COM2001 — Advanced Programming Topics

Exercise Sheet 1: Type Classes

Spring Semester

Problem 1. Look online to find out what functions are associated with members of the type classes `Show` and `Num`.

Solution. According to the manuals at `haskell.org`, we have

```
class  Show a  where
    showsPrec :: Int -> a -> ShowS
    show      :: a -> String
    showList  :: [a] -> ShowS
    -- Minimal complete definition is one of show or showsPrec.

class  (Eq a, Show a) => Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
    -- Minimal complete definition. One of (negate or (-)) + all the others.
    -- The functions abs and signum must satisfy: abs x * signum x == x
    -- For real numbers the signum is either -1 (negative), 0 (zero) or 1 (positive).
```

A programmer defines the type `Nat` (representing the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of *natural numbers*) as follows:

```
data Nat = Zero | Succ Nat  deriving Eq
```

(a) Write down suitable code to make `Nat` an instance of `Num`. Subtraction should be defined so that $x - y = 0$ whenever $y \geq x$.

Solution.

```
instance Num Nat where

    x + Zero     = x
    x + (Succ n) = Succ (x + n)

    Zero - _     = Zero
    x - Zero     = x
    (Succ u) - (Succ v)  = u - v

    x * Zero     = Zero
    x * (Succ n) = x*n + x

    abs x    = x

    signum Zero = 0
    signum _    = 1

    fromInteger n
      | n ≤ 0     = Zero
      | otherwise = Succ (fromInteger (n-1))
```

(b) Show how to make `Nat` a member of `Show` so that natural numbers are printed as integers, e.g.,

– `show Zero` ⇝ `"0"`
– `show (Succ Zero)` ⇝ `"1"`
– `show (Succ (Succ Zero))` ⇝ `"2"`
– `show (Succ (Succ (Succ Zero)))` ⇝ `"3"`

Solution.

```
instance Show Nat where
  show = show ∘ toInteger
    where toInteger Zero = 0
          toInteger (Succ n) = 1 + toInteger n
```

Problem 2. Recall the following definition from the lectures of a computational model:

```
class (Eq cfg) ⇒ Model cfg where
    initialise  :: String → cfg
    acceptState :: cfg    → Bool
    doNextMove  :: cfg    → cfg
    runFrom     :: cfg    → cfg
    runModel    :: String → cfg

    -- Default implementation
    runModel = runFrom ∘ initialise
```

Look online to refresh your memory as to what a pushdown automaton (PDA) is. Show in detail how to implement a PDA using the class `Model`.

Solution. The configuration of a PDA is known once you're told its current state, current stack and remaining input string. I'll represent the stack using a list. Pushing x into the list is given by x:list, and popping uses tail. To find the top of the stack use head. The stack alphabet a must contain a special symbol (epsilon). I'll use ordinary alphabetic characters for both the input and the stack alphabets, and '0' to represent epsilon.

The FSM component is as before, except that the labels are a bit more complicated. For a PDA, the labels on the FSM have the form `c , x → y`. If this arrow goes from state `s` to state `t`, we'll represent it as the tuple `(s,c,x,y,t)`. So:

```
type Transitions s = [(s, Char, Char, Char, s)]

class (Eq s, Show s) ⇒ PDA s where
  initialState :: s
  haltStates :: [s]
  transitions :: Transitions s

data PDAConfig s = PDAConfig {
  state :: s,
  stack :: String,
  input :: String
  } deriving (Eq, Show)

instance (PDA s) ⇒ Model (PDAConfig s) where

  -- initialise  :: String    → PDAConfig s
  initialise str = PDAConfig initialState [] str

  -- acceptState :: PDAConfig s → Bool
  acceptState (PDAConfig s stk ins)
    = null stk && null ins && s ‘elem‘ haltStates

  -- doNextMove  :: PDAConfig s → PDAConfig s
```

```
doNextMove cfg@(PDAConfig q stk ins)
  | null ins        = cfg   -- nothing left to process
  | acceptState cfg = cfg   -- already got the answer
  | otherwise       = (PDAConfig q' stk' ins')
  where
       (q', stk', ins') = if (null next3) then (q,stk,ins)
                          else head next3

       next3 = [ (q2, adjust x y stk, tail ins)
               | (q1, c, x, y, q2) <- transitions
               , q1 == q
               , enabled x stk
               , c == head ins ]

       enabled x stk =
         (x == '0') || (if (null stk) then False else (x == head stk))

       adjust x y stk = case (x, y, stk) of
         ('0','0',   _   )    -> stk
         ('0', y ,   _   )    -> y : stk
         ( x ,'0', (_:ts))    -> ts
         ( x , y , (_:ts))    -> y : ts


-- runFrom     :: PDAConfig s -> PDAConfig s
runFrom cfg@(PDAConfig q stk ins)
  | null ins         = cfg
  | acceptState cfg  = cfg
  | isStuck cfg      = cfg
  | otherwise        = runFrom (doNextMove cfg)
  where
    isStuck cfg@(PDAConfig q stk ins) = null moves
    moves = [ (q1,c,x,y,q2) | (q1,c,x,y,q2) <- transitions
                            , q1 == q, c == head ins
                            , (x == '0' || (if (null stk) then False else (x == head stk))) ]
```

And finally, here is an actual PDA.

```
data MyStates = A | B
  deriving (Show, Eq)

instance PDA MyStates where
  initialState = A
  haltStates = [B]
  transitions = [ (A, 'a', '0', 'u', B),
                  (B, 'b', 'u', '0', B)]
```

Let's see whether it works.

```
recognises :: String -> Bool
recognises str = acceptState ( runModel str :: PDAConfig MyStates )
```