")) (Sub (V "y") (N "2"))

Value z1 = nsAexp (Redex expr1 s0) -- z1 = 8
```

PRÁCTICA 3

StructuralSemantics.hs

```haskell
module StructuralSemantics where

import          Aexp
import          Bexp
import          State
import          While

-- representation of configurations for WHILE

data Config = Inter Stm State  -- <S, s>
            | Final State      -- s
            | Stuck Stm State  -- <S, s>

isFinal :: Config -> Bool
isFinal (Final _) = True
isFinal _         = False

isInter :: Config -> Bool
isInter (Inter _ _) = True
isInter _           = False

isStuck :: Config -> Bool
isStuck (Stuck _ _) = True
isStuck _           = False

-- representation of the transition relation <S, s> => gamma

update :: Var -> Aexp -> State -> State
update x a s = (\v -> if v==x then (aVal a s) else s v)

sosStm :: Config -> Config

-- x := a
sosStm (Inter (Ass x a) s) = Final (update x a s)

-- skip
sosStm (Inter Skip s) = Final s

-- s1; s2
sosStm (Inter (Comp ss1 ss2) s)
    | isFinal s1' = Inter ss2 s'
    where
        s1' = sosStm (Inter ss1 s)
        Final s' = s1'

sosStm (Inter (Comp ss1 ss2) s)
    | isInter s1' = Inter (Comp ss1' ss2) s'
    where
        s1' = sosStm (Inter ss1 s)
        Inter ss1' s' = s1'

sosStm (Inter (Comp ss1 ss2) s)
    | isStuck s1' = Stuck ss1' s'
        where
            s1' = sosStm (Inter ss1 s)
            Stuck ss1' s' = s1'

-- if b then s1 else s2
sosStm (Inter (If b ss1 ss2) s)
    | bVal b s = Inter ss1 s

sosStm (Inter (If b ss1 ss2) s)
    | not(bVal b s) = Inter ss2 s

-- while b do
sosStm (Inter (While b ss) s) = Inter (If b s1 Skip) s
    where
        s1 = Comp ss (While b ss)

-- repeat s until b
sosStm (Inter (Repeat ss b) s) = Inter (Comp ss s2) s
    where s2 = If b Skip (Repeat ss b)
```

```haskell
-- for x a1 to a2 s
sosStm (Inter (For x a1 a2 ss) s) = Inter (Comp ass iff) s
    where
        ass = Ass x a1
        iff = If (Leq a1 a2) ss_for Skip
        ss_for = Comp ss (For x (Add v1 (N "1")) v2 ss)
        v1= N (show (aVal a1 s))
        v2= N (show (aVal a2 s))

-- abort
sosStm (Inter Abort s) = Stuck Abort s
```

Exercises03.hs

```haskell
module Exercises03 where

import           Aexp
import           Bexp
import           State
import           StructuralSemantics
import           While
import           WhileExamples
import           WhileParser

-- |----------------------------------------------------------------------
-- | Exercise 1
-- |----------------------------------------------------------------------

-- | Given the type synonym 'DerivSeq' to represent derivation sequences
-- | of the structural operational semantics:

type DerivSeq = [Config]

-- | Define a function 'derivSeq' that given a WHILE statement 'st' and an
-- | initial state 's' returns the corresponding derivation sequence:

derivSeq :: Stm -> State -> DerivSeq
derivSeq ss s = derivSeq' (Inter ss s)
  where
    derivSeq' (Final s) = [Final s]
    derivSeq' (Inter ss s) = (Inter ss s) : derivSeq' (sosStm (Inter ss s))

-- | The function 'showDerivSeq' returns a String representation  of
-- | a derivation sequence 'dseq'. The 'vars' argument is a list of variables
-- | that holds the variables to be shown in the state:

showDerivSeq :: [Var] -> DerivSeq -> String
showDerivSeq vars dseq = unlines (map showConfig dseq)
  where
    showConfig (Final s) = "Final state:\n" ++ unlines (showVars s vars)
    showConfig (Stuck stm s) = "Stuck state:\n" ++ show stm ++ "\n" ++ unlines (showVars s vars)
    showConfig (Inter ss s) = show ss ++ "\n" ++ unlines (showVars s vars)
    showVars s vs = map (showVal s) vs
    showVal s x = " s(" ++ x ++ ")= " ++ show (s x)

-- | Use the function 'run' below to execute the WHILE programs 'Divide.w'
-- | and 'Factorial.w' in the directory 'Examples' to check your implementation
-- | of the Structural Semantics. For example:
-- |
-- |  > run "Examples/Factorial.w"
-- |
-- | Write a few more WHILE programs. For example, write a WHILE program to
-- | compute x^y.

-- | Run the WHILE program stored in filename and show final values of variables

run :: FilePath -> IO()
run filename =
  do
     (_, vars, stm) <- parser filename
     let  dseq = derivSeq stm (const 0)
     putStr $ showDerivSeq vars dseq

-- | The function 'sSoS' below is the semantic function of the
-- | structural operational semantics of WHILE. Given a WHILE statement 'st'
-- | and an initial state 's' returns the final configuration of the
-- | corresponding derivation sequence:

sSos :: Stm -> State -> State
sSos ss s = s'
  where Final s' = last (derivSeq ss s)
```

```haskell
-- |----------------------------------------------------------------------
-- | Exercise 2
-- |----------------------------------------------------------------------
-- | The WHILE language can be extended with a 'repeat S until b' statement.

-- | Exercise 2.1
-- | Define the structural operational semantics of this new statement. You
-- | are not allowed to rely on the 'while b do S' statement.

{- Formal definition of 'repeat S until b'


[repeat sos] < repeat S until b , s > => < S ; If b then Skip else (Repeat S until b), s>

-}


-- |----------------------------------------------------------------------
-- | Exercise 3
-- |----------------------------------------------------------------------

-- | The WHILE language can be extended with a 'for x:= a1 to a2' statement.

-- | Exercise 3.1
-- | Define the structural operational semantics of this new statement. You
-- | are not allowed to rely on the 'while b do s' statement.

{- Formal definition of 'for x:= a1 to a2'

[for sos] < for x:=a1 to a2 do S, s > => < x:=a1 ; If (a1 <= a2) then (S ; for x:=v1+1 to v2 do S) else skip, s
>

where
        v1 = N^-1 (A[a1]s)
        v2 = N^-1 (A[a2]s)

-}


-- |----------------------------------------------------------------------
-- | Exercise 5
-- |----------------------------------------------------------------------

-- | Implement in Haskell the Structural Operational Semantics for the
-- | evaluation of arithmetic expressions Aexp.

{-
   Structural Operational Semantics for the left-to-right evaluation of
   arithmetic expressions:

   A configuration is either intermediate <Aexp, State> or final Z.

   Note we are abusing notation and write 'n' for both a literal (syntax)
   and an integer (semantics). The same goes for arithmetic operators (+,-,*).

   [N]  ----------------
         < n, s > => n

   [V] ------------------------
        < x, s > => < s x, s >

   [+] -----------------------------  where n3 = n1 + n2
        < n1 + n2, s > => < n3, s >

            < a2, s > => < a2', s >
   [+] ----------------------------------
        < n1 + a2, s > => < n1 + a2', s >

            < a1, s > => < a1', s >
   [+] -----------------------------------
        < a1 + a2, s > => < a1' + a2, s >

   The rules for * and - are similar.

