

## Max Wisniewski , Alexander Steen

Tutor: Ansgar Schneider

## Aufgabe 1

Spezifizieren Sie einen geeigneten Datentyp in Haskell oder Java, zur Behandlung 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
      | Id 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
```

Die Formulierung von algebraischen Datentypen findet in Haskell schon in einer grammatikartigen Form statt. Für die einzelnen Terme müssen nur noch eindeutige Konstruktoren vergeben werden, sonst kann alles genau so übernommen werden.

## 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 (Id "x") Gte (Id "y"))
    (Seq
      (Assign ("g") (TApp (Id "g") Plus (Z 1)))
```

```
(Assign ("x") (TApp (Id "x") Minus (Id "y")))))
(TOut (Id "g"))
```

Die Konstante sollte sich von selbst erklären. Zunächst brauchen wir eine Sequenz, der erste Teil ist die Berechnung, der zweite die Asugabe. Im While machen wir den Vergleich, ob x größer als y ist. Innerhalb der Schleife incrementieren wir g einmal und ziehen von x y ab.

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

```
evalT :: T
      -> Map I Z
      -> [K]
      -> (Z, [K])
evalT (Z v) _ e      = (v,e)
evalT (Id 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"
```

Bei beiden geben wir im einfachen (Z, Literate) Fall den Wert einfach zurück. Beim Identifier suchen wir in der Variablenbelegung nach dem Wert und geben diesen, sofern vorhanden, zurück. Bei der Anwendung der Operatoren werten wir ersten linken Term aus, geben die neue Eingabeliste in die rechte Berechnung und setzen danach den Wert zusammen.

```
evalB :: B
      -> Map I Z
      -> [K]
      -> (W, [K])
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."
```

Die folgende Operation sorgt dafür, dass die booleschen und arithmetischen Operationen korrekt umgesetzt werden. Falls es von Interesse ist, was darin steht, wird empfohlen in das Programm zu sehen. Wir haben darüber hinaus fünf Testfälle (test1, ..., test5) und die Konstante (var), die schon 3 Variablen vorbelegt enthält damit man das ganze einmal testen kann.

```
decodeOP :: OP -> Z -> Z -> Z
decodeBOP :: BOP -> Z -> Z -> W
```

## Aufgabe 4

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

### Lösung:

Das parsen passiert hier wieder Rekursiv durch den Baum, indem wir immer zuerst links und danach rechts in die Unterbäume absteigen.

```
parseT :: T → [String]
parseT (Z v)          = ["PUSH " ++ (show v)]
parseT (Id 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]
parseB BRead          = ["READ"]
```

Diese Operation übersetzt die Operationen des Konstruktorterms in die Befehle des Stackautomaten. Hier findet nur ein Namensmapping statt, kann aber wahlweise wiederum im Dokument nachgelesen werden.

Testen kann man das ganze an den selben Fälle, wie Aufgabe 3. Wenn man das ganze mit 'showNice.parseT \$ test1' startet, wird die Ausgabe auch noch schön formatiert.

```
nameOP :: OP → String
nameBOP :: BOP → String
```