Max Wisniewski , Alexander Steen

Tutor: Ansgar Schneider

## Aufgabe 1

Spezifizieren Sie einen geeigneten Datentyp in Haskell oder Java, zur Be- handlung der abstrakten Syntax der Sprache WHILE.
Elementare Einheiten:

```
type Z = Int
type W = Bool
type I = String
data K = LiteralInteger Z | LiteralBool W
data OP = Plus | Minus | Mult | Div | Mod
data BOP = Eq | Lt | Gt | Lte | Gte | Neq
```

Induktiv aufgebaute Einheiten:

```
data T = Z Z
       | I I
       | TApp T OP T
       | TRead


data B = Literal W
       | Not B
       | BApp T BOP T
       | BRead


data C = Skip
       | Assign I T
       | Seq C C
       | If B C C
       | While B C
       | BOut B
       | TOut T
        deriving Show

type P = C
```

## Aufgabe 2

Vereinbaren Sie das Divisionsbeispiel aus der Vorlesung als Konstante divprog unter Verwendung der in Aufgabe 1 vereinbarten Datentypen.

```
divprog :: P
divprog = Seq
            (While
                (BApp (I "x") Gte (I "y"))
                   (Seq
                    (Assign ("g") (TApp (I "g") Plus (Z 1)))
                    (Assign ("x") (TApp (I "x") Minus (I "y")))))
            (TOut (I "g"))
```

## Aufgabe 3

Definieren Sie je eine Funktion zur Berechnung der Werte von Termen, bzw. booleschen Termen, die die aktuelle Speicherbelegung und die aktuelle Eingabe als Parameter erhält.

Zur Auswertung bekommt von T bekomt evalT ein Wörterbuch von Identifier nach Zahl. Dazu bekommt er eine List von Eingabewerten. Das selbe gilt für den boolschen Term. Zurück kommt der Wert, den der Term ergeben sollte und die Restliste von Eingaben.

```
evalT :: T                          -- arithmetischer Term
        → Map I Z                   -- Variablenbelegung
    → [K]                       -- Liste von Eingaben
    → (Z, [K])                  -- Tupel (Rueckgabewert (Zahl), Input)
evalT (Z v) _ e       = (v,e)
evalT (I v) m e       =
  let mvalue = Map.lookup v m in
    case mvalue of
      Just value  →  (value, e)
      Nothing   → error "No Variable was found"
evalT (TApp t1 op t2) m e =
  let (v1, e1) = evalT t1 m e in
    let (v2, e2) = evalT t2 m e1 in
      (decodeOP op v1 v2, e2)
evalT (TRead) _ ((LiteralInteger e):es) = (e,es)
evalT _ _ _        = error "Not enough or wrong  Input"


evalB :: B                          -- boolscher Term
         → Map I Z                  -- Variablenbelegung
     → [K]                      -- List von Eingaben
     → (W, [K])                 -- Rueckgabewert (Bool)
evalB (Literal b) _ e       = (b,e)
evalB (Not b) m e           =
    let (rBool, e1) = evalB b m e in
        (not rBool, e1)
evalB (BApp t1 bop t2) m e  =
  let (b1, e1) = evalT t1 m e in
    let (b2, e2) = evalT t2 m e1 in
      (decodeBOP bop b1 b2, e2)
evalB BRead m ((LiteralBool b):es)        = (b,es)
evalB _ _ _                 = error "Not enough or wrong Input."


decodeOP :: OP → Z → Z → Z
decodeOP Plus v1 v2     = v1 + v2
decodeOP Minus v1 v2    = v1 - v2
decodeOP Mult v1 v2     = v1 * v2
decodeOP Div v1 v2      = v1 `div` v2
decodeOP Mod v1 v2      = v1 `mod` v2

decodeBOP :: BOP → Z → Z → W
decodeBOP Eq v1 v2    = v1 == v2
decodeBOP Lt v1 v2    = v1 < v2
decodeBOP Gt v1 v2    = v1 > v2
decodeBOP Lte v1 v2   = v1 ≤ v2
decodeBOP Gte v1 v2   = v1 ≥ v2
decodeBOP Neq v1 v2   = v1 /= v2

var :: Map I Z
var = Map.fromList [("x", 5),("y", 6),("a", 10)]

test1,test2 :: T
test1 = TApp (Z 5) Plus (TApp (Z 3) Mult (I "x"))

test2 = TApp (Z 10) Plus (I "b")

test3,test4,test5:: B
```

```
test3 = BApp test1 Eq (I "y")

test4 = BRead

test5 = BApp TRead Lte (TApp TRead Plus (Z 1))
```

## Aufgabe 4

Schreiben Sie einen Ubersetzer für T und B, für eine einfache Kellermaschine.

```
parseT :: T → [String]
parseT (Z v)            = ["push "++(show v)]
parseT (I v)            = ["load "++(show v)]
parseT (TApp t1 op t2)    = (parseT t1) ++ (parseT t2) ++ [nameOP op]
parseT TRead              = ["read"]

parseB :: B → [String]
parseB (Literal b)      = ["push ",(show b)]
parseB (Not b)            = "not":(parseB b)
parseB (BApp t1 bop t2)   = (parseT t1)++(parseT t2)++[nameBOP bop]
paeseB BRead              = ["read"]


nameOP :: OP → String
nameOP Plus   = "plus"
nameOP Minus  = "minus"
nameOP Mult   = "mult"
nameOP Div    = "div"
nameOP Mod    = "mod"

nameBOP :: BOP → String
nameBOP Eq    = "eq"
nameBOP Lt    = "lt"
nameBOP Gt    = "gt"
nameBOP Lte   = "lte"
nameBOP Gte   = "gte"
nameBOP Neq   = "neq"
```