-}

-- | We use the algebraic data type 'AexpConfig' to represent the
-- | configuration of the transition system

data AexpConfig = Redex Aexp State  -- a redex is a reducible expression
                | Value Z           -- a value is not reducible; it is in normal form
```

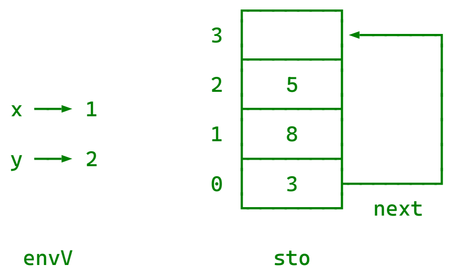
```haskell
-- |----------------------------------------------------------------------
-- | Exercise 5.1
-- |----------------------------------------------------------------------

-- | Define a function 'sosAexp' that given a configuration 'AexpConfig'
-- | evaluates the expression in 'AexpConfig' one step and returns the
-- | next configuration.

sosAexp :: AexpConfig -> AexpConfig

sosAexp (Redex (N n) s) = Value n
sosAexp (Redex (V x) s) = Redex (N (s x)) s

sosAexp (Redex (Add (N n1) (N n2)) s) = Redex (N n3) s
  where n3 =  n1 + n2
sosAexp (Redex (Add (N n1) a2) s) = Redex (Add (N n1) a2') s'
  where Redex a2' s' = sosAexp (Redex a2 s)
sosAexp (Redex (Add a1 a2) s) = Redex (Add a1' a2) s'
  where Redex a1' s' = sosAexp (Redex a1 s)

sosAexp (Redex (Mult (N n1) (N n2)) s) = Redex (N n3) s
  where n3 =  n1 *  n2
sosAexp (Redex (Mult (N n1) a2) s) = Redex (Mult (N n1) a2') s'
  where Redex a2' s' = sosAexp (Redex a2 s)
sosAexp (Redex (Mult a1 a2) s) = Redex (Mult a1' a2) s'
  where Redex a1' s' = sosAexp (Redex a1 s)

sosAexp (Redex (Sub (N n1) (N n2)) s) = Redex (N n3) s
  where n3 =  n1 -  n2
sosAexp (Redex (Sub (N n1) a2) s) = Redex (Sub (N n1) a2') s'
  where Redex a2' s' = sosAexp (Redex a2 s)
sosAexp (Redex (Sub a1 a2) s) = Redex (Sub a1' a2) s'
  where Redex a1' s' = sosAexp (Redex a1 s)


-- |----------------------------------------------------------------------
-- | Exercise 5.2
-- |----------------------------------------------------------------------

-- | Given the type synonym 'AexpDerivSeq' to represent derivation sequences
-- | of the structural operational semantics for arithmetic expressions 'Aexp':

type AexpDerivSeq = [AexpConfig]

-- | Define a function 'aExpDerivSeq' that given a 'Aexp' expression 'a' and an
-- | initial state 's' returns the corresponding derivation sequence:

isValue :: AexpConfig -> Bool
isValue (Value _) = True
isValue _ = False

isRedex :: AexpConfig -> Bool
isRedex (Redex _ _) = True
isRedex _ = False

aExpDerivSeq :: Aexp -> State -> AexpDerivSeq
aExpDerivSeq a s
  | isValue next = [Redex a s] ++ [Value n]
    where
      next = sosAexp (Redex a s)
      Value n = next

aExpDerivSeq exp s
  | isRedex next = [Redex exp s] ++ (aExpDerivSeq exp' s')
    where
      next = sosAexp (Redex exp s)
      Redex exp' s' = next

-- | To test your code, you can use the function 'showAexpDerivSeq' that
-- | returns a String representation  of a derivation sequence 'dseq':

showAexpDerivSeq :: [Var] -> AexpDerivSeq -> String
showAexpDerivSeq vars dseq = unlines (map showConfig dseq)
  where
    showConfig (Value n)    = "Final value:\n" ++ show n
    showConfig (Redex ss s) = show ss ++ "\n" ++ unlines (map (showVal s) vars)
    showVal s x = " s(" ++ x ++ ")= " ++ show (s x)

-- | Therefore, you can print the derivation sequence of an 'Aexp' with:

exp1 :: Aexp
exp1 = ( (V "x") `Add` (V "y") ) `Add` (V "z")
```

```haskell
exp2 :: Aexp
exp2 =  (V "x") `Add` ( (V "y") `Add` (V "z") )

exp3 :: Aexp
exp3 = Mult (V "x") (Add (V "y") (Sub (V "z") (N "1")))

sExp :: State
sExp "x" = 1
sExp "y" = 2
sExp "z" = 3
sExp  _  = 0

showAexpSeq :: Aexp -> State -> IO()
showAexpSeq a s = putStrLn $ showAexpDerivSeq ["x", "y", "z"] (aExpDerivSeq a s)

-- | Test you code printing derivation sequences for the expressions above as follows:

showExp1 :: IO ()
showExp1 = showAexpSeq exp1 sExp

-- | Convert concrete syntax to abstract syntax

concreteToAbstract :: FilePath -> FilePath -> IO()
concreteToAbstract inputFile outputFile =
  do
    (_, _, stm) <- parser inputFile
    let s = show stm               -- | have 'show' replaced by a pretty printer
    if null outputFile
      then putStrLn s
      else writeFile outputFile s
```

PRÁCTICA 4

NaturalSemantics.hs

```haskell
module NaturalSemantics where

import           Aexp
import           Bexp
import           State
import           While

----------------------------------------------------------------------
-- Variable Declarations
----------------------------------------------------------------------

-- locations
type Loc = Z

-- variable environment
type EnvVar = Var -> Loc

-- store
type Store = Loc -> Z

-- the register 'next' is actually stored at location 0 of the store:
-- 'sto next' refers to the first available cell in the store 'sto'

next :: Loc
next = 0

-- rudimentary stack-based memory allocation
new :: Loc -> Loc
new l = l + 1

{-

   After processing the local variable declarations:

     var x:= 8;
     var y:= 5;

   we get the envV and sto shown below:
```




```haskell
-}

-- | Exercise 1.1 - update envV and sto

-- update a variable environment with a new binding envV [x -> l]
updateV :: EnvVar -> Var -> Loc -> EnvVar
updateV envV x l = (\y -> if y == x then l else envV y)
-- ahora actualiza el entorno: en la variable x mete la localización nueva l

-- update a store with a new binding sto [l -> v]
updateS :: Store -> Loc -> Z -> Store
updateS sto l v = (\p -> if p == l then v else sto p)
-- ahora actualiza el store: en la posición l mete a v

-- | Exercise 1.2 - natural semantics for variable declarations

-- variable declaration configurations

data ConfigD = InterD DecVar EnvVar Store  -- <Dv, envV, store>
             | FinalD EnvVar Store         -- <envV, store>
```

```haskell
-- pilla un Dv (instrucciones con x:=a; Dv), un envV y un store
-- y te devuelve el envV y el store montao con cada variable en su sitio asignada

nsDecV :: ConfigD -> ConfigD
-- var x:= a
nsDecV (InterD (Dec x a decs) envV store) = nsDecV (InterD decs envV' store')
  where
    envV' = updateV envV x l
    store' = updateS (updateS store l v) next (new l)
    l = store next
    v = aVal a (store. envV)

-- epsilon
nsDecV (InterD EndDec envV store)        = FinalD envV store

----------------------------------------------------------------------
-- Procedure Declarations
----------------------------------------------------------------------

