Aufgabenblatt 7 - Lars Gröber

1)

a)

Befehl	Stack
PushK 11	11
PushK 13	11,13
Push 1	11,13,11
Sub	11,2
Push 0	11,2,2
Mark 1	11,2,2
Branchz 2	11,2
Slide 1 1	2
PushK 2	2,2
PushK 4	2,2,4
Mult	2,8
PushK 8	2,8,8
Sub	2,0
Jump 1	2,0
Branchz 2	2
Mark 2	2

b)

Siehe Blatt7_LarsGroeber.hs

2)

a)

```
pushK 1
pushK 1
slide 2 2
```

b)

```
pushK 1
-
branchz Second
pop
pop
```

```
jump False

Second.
pushK 1
-
branchz Third
pop
jump False

Third.
branchz True
False.
pushK 0
jump End

True.
pushK 1

End.
```

c)

```
START.
# 2
pop
-
push 0
pushK -1

# 8
pop
pop
pop
pushK 1
push 0
pushK -1

# 3
pop
pop
-
pushK 1
pushK 0
pushK -1

# 8
pop
slide 1 2
pop
pushK 1
pushK 1
pushK 1
pushK -1

# 4
+
pushK -1

# 8
-
pushK -1

# 5
+
*
*
```

```
push 0
pushK 1
pushK -1

# 8
pop
pop
pop
pop
pushK 1
push 0
push 0
pushK -1

# 6
+
*
push 1
pushK -1

# 8
+
pop
pop
pushK 1
pushK 1
pushK -1

# 7
+
*
*
pushK 1
push 0
pushK -1

# 8
pop
pop
pop
pop
pushK 1
push 0
push 0
pushK -1

jump START
```

```haskell
--
================================================================================
-- Grundlagen der Programmierung 2
-- Aufgabenblatt 7
--
================================================================================

  module Blatt7_LarsGroeber where
  import Data.Char
  import Data.List

  --
================================================================================
  -- =
  -- = Aufgabe 1
  -- =

  ---
  --- b)
  ---
  data Befehl = PushK Int | Pop | Push Int | Mult | Add | Sub | Mark Marke
                | Jump Marke | Branchz Marke | Slide Int Int
   deriving (Eq,Show)

  type Marke = Int

  type StackProgramm = [Befehl]
  type Stack = [Int]

  run :: StackProgramm -> Stack
  -- Funktion, die ein Stackprogramm entgegen nimmt, ausführt und den
Ergebnisstack zurückgibt.
  -- Beispiel: run [PushK 1] soll [1] ergeben
  run programm = interpretiere programm [] programm

  interpretiere :: StackProgramm -> Stack -> StackProgramm -> Stack
  -- Funktion, die die Funktion I einer Stackmaschine implementiert.
  -- Beispiel: interpretiere [PushK 1] [] [PushK 1] soll [1] ergeben.
  interpretiere [] a prg = a
  interpretiere (PushK i : xs) a prg = interpretiere xs (i:a) prg
  interpretiere (Pop : xs) a prg = interpretiere xs (tail a) prg
  interpretiere (Push i: xs) a prg = interpretiere xs ((a!!i):a) prg
  interpretiere (Mult : xs) (a:b:as) prg = interpretiere xs (b*a:as) prg
  interpretiere (Add : xs) (a:b:as) prg = interpretiere xs (b+a:as) prg
  interpretiere (Sub : xs) (a:b:as) prg = interpretiere xs (b-a:as) prg
  interpretiere (Mark m : xs) a prg = interpretiere xs a prg
  interpretiere (Jump m : xs) a prg = interpretiere (dropWhile (Mark m /=)
prg) a prg
  interpretiere (Branchz m : xs) a prg = if 0 == head a then interpretiere
(dropWhile (Mark m /=) prg) (tail a) prg
                                       else interpretiere xs (