Übungszettel 3: Semantik von Programmiersprachen

Jakob Pfender und Yves Müller

Aufgabe 1

Für alle Opertaionen, für die vorher eine Regel der From

```
delta ( < W | S | (T1 OP T2).K | E | A > ) =
       < W | S | T1.T2.OP.K   | E | A >
```

existierte muss diese entfernt werden uns stattdessen folgende Regel hinzugefügt werden:

```
delta ( < W | S | (T1 OP T2).K | E | A > ) =
       < W | S | T2.T1.OP.K   | E | A >
```

Wichtig ist die Auswertungsreihenfolge dann, wenn ein Aufruf Seiteneffekte erzeugt. Dies passiert zum Beispiel beim ändern von Variablen Werten oder lesen von der Eingabe. Dies ist insbesondere kritisch wenn die Operation nicht kommutativ ist.

Hier ein Beispiel in der Haskelldatenstruktur vom letzten Übungsblatt:

```
example :: T
example =  ( Minus ReadInt ReadInt )
```

Aufgabe 2

Der Anfangs der Zustand der neuen Komponente sollte das leere Wort sein, da es am Anfang noch keine Fehler- oder Erfolgsberichte gibt. Für alle Zustände die keinen Nachfolgezustand besitzen, führen wir einen solchen ein, so dass:

```
delta ( < W | S | K | E | A | ε > ) =
       < W | S | K | E | A | m >
```

wobei m eine sinnvolle Meldung sein sollte warum der Zustand keinen Folgezustand hat. Hier einige Beispiele:

```
delta ( < 0.n.W | S | /.K | E | A | ε > ) =
       < W     | S | K   | E | A | "Devide by zero" >

delta ( < W | S | read.K | ε | A | ε > ) =
       < W | S | K      | E | A | "Not enough input" >

delta ( < true.n.W | S | +.K | E | A | ε > ) =
       <        W | S | K   | E | A | "Type conflict" >
```

Zuletzt fügen wir folgende Übergänge hinzu, damit die Maschine auch das Ende der Ausführung anzeigen kann:

```
delta ( < W | S | ε | E | A | ε > ) =
       < W | S | ε | E | A | "Ausführung beendet"  >
```

```
delta ( < W | S | K | E | A | "Ausführung beendet" > ) =
        < W | S | K | E | A | "Ausführung beendet"  >
```

Aufgabe 3

Die Menge komm muss um ein Symbol *repeat* erweiter werden, und natürlich auch um die neue Grammtikregel, da diese ja ein Element von *C* ist. Folgende Übergänge müssen zur delta-Funktion hinzukommen:

```
delta ( < W | S | (repeat C until B).K | E | A > ) =
        < W | S | C.B.repeat.C.B.K | E | A >

delta ( < false.W | S | repeat.C.B.K | E | A > ) =
        < W | S | C.B.repeat.C.B.K | E | A >

delta ( < true.W | S | repeat.C.B.K | E | A > ) =
        < W | S | K | E | A >
```

Aufgabe 4

Wir haben versucht die Aufgabe in Haskell zu implementieren, auf Grund von einer nicht optimalen Zeitplannung ist sie leider nicht ganz fertig.

```
import Data.Map as Map
-- Definition of while

data Konst = Z Int | W Bool
    deriving Show

type I = String

data OP =  Plus | Minus | Mul | Div | Mod
    deriving (Show, Eq)

data BOP = Gt | Lt | Geq | Leq | Eq | Neq
    deriving (Show, Eq)


data T = Num Int | Id I | Term T OP T | ReadInt
    deriving Show

data B = Bool Bool | Not B | Expr T BOP T | ReadBool
    deriving Show

data C = Skip | Assign I T | Seq C C | If B C C | While B C | OutputInt T | OutputBool B
    deriving Show

type P = C

-- definition of wskea machine
data WE = ZW Bool | WW Int | I I
    deriving Show
data KE = GProg P | GTerm T | GBTerm B | Op OP | Bop BOP | AssignK | WhileK | IfK | NotK | OutputK
    deriving Show

type W = [WE]
type S = Map.Map String Int
type K = [KE]
type E = [Konst]
type A = [Konst]
```

```
data State = State  { w :: W
                    , s :: S
                    , k :: K
                    , e :: E
                    , a :: A }


start :: P -> E  -> State
start p e = State { w = []
                  , s = Map.empty :: Map String Int
                  , k = [GProg p]
                  , e = e
                  , a = [] }
```