-- procedure environment (note this environment is not a function)

--                    p    s    snapshots    previous
--                    |    |     /     \        |
data EnvProc = EnvP Pname Stm EnvVar EnvProc EnvProc
             | EmptyEnvProc

-- | Exercise 2.1 - update envP
{-
data DecProc = Proc Pname Stm DecProc
             | EndProc
             deriving Show
-}

--  pilla la declaracion de procedimientos, el envV y el envP y te devuelve el nuevo envP con los nuevos
procedimientos añadidos
-- cuando mete uno nuevo lo unico que cambia es el envP

-- update the procedure environment envP
updP :: DecProc -> EnvVar -> EnvProc -> EnvProc
updP (Proc p s procs) envV envP = updP procs envV (EnvP p s envV envP envP)
updP EndProc envV envP          = envP

-- | Exercise 2.2 - look up procedure definitions

-- lookup procedure p
envProc :: EnvProc -> Pname -> (Stm, EnvVar, EnvProc)
envProc (EnvP q s envV envP envs) p = if p == q then (s, envV, envP) else envProc envs p
envProc EmptyEnvProc p              = error $ "undefined procedure" ++ p

----------------------------------------------------------------------
-- Natural Semantics for WHILE
----------------------------------------------------------------------

-- representation of configurations for WHILE

data Config = Inter Stm Store  -- <S, sto>
            | Final Store      -- sto

-- representation of the transition relation envV, envP |- <S, sto> -> sto'


nsStm :: EnvVar -> EnvProc -> Config -> Config

-- | Exercise 3.1

nsStm envV envP (Inter (Ass x a) sto) = Final sto'
  where
    sto' = updateS sto l z
    l = envV x                           -- envV x devuelve la localización de x en store
    z = aVal a (sto . envV)              -- (sto . envV) = s

nsStm envV envP (Inter Skip sto) = Final sto

nsStm envV envP (Inter (Comp ss1 ss2) sto) = Final sto''
  where
    Final sto' = nsStm envV envP (Inter ss1 sto)
    Final sto'' = nsStm envV envP (Inter ss2 sto')

nsStm envV envP (Inter (If b ss1 ss2) sto)
  | bVal b (sto . envV) = Final sto'
    where Final sto' = nsStm envV envP (Inter ss1 sto)
```

```haskell
nsStm envV envP (Inter (If b ss1 ss2) sto)
  | not( bVal b (sto . envV) ) = Final sto'
    where Final sto' = nsStm envV envP (Inter ss2 sto )


nsStm envV envP (Inter (While b ss) sto)
  | bVal b (sto . envV) = Final sto''
  where
    Final sto' = nsStm envV envP (Inter ss sto)
    Final sto'' = nsStm envV envP (Inter (While b ss) sto')

nsStm envV envP (Inter (While b ss) sto)
  | not(bVal b (sto . envV)) = Final sto


-- | Block DecVar DecProc Stm

-- nsDecV pilla todas las variables y monta el envV y store
-- updP pilla la declaracion de procedimientos, el envV y el envP y te devuelve el nuevo envP con los nuevos
procedimientos añadidos
nsStm envV envP (Inter (Block decVar decProc ss) sto) = Final sto''
  where
    FinalD envV' sto' = nsDecV (InterD decVar envV sto)
    envP' = updP decProc envV' envP
    Final sto'' = nsStm envV' envP' (Inter ss sto')

-- Call Pname       sin recursividad
-- envProc :: EnvProc -> Pname -> (Stm, EnvVar, EnvProc)
nsStm envV envP (Inter (Call p) sto) = Final sto'
  where
    (ss, envV', envP') = envProc envP p
    Final sto' = nsStm envV' envP' (Inter ss sto)

-- Call Pname       con recursividad
nsStm envV envP (Inter (Call p) sto) = Final sto'
  where
    (ss, envV', envP') = envProc envP p
    envP'' = EnvP p ss envV' envP' envP'
    Final sto' = nsStm envV' envP'' (Inter ss sto)

-- semantic function for Natural Semantics
sNs :: Stm -> Store -> Store
sNs s sto = sto'
   where
     Final sto' = nsStm initEnvV EmptyEnvProc (Inter s sto)
     initEnvV :: EnvVar
     initEnvV x = error $ "undefined variable " ++ x
```

Exercises04.hs

```haskell
module Exercises04 where

import           Aexp
import           NaturalSemantics
import           State
import           While
import           WhileParser

-- |----------------------------------------------------------------------
-- | Exercise 1 - Local Variable Declarations
-- |----------------------------------------------------------------------

-- | The code below tests your definitions.

-- | First, we initialize the variable environment and the store:

-- note that global variables are not allowed in WHILE
initEnvV :: EnvVar
initEnvV x = error $ "undefined variable " ++ x

-- note that accessing a non-allocated location yields an error
initStore :: Store
initStore l
  | l == next = 1
  | otherwise = error $ "undefined location " ++ show l

-- | Then, we define some variable declarations:

declarations :: DecVar
declarations =  Dec "x" (N "5")                    -- var x:= 5;
               (Dec "y" (N "2")                    -- var y:= 2;
               (Dec "z" (Mult (V "x") (V "y"))     -- var z:= x * y;
               (Dec "x" (Add (V "x") (N "1"))      -- var x:= 1;
                EndDec)))

-- | and a function 'showDecV' that shows the variables declared in a
-- | 'DecVar'. For each variable 'v' in the list 'vars', it shows
-- | both its location and value:

showDecV :: DecVar -> EnvVar -> Store -> [Var] -> String
showDecV decs env sto vars = foldr (showVar env' sto') [] vars
  where
     FinalD env' sto' = nsDecV (InterD decs env sto)
     showVar env sto x s = "var " ++ x ++ " loc " ++ show (env' x) ++ " val " ++ show (sto' . env' $ x) ++ "\n"
++ s

-- | Finally, we have a simple test for variable declarations:

testVarDec :: IO()
testVarDec = putStr $ showDecV declarations initEnvV initStore ["x", "y", "z"]

-- | and the expected output:

{-

> testVarDec
var x loc 4 val 6  -- note that the first declaration of x is shadowed
var y loc 2 val 2
var z loc 3 val 10

-}

-- |----------------------------------------------------------------------
-- | Exercise 2 - Procedure Declarations
-- |----------------------------------------------------------------------

-- | The code below tests your definitions.

-- | First, we initialize the procedure environment:

initEnvP :: EnvProc
initEnvP = EmptyEnvProc

-- | Then, we define some procedure declarations:

procedures :: DecProc
procedures =  Proc "p" Skip      -- proc p is skip;
             (Proc "q" Skip      -- proc q is skip;
```

```haskell
             (Proc "r" Skip      -- proc r is skip;
              EndProc))

-- | and the function 'showDecP' that shows the procedures declared in
-- | a 'DecProc'. For each procedure 'p', it shows the other procedures
-- | it knows (i.e. those that can be invoked from 'p').

showDecP :: DecProc -> String
showDecP procs =
   showDecP' $ updP procs undefined initEnvP
   where
      showDecP' EmptyEnvProc = ""
      showDecP' (EnvP p s envV envP envP') = p ++ " knows " ++ knows envP ++ "\n" ++ showDecP' envP'
      knows EmptyEnvProc               = ""
      knows (EnvP p s envV envP envP') = p ++ " " ++ knows envP'
--      showDecP' (EnvP p s envV envP envP') = p ++ "\n" ++ showDecP' envP'

-- | Finally, we have a simple test for procedure declarations:

testProcDec :: IO()
testProcDec = putStr $ showDecP procedures

-- | and the expected output:

{-

> testProcDec
r knows q p
q knows p
p knows

-}

-- |----------------------------------------------------------------------
-- | Exercise 3 - Natural Semantics for While
-- |----------------------------------------------------------------------

-- | The code below tests your definitions.

-- | The function 'showStore' shows the contents of a 'Store' (i.e. a
-- | memory dump). Recall that variable names are missing, but you can
-- | relate memory cells to variables by numbering the variables from 1.

showStore :: Store -> [(Integer, Integer)]
showStore sto = [ (l, v) | l <- [0..sto next - 1], let v = sto l ]

-- | Use the function 'run' below to execute the While programs 'CallTest.w'
-- | and 'RecursiveFactorial.w' in the directory 'Examples' to check your implementation
-- | of the Natural Semantics. For example:
-- |
-- |  > run "Examples/CallTest.w"

-- | Run the While program stored in filename and show the final content of the store
run :: FilePath -> IO()
run filename =
  do
     (program, _, stm) <- parser filename
     let Final store = nsStm emptyEnvV EmptyEnvProc (Inter stm emptyStore)
     putStrLn $ "Program " ++ program ++ " finalized."
     putStr "Memory dump: "
     print $ showStore store
  where
      emptyEnvV x = error $ "undefined variable " ++ x
      emptyStore l
         | l == next = 1
         | otherwise = error $ "undefined location " ++ show l
```

EXAMEN FEBRERO

AexpSOS.hs

```haskell
module AexpSOS where

-- En este fichero solo necesitas completar:
--
--   - 1.a la definición semántica de Aexp
--   - 1.b la implementación de la semántica de Aexp
--   - 1.c la implementación de la secuencia de derivación
--
-- No modifiques el resto del código. Puedes probar
-- tu implementación con la función eval, definida al final.

import           While21

-- |----------------------------------------------------------------------
-- | Exercise 1.a
-- |----------------------------------------------------------------------

-- | Define the Structural Operational Semantics of Aexp extended with
--   integer division.

{-

    Completa la definición semántica de Aexp detallando reglas y axiomas
    con judgements de la forma <a,s> = > <a',s>  y  <a,s> => n.

   [Integer] ___________________________
                   < n, s > => n


   [Var] __________________________
            < v , s > => s v


?? se pone asi lo de N^-1? pq tiene q ser objeto de sintaxis

   [Suma 1] _______________________________   where n3 = N[n1] + N[n2]
              < n1 + n2, s > => N^-1[n3]

                    < a2, s > => < a2', s >
   [Suma 2] ______________________________________
              < n1 + a2 , s > => < n1 + a2' , s >

                      < a1, s > => < a1', s >
   [Suma 3] ______________________________________
              < a1 + a2 , s > => < a1' + a2 , s >

   [Div 1] _______________________________   where n3 = N[n1] / N[n2]
              < n1 / n2, s > => N^-1[n3]

                    < a2, s > => < a2', s >
   [Div 2] ______________________________________
              < n1 / a2 , s > => < n1 / a2' , s >

                      < a1, s > => < a1', s >
   [Div 3] ______________________________________
              < a1 / a2 , s > => < a1' / a2 , s >

-}

-- |----------------------------------------------------------------------
-- | Exercise 1.b
-- |----------------------------------------------------------------------

-- | Implement the Structural Operational Semantics for the
-- | evaluation of arithmetic expressions Aexp.

-- | Use the algebraic data type 'AexpConfig' to represent the
-- | configuration of the transition system

data AexpConfig = Redex Aexp State  -- a redex is a reducible expression
                | Stuck Aexp State  -- a stuck is neither reducible nor a value
                | Value Z           -- a value is not reducible; it is in normal form
```

```haskell
-- | Define a function 'sosAexp' that given a configuration 'AexpConfig'
-- | evaluates the expression in 'AexpConfig' one step and returns the
-- | next configuration.
sosAexp :: AexpConfig -> AexpConfig

sosAexp (Redex (N n) s) = Value n
sosAexp (Redex (V x) s) = Redex (N (s x)) s

sosAexp (Redex (Add (N n1) (N n2)) s) = Redex (N n3) s
  where n3 =  n1 + n2
sosAexp (Redex (Add (N n1) a2) s) = Redex (Add (N n1) a2') s'
  where Redex a2' s' = sosAexp (Redex a2 s)
sosAexp (Redex (Add a1 a2) s) = Redex (Add a1' a2) s'
  where Redex a1' s' = sosAexp (Redex a1 s)

sosAexp (Redex (Mult (N n1) (N n2)) s) = Redex (N n3) s
  where n3 =  n1 *  n2
sosAexp (Redex (Mult (N n1) a2) s) = Redex (Mult (N n1) a2') s'
  where Redex a2' s' = sosAexp (Redex a2 s)
sosAexp (Redex (Mult a1 a2) s) = Redex (Mult a1' a2) s'
  where Redex a1' s' = sosAexp (Redex a1 s)

sosAexp (Redex (Sub (N n1) (N n2)) s) = Redex (N n3) s
  where n3 =  n1 -  n2
sosAexp (Redex (Sub (N n1) a2) s) = Redex (Sub (N n1) a2') s'
  where Redex a2' s' = sosAexp (Redex a2 s)
sosAexp (Redex (Sub a1 a2) s) = Redex (Sub a1' a2) s'
  where Redex a1' s' = sosAexp (Redex a1 s)

sosAexp (Redex (Div (N n1) (N n2)) s)
  | n2 == 0 = Stuck (Div (N n1) (N n2) ) s
  | otherwise = Redex (N n3) s
  where n3 = n1 `div` n2

sosAexp (Redex (Div (N n1) a2) s) = Redex (Div (N n1) a2') s'
  where Redex a2' s' = sosAexp (Redex a2 s)
sosAexp (Redex (Div a1 a2) s) = Redex (Div a1' a2) s'
  where Redex a1' s' = sosAexp (Redex a1 s)
-- |----------------------------------------------------------------------
-- | Exercise 1.c
-- |----------------------------------------------------------------------

-- | Given the type synonym 'AexpDerivSeq' to represent derivation sequences
-- | of the structural operational semantics for arithmetic expressions 'Aexp':

type AexpDerivSeq = [AexpConfig]

-- | Define a recursive function 'aExpDerivSeq' that given a 'Aexp'
--   expression 'a' and an initial state 's' returns the corresponding
--   derivation sequence:

isValue :: AexpConfig -> Bool
isValue (Value _) = True
isValue _ = False

isStuck :: AexpConfig -> Bool
isStuck (Stuck _ _) = True
isStuck _ = False

isRedex :: AexpConfig -> Bool
isRedex (Redex _ _) = True
isRedex _ = False

aExpDerivSeq :: Aexp -> State -> AexpDerivSeq
aExpDerivSeq a s
  | isValue next = [Redex a s] ++ [Value n]
    where
      next = sosAexp (Redex a s)
      Value n = next

aExpDerivSeq exp s
  | isRedex next = [Redex exp s] ++ (aExpDerivSeq exp' s')
    where
      next = sosAexp (Redex exp s)
      Redex exp' s' = next

aExpDerivSeq exp s
  | isStuck next = [Redex exp s] ++ [Stuck exp' s']
    where
      next = sosAexp (Redex exp s)
      Stuck exp' s' = next
```

```haskell
----------------------------------------------------------------------
-- NO MODIFICAR EL CODIGO DE ABAJO
----------------------------------------------------------------------

eval :: Aexp -> State -> IO()
eval a s = putStrLn $ showAexpDerivSeq ["x", "y", "z"] (aExpDerivSeq a s)
  where
    showAexpDerivSeq vars dseq = unlines (map showConfig dseq)
      where
        showConfig (Value n) = "Final value:\n" ++ show n
        showConfig (Stuck e st) = "Stuck expression:\n" ++ show e ++ "\n" ++ unlines (map (showVal st) vars)
        showConfig (Redex e st) =  show e ++ "\n" ++ unlines (map (showVal st) vars)
        showVal st x = " s(" ++ x ++ ")= " ++ show (st x)

-- Tests

-- test your implementation with the examples below, for example:
-- eval exp1 sInit

sInit :: State
sInit "x" = 1
sInit "y" = 2
sInit "z" = 4
sInit  _  = 0


exp1 :: Aexp
exp1 = ( (V "x") `Add` (V "y") ) `Add` (V "z") -- (x + y) + z

exp2 :: Aexp
exp2 =  (V "x") `Add` ( (V "y") `Add` (V "z") ) -- x + (y + z)

exp3 :: Aexp
exp3 = Mult (V "x") (Add (V "y") (Sub (V "z") (N 1))) -- x * (y + (z - 1))

exp4 :: Aexp
exp4 = Mult (Add (V "x") (V "y")) (Sub (N 9) (V "z")) -- (x + y) * (9 - z)

exp5 :: Aexp
exp5 = Div (Mult (V "y") (V "z")) (Add (V "x") (N 1)) -- (y * z) / (x + 1)

exp6 :: Aexp
exp6 = Div (Mult (V "y") (V "z")) (Sub (V "x") (N 1)) -- (y * z) / (x - 1)
```

NaturalSemantics.hs

```haskell
-- ---------------------------------------------------------------------
-- Natural Semantics for WHILE 2021.
-- Examen de Lenguajes de Programación. UMA.
-- 1 de febrero de 2021
--
-- Apellidos, Nombre:
-- ---------------------------------------------------------------------

module NaturalSemantics where

-- En este fichero solo necesitas completar:
--
--   - 2.a la definición semántica de la sentencia case
--   - 2.a la implementación de la sentencia case
--
-- No modifiques el resto del código. Puedes probar
-- tu implementación con la función run, definida al final.

----------------------------------------------------------------------
-- NO MODIFICAR EL CODIGO DE ABAJO
----------------------------------------------------------------------

import           While21
import           While21Parser

updateState :: State -> Var -> Z -> State
updateState s x v y = if x == y then v else s y

-- representation of configurations for While

data Config = Inter Stm State  -- <S, s>
            | Final State      -- s

-- representation of the transition relation <S, s> -> s'

nsStm :: Config -> Config

-- x := a

nsStm (Inter (Ass x a) s) = Final (updateState s x (aVal a s))

-- skip

nsStm (Inter Skip s) = Final s

-- s1; s2

nsStm (Inter (Comp ss1 ss2) s) = Final s''
  where
    Final s'  = nsStm (Inter ss1 s)
    Final s'' = nsStm (Inter ss2 s')

-- if b then s1 else s2

nsStm (Inter (If b ss1 ss2) s)
  | bVal b s = Final s'
  where
    Final s' = nsStm (Inter ss1 s)

nsStm (Inter (If b ss1 ss2) s)
  | not(bVal b s) = Final s'
  where
    Final s' = nsStm (Inter ss2 s)

----------------------------------------------------------------------
-- NO MODIFICAR EL CODIGO DE ARRIBA
----------------------------------------------------------------------

-- case a of

-- |--------------------------------------------------------------------
-- | Exercise 2.a
-- |--------------------------------------------------------------------

-- | Define the Natural Semantics of the case statement.

{-

    Completa la definición semántica de la sentencia case.
```

```
  Regla 1: Si encontramos una etiqueta n_ij cuyo valor coincida con el de a, se ejecuta la sentencia S_i
correspondiente y se ignora el resto de casos.

                          <S, s> -> s'
  [Case1NStt]  ------------------------------------            if LC = (LL : S LC') and (A[a]s isElem LL) = tt
                     <Case a of LC end, s> -> s'

                       <case a of LC' end, s> -> s'
  [Case1NSff]  -----------------------------------------       if LC = (LL : S LC') and (A[a]s isElem LL) = ff
                       <Case a of LC end, s> -> s'

    Regla 2:  Si ninguna etiqueta n_ij coincide con el valor de a y al final aparece un caso default, se
ejecuta la sentencia S_d

                          < Sd, s> -> s_d
  [Case2NS] --------------------------------------------          if LC = default:Sd
                     < Case a of LC, s > -> s_d

    Regla 3:  Si ninguna etiqueta n_ij coincide con el valor de a y no aparece un caso default, se aborta la
ejecución del programa.

  [Case3NS] --------------------------------------                 if LC = End
                <Case a of LC end, s> -> error


-}

-- |----------------------------------------------------------------------
-- | Exercise 2.a
-- |----------------------------------------------------------------------

-- | Implement the Natural Semantics of the case statement.

nsStm (Inter (Case a (LabelledStm ll ss lc')) s)      -- cuidado con las mayusculas q peta
  | elem (aVal a s) ll = Final s'
  | otherwise = Final s''
    where
      Final s' = nsStm (Inter ss s)
      Final s'' = nsStm (Inter (Case a lc') s)


nsStm (Inter (Case a (Default ss)) s) = Final s'
  where Final s' = nsStm (Inter ss s)

nsStm (Inter (Case a EndLabelledStms) s) = error "no se ha encontrado coincidencia en ninguna lista"

----------------------------------------------------------------------
-- NO MODIFICAR EL CODIGO DE ABAJO
----------------------------------------------------------------------

-- | Run the While program stored in filename and show final values of variables.
--   For example: run "TestCase.w"

run :: String -> IO()
run filename =
  do
     (_, vars, stm) <- parser filename
     let Final s = nsStm (Inter stm (const 0))
     print $ showState s vars
     where
       showState s = map (\ v -> v ++ "->" ++ show (s v))
```

StructuralSemantics.hs

```haskell
module StructuralSemantics where

-- En este fichero solo necesitas completar:
--
--   - 2.b la definición semántica de la sentencia case
--   - 2.b la implementación de la sentencia case
--
-- No modifiques el resto del código. Puedes probar
-- tu implementación con la función run, definida al final.

----------------------------------------------------------------------
-- NO MODIFICAR EL CODIGO DE ABAJO
----------------------------------------------------------------------

import           Data.List.HT  (takeUntil)

import           While21
import           While21Parser

-- representation of configurations for While

data Config = Inter Stm State  -- <S, s>
            | Final State      -- s
            | Stuck Stm State  -- <S, s>

isFinal :: Config -> Bool
isFinal (Final _) = True
isFinal _         = False

isInter :: Config -> Bool
isInter (Inter _ _) = True
isInter _           = False

isStuck :: Config -> Bool
isStuck (Stuck _ _) = True
isStuck _           = False

-- representation of the transition relation <S, s> -> s'

sosStm :: Config -> Config

-- x := a

sosStm (Inter (Ass x a) s) = Final (update s x (aVal a s))
  where
    update s x v y = if x == y then v else s y

-- skip

sosStm (Inter Skip s) = Final s

-- s1; s2

sosStm (Inter (Comp ss1 ss2) s)
  | isFinal next = Inter ss2 s'
  where
    next = sosStm (Inter ss1 s)
    Final s' = next

sosStm (Inter (Comp ss1 ss2) s)
  | isStuck next = Stuck (Comp stm ss2) s'
  where
    next = sosStm (Inter ss1 s)
    Stuck stm s' = next

sosStm (Inter (Comp ss1 ss2) s)
  | isInter next = Inter (Comp ss1' ss2) s'
  where
    next = sosStm (Inter ss1 s)
    Inter ss1' s' = next

-- if b then s1 else s2

sosStm (Inter (If b ss1 ss2) s)
  | bVal b s = Inter ss1 s

sosStm (Inter (If b ss1 ss2) s)
  | not (bVal b s) = Inter ss2 s
```

```haskell
-- abort
sosStm (Inter Abort s) = Stuck Abort s


----------------------------------------------------------------------
-- NO MODIFICAR EL CODIGO DE ARRIBA
----------------------------------------------------------------------

-- case a of

-- |----------------------------------------------------------------------
-- | Exercise 2.b
-- |----------------------------------------------------------------------

-- | Define the Structural Operational Semantics of the case statement.

{-
    Completa la definición semántica de la sentencia case.

  Regla 1: Si encontramos una etiqueta n_ij cuyo valor coincida con el de a, se ejecuta la sentencia S_i correspondiente y se ignora el resto de casos.


  [Case1NStt]     ------------------------------------          if LC = (LL : S LC') and (A[a]s isElem LL) = tt
                     <Case a of LC end, s> => <S, s>


  [Case1NSff]  ------------------------------------------------ if LC = (LL : S LC') and (A[a]s isElem LL) = ff
               <Case a of LC end, s> => <case a of LC' end, s>

    Regla 2:  Si ninguna etiqueta n_ij coincide con el valor de a y al final aparece un caso default, se ejecuta la sentencia S_d


  [Case2NS] --------------------------------------------          if LC = default:Sd
                     < Case a of LC, s > => < Sd, s>

    Regla 3:  Si ninguna etiqueta n_ij coincide con el valor de a y no aparece un caso default, se aborta la ejecución del programa.

  [Case3NS] --------------------------------------                 if LC = End
                <Case a of LC end, s> => < abort, s >

-}

-- |----------------------------------------------------------------------
-- | Exercise 2.b
-- |----------------------------------------------------------------------

-- | Implement in Haskell the Structural Semantics of the case statement.


sosStm (Inter (Case a (LabelledStm ll ss lc')) s)
  | elem (aVal a s) ll = Inter ss s
  | otherwise = Inter (Case a lc') s

sosStm (Inter (Case a (Default ss)) s) = Inter ss s

sosStm (Inter (Case a EndLabelledStms) s) = Inter Abort s
```

EXAMEN SEPTIEMBRE

NaturalSemantics.hs

```haskell
module NaturalSemantics where

import           While21


isN :: Aexp -> Bool
isN (N _) = True
isN _ = False

reduce :: Aexp -> Aexp
reduce (N n) = N n
reduce (V v) = V v

reduce (Add a1 a2)
  | isN a1' && isN a2' = N ( a + b )
    where
      a1' = reduce a1
      a2' = reduce a2
      (N a) = reduce a1
      (N b) = reduce a2
reduce (Add a1 a2) = Add (reduce a1) (reduce a2)

reduce (Mult a1 a2)
  | isN a1' && isN a2' = N ( a * b )
    where
      a1' = reduce a1
      a2' = reduce a2
      (N a) = reduce a1
      (N b) = reduce a2
reduce (Mult a1 a2) = Mult (reduce a1) (reduce a2)

reduce (Sub a1 a2)
  | isN a1' && isN a2' = N ( a - b )
    where
      a1' = reduce a1
      a2' = reduce a2
      (N a) = reduce a1
      (N b) = reduce a2
reduce (Sub a1 a2) = Sub (reduce a1) (reduce a2)

exp1 :: Aexp
exp1 =  (V "x")  `Add`  ( (N 5) `Mult` (N 3) ) -- x + 5 * 3

exp2 :: Aexp
exp2 =  (V "x") `Add` ( (N 3) `Add` (N 5) ) -- x + (3 + 5)

exp4 :: Aexp
exp4 = Mult (Add (V "x") (V "y")) (Sub (N 9) (V "z")) -- (x + y) * (9 - z)

exp5 :: Aexp
exp5 = Add (Mult (N 8) (V "y")) (Add (Mult (N 3) (N 2)) (N 5)) -- 8*y + (3*2 +5)


updateState :: State -> Var -> Z -> State
updateState s x v y = if x == y then v else s y

-- representation of configurations for While

data Config = Inter Stm State  -- <S, s>
            | Final State      -- s

-- representation of the transition relation <S, s> -> s'

nsStm :: Config -> Config

-- x := a
nsStm (Inter (Ass x a) s) = Final (updateState s x (aVal a s))

-- skip
nsStm (Inter Skip s) = Final s
```

```haskell
-- s1; s2
nsStm (Inter (Comp ss1 ss2) s) = Final s''
  where
    Final s'  = nsStm (Inter ss1 s)
    Final s'' = nsStm (Inter ss2 s')

-- if b then s1 else s2
nsStm (Inter (If b ss1 ss2) s)
  | bVal b s = Final s'
  where
    Final s' = nsStm (Inter ss1 s)

nsStm (Inter (If b ss1 ss2) s)
  | not(bVal b s) = Final s'
  where
    Final s' = nsStm (Inter ss2 s)

-- swap x y
--     [swap ns] < swap x y , s > -> s'[ y -> A[x] s]     donde s' = s[x -> A[y]s]
--     updateState :: State -> Var -> Z -> State

nsStm (Inter (Swap x y) s) = Final s''
  where
    s' = updateState s x (s y)
    s'' = updateState  s' y (s x)


-- for S1 b S2 S3
nsStm (Inter (For ss1 b ss2 ss3) s)
  | bVal b s' = Final sf
    where
      Final s' = nsStm (Inter ss1 s)
      Final s'' = nsStm (Inter ss3 s')
      Final s''' = nsStm (Inter ss2 s'')
      Final sf = nsStm (Inter (For Skip b ss2 ss3) s''')

nsStm (Inter (For ss1 b ss2 ss3) s)
  | not (bVal b s') = Final s'
    where
      Final s' = nsStm (Inter ss1 s)


----------------------------------------------------------------------
-- DEFINICIONES SEMÁNTICAS
----------------------------------------------------------------------

{-

    Completa la definición semántica de la sentencia swap x y.

    [swap ns] < swap x y , s > -> s'[ y -> A[x] s]     donde s' = s[x -> A[y]s]

-}

{-

    Completa la definición semántica de la sentencia for(S1;b;S2) S3.

                  < S1, s > -> s'  < S3 , s' > -> s'' < S2 , s'' > -> s'''  < for Skip b S2 S3 , s''' > -> sf
    [for tt ns] -------------------------------------------------------------------------------------------
-     si B[b]s' = tt
                                           < for S1 b S2 S3 , s > -> sf

                                              < S1, s > -> s'
    [for ff ns] -----------------------------------------------------------------------------------------
-    si B[b]s' = ff
                                          < for S1 b S2 S3 , s > -> s'

-}


{-

    Completa la definición semántica de la sentencia do Gs od        con GS ::= b -> S ; GS | end

    habria que implementar el or para el no determinismo no?

-}
```

StructuralSemantics.hs

```haskell
module StructuralSemantics where

import          While21


-- representation of configurations for While

data Config = Inter Stm State  -- <S, s>
            | Final State      -- s
            | Stuck Stm State  -- <S, s>

isFinal :: Config -> Bool
isFinal (Final _) = True
isFinal _         = False

isInter :: Config -> Bool
isInter (Inter _ _) = True
isInter _           = False

isStuck :: Config -> Bool
isStuck (Stuck _ _) = True
isStuck _           = False

updateState :: State -> Var -> Z -> State
updateState s x v y = if x == y then v else s y

-- representation of the transition relation <S, s> -> s'

sosStm :: Config -> Config

-- x := a
sosStm (Inter (Ass x a) s) = Final (update s x (aVal a s))
  where
    update s x v y = if x == y then v else s y

-- skip
sosStm (Inter Skip s) = Final s

-- s1; s2
sosStm (Inter (Comp ss1 ss2) s)
  | isFinal next = Inter ss2 s'
  where
    next = sosStm (Inter ss1 s)
    Final s' = next

sosStm (Inter (Comp ss1 ss2) s)
  | isStuck next = Stuck (Comp stm ss2) s'
  where
    next = sosStm (Inter ss1 s)
    Stuck stm s' = next

sosStm (Inter (Comp ss1 ss2) s)
  | isInter next = Inter (Comp ss1' ss2) s'
  where
    next = sosStm (Inter ss1 s)
    Inter ss1' s' = next

-- if b then s1 else s2
sosStm (Inter (If b ss1 ss2) s)
  | bVal b s = Inter ss1 s

sosStm (Inter (If b ss1 ss2) s)
  | not (bVal b s) = Inter ss2 s

-- swap x y
sosStm (Inter (Swap x y) s) = Final s''
  where
    s' = updateState s x (s y)
    s'' = updateState  s' y (s x)

-- for S1 b S2 S3
sosStm (Inter (For ss1 b ss2 ss3) s) = Inter (Comp ss1 ss_if) s
  where
    ss_if = If b ss Skip
    ss = Comp ss3 (Comp ss2 (For Skip b ss2 ss3))
```

```
----------------------------------------------------------------------
-- DEFINICIONES SEMÁNTICAS
----------------------------------------------------------------------


{-
    Completa la definición semántica de la sentencia swap.

    [swap sos] < swap x y , s > => s'[ x -> A[y] s]     donde s' = s[x -> A[y]s]


-}

{-

    Completa la definición semántica de la sentencia for(S1;b;S2) S3.

    [for sos] < for S1 b S2 S3 , s > => < S1 ; if b then (S3;S2; for Skip b S2 S3) else Skip, s >

-}
```

PRIMER PARCIAL 2023

```haskell
{-

Parcial 1 2023 - Teoría de los lenguajes de programación

-}

import           Test.HUnit hiding (State)

-- Ejercicio 1, implementa un fold para Qexp,
-- una funcion val y un test para la función

type NumLit = String
type Var = String

data Val = N NumLit
        | V Var

data Qexp = ZV Val
        | QV Val Val
        | Add Qexp Qexp
        | Mult Qexp Qexp

-- 2/3*y + x/2
q0 :: Qexp
q0 = Add (Mult (QV (N "2") (N "3")) (ZV (V "y"))) (QV (V "x") (N "2"))


foldQexp :: (b -> b -> b) -> (b -> b -> b) -> (Val -> Val -> b) -> (Val -> b) -> Qexp -> b
foldQexp fMult fAdd fQv fZv a = plegar a
    where
        plegar (ZV a) = fZv a
        plegar (QV a b) = fQv a b
        plegar (Add a b) = fAdd (plegar a) (plegar b)
        plegar (Mult a b) = fMult (plegar a) (plegar b)

type  Z      =  Integer
type  State  =  Var -> Z

s0 :: State
s0 "x" = 1
s0 "y" =  3
s0 _   =  0

data Q = Q Z Z
        deriving (Show, Eq)

-- val q0 s0
-- Q 15 6

valToZ :: Val -> State -> Z
valToZ (N a) _ = read a
valToZ (V a) s = s a

val :: Qexp -> State -> Q
val a s = foldQexp fMult fAdd fQV fZV a
        where
                fZV a = Q (valToZ a s) 1
                fQV a b = Q (valToZ a s) (valToZ b s)
                fAdd (Q a b) (Q c d) = Q (a*d+c*b) (b*d)
                fMult (Q a b) (Q c d) = Q (a*c) (b*d)

s1 :: State
s1 "x" = 2
s1 "y" =  3
s1 _   =  0

-- 1/4*y + x/2
q1 :: Qexp
q1 = Add (Mult (QV (N "1") (N "4")) (ZV (V "y"))) (QV (V "x") (N "2"))

testVal = test [ "val of 2/3*y + x/2 con x = 1 e y = 3" ~: Q 15 6  ~=? val q0 s0,
                "val of 2/3*y + x/2 con x = 2 e y = 3" ~: Q 18 6  ~=? val q0 s1,
                "val of 1/8*y + x/2 con x = 2 e y = 3" ~: Q 14 8  ~=? val q1 s1]
```

```haskell
-- Ejercicio 2, implementa un partial state

type PartialState = Var -> Maybe Z

-- A) Completa según el enunciado
ps0 :: PartialState
ps0 "x" = Just 1
ps0 "y" =  Just 5
ps0 "z" =  Just 9
ps0 _   =  Nothing

-- B) Implementa un set para partial state
-- (set ps0 "x" 2) "x" -> Just 2
-- (set ps0 "y" 15) "y" -> Just 15
-- (set ps0 "y" 15) "x" -> Just 1 (Devuelve el valor de x de ps0)
-- (set ps0 "x" 2) "k" -> Nothing

set :: PartialState -> Var -> Z -> PartialState
set ps var value b = if (var == b) then Just value else ps b

-- C) Implementa un unset para partial state
-- (unset ps0 "x") "x" -> Nothing
-- (unset ps0 "y") "x" -> Just 1 (Devuelve el valor de x de ps0)

unset :: PartialState -> Var -> PartialState
unset ps var b = if (var == b) then Nothing else ps b

-- D) Implementa un map para partial state
-- (partialMap ps0 (const 0)) "x" -> Just 0

partialMap :: PartialState -> (Z -> Z) -> PartialState
-- Dos versiones diferentes
{-partialMap ps f b = case ps b of
                        Just a -> Just (f a)
                        Nothing -> Nothing-}
partialMap ps f b
        | ps b == Nothing = Nothing
        | otherwise = Just (f a)
        where Just a = ps b
```

TEORÍA SEMÁNTICA

Semántica Natural

$[\text{ass}_{\text{ns}}]$ $\quad \langle x := a,\ s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$

$[\text{skip}_{\text{ns}}]$ $\quad \langle \texttt{skip},\ s\rangle \to s$

$$[\text{comp}_{\text{ns}}] \quad \frac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1;S_2,\ s\rangle \to s''}$$

$$[\text{if}^{\text{tt}}_{\text{ns}}] \quad \frac{\langle S_1,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'} \text{ if } \mathcal{B}[\![b]\!]s = \textbf{tt}$$

$$[\text{if}^{\text{ff}}_{\text{ns}}] \quad \frac{\langle S_2,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'} \text{ if } \mathcal{B}[\![b]\!]s = \textbf{ff}$$

$$[\text{while}^{\text{tt}}_{\text{ns}}] \quad \frac{\langle S,\ s\rangle \to s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''} \text{ if } \mathcal{B}[\![b]\!]s = \textbf{tt}$$

$[\text{while}^{\text{ff}}_{\text{ns}}]$ $\quad \langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s \text{ if } \mathcal{B}[\![b]\!]s = \textbf{ff}$

```
              < S, s > → s'
[repeat tt] ───────────────────────────      if bVal[b]s' = tt
              < repeat S b , s > → s'


              < S, s > → s'   < repeat S b , s' > → s''
[repeat ff] ───────────────────────────────────────────      if bVal[b]s' = ff
              < repeat S b , s > → s''

          < x := a1 , S > → s'   < S , s' > → s''   < for x (v1 + 1) v2 S, s'' > → s'''
[for tt] ─────────────────────────────────────────────────────────────────────────────      if aVal[a1]s ≤ aVal[a2]s = tt
          < for x a1 a2 S , s > → s'''

          where
            v1 = N^-1[ aVal[a1]s ]
            v2 = N^-1[ aVal[a2]s ]


[for ff] ──────────────────────────────────────      if aVal[x]s ≤ aVal[a2]s = ff
          < for x a1 a2 S , s > → s
```

```
[swap ns] < swap x y , s > → s'[ y → A[x] s]     donde s' = s[x → A[y]s]
```

```
              < S1, s > → s'   < S3 , s' > → s'' < S2 , s'' > → s'''   < for Skip b S2 S3 , s''' > → sf
[for tt ns] ──────────────────────────────────────────────────────────────────────────────────────      si B[b]s' = tt
              < for S1 b S2 S3 , s > → sf

              < S1, s > → s'
[for ff ns] ──────────────────────────────────      si B[b]s' = ff
              < for S1 b S2 S3 , s > → s'
```

Semántica Estructural

$[\text{ass}_{\text{sos}}]$ $\quad \langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$

$[\text{skip}_{\text{sos}}]$ $\quad \langle \texttt{skip},\ s\rangle \Rightarrow s$

$[\text{comp}^1_{\text{sos}}]$ $\quad \dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S'_1,\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S'_1;S_2,\ s'\rangle}$

$[\text{comp}^2_{\text{sos}}]$ $\quad \dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$

$[\text{if}^{\text{tt}}_{\text{sos}}]$ $\quad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$

$[\text{if}^{\text{ff}}_{\text{sos}}]$ $\quad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

$[\text{while}_{\text{sos}}]$ $\quad \langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow$

$$\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle$$

[repeat sos] < repeat S until b , s > ⇒ < S ; If b then Skip else (Repeat S until b), s>

[for sos] < for x:=a1 to a2 do S, s > ⇒ < x:=a1 ; If (a1 ≤ a2) then (S ; for x:=v1+1 to v2 do S) else skip, s >

where
v1 = N^-1 (A[a1]s)
v2 = N^-1 (A[a2]s)

[swap sos] < swap x y , s > ⇒ s'[x → A[y] s]   donde s' = s[x → A[y]s]

[for sos] < for S1 b S2 S3 , s > ⇒ < S1 ; if b then (S3;S2; for Skip b S2 S3) else Skip, s >

$$[\mathrm{var_{ns}}] \qquad \frac{\langle D_V,\ env_V[x\mapsto l],\ sto[l\mapsto v][\mathsf{next}\mapsto \mathrm{new}\ l]\rangle \to_D (env'_V,\ sto')}{\langle \mathtt{var}\ x := a;\ D_V,\ env_V,\ sto\rangle \to_D (env'_V,\ sto')}$$

where $v = \mathcal{A}[\![a]\!](sto \circ env_V)$ and $l = sto$ next

$$[\mathrm{none_{ns}}] \qquad \langle \varepsilon,\ env_V,\ sto\rangle \to_D (env_V,\ sto)$$

$$[\mathrm{ass_{ns}}] \qquad env_V,\ env_P \vdash \langle x := a,\ sto\rangle \to sto[l\mapsto v]$$

where $l = env_V\ x$ and $v = \mathcal{A}[\![a]\!](sto \circ env_V)$

$$[\mathrm{skip_{ns}}] \qquad env_V,\ env_P \vdash \langle \mathtt{skip},\ sto\rangle \to sto$$

$$[\mathrm{comp_{ns}}] \qquad \frac{env_V,\ env_P \vdash \langle S_1,\ sto\rangle \to sto',\quad env_V,\ env_P \vdash \langle S_2,\ sto'\rangle \to sto''}{env_V,\ env_P \vdash \langle S_1;S_2,\ sto\rangle \to sto''}$$

$$[\mathrm{if_{ns}^{tt}}] \qquad \frac{env_V,\ env_P \vdash \langle S_1,\ sto\rangle \to sto'}{env_V,\ env_P \vdash \langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2,\ sto\rangle \to sto'}$$

if $\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{tt}$

$$[\mathrm{if_{ns}^{ff}}] \qquad \frac{env_V,\ env_P \vdash \langle S_2,\ sto\rangle \to sto'}{env_V,\ env_P \vdash \langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2,\ sto\rangle \to sto'}$$

if $\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{ff}$

$$[\mathrm{while_{ns}^{tt}}] \qquad \frac{\begin{array}{c} env_V,\ env_P \vdash \langle S,\ sto\rangle \to sto', \\ env_V,\ env_P \vdash \langle \mathtt{while}\ b\ \mathtt{do}\ S,\ sto'\rangle \to sto'' \end{array}}{env_V,\ env_P \vdash \langle \mathtt{while}\ b\ \mathtt{do}\ S,\ sto\rangle \to sto''}$$

if $\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{tt}$

$$[\mathrm{while_{ns}^{ff}}] \qquad env_V,\ env_P \vdash \langle \mathtt{while}\ b\ \mathtt{do}\ S,\ sto\rangle \to sto$$

if $\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{ff}$

$$[\mathrm{block_{ns}}] \qquad \frac{\begin{array}{c} \langle D_V,\ env_V,\ sto\rangle \to_D (env'_V,\ sto'), \\ env'_V,\ env'_P \vdash \langle S,\ sto'\rangle \to sto'' \end{array}}{env_V,\ env_P \vdash \langle \mathtt{begin}\ D_V\ D_P\ S\ \mathtt{end},\ sto\rangle \to sto''}$$

where $env'_P = \mathrm{upd_P}(D_P,\ env'_V,\ env_P)$

$$[\mathrm{call_{ns}}] \qquad \frac{env'_V,\ env'_P \vdash \langle S,\ sto\rangle \to sto'}{env_V,\ env_P \vdash \langle \mathtt{call}\ p,\ sto\rangle \to sto'}$$

where $env_P\ p = (S,\ env'_V,\ env'_P)$

$$[\mathrm{call_{ns}^{rec}}] \qquad \frac{env'_V,\ env'_P[p\mapsto(S,\ env'_V,\ env'_P)] \vdash \langle S,\ sto\rangle \to sto'}{env_V,\ env_P \vdash \langle \mathtt{call}\ p,\ sto\rangle \to sto'}$$

where $env_P\ p = (S,\ env'_V,\ env'_P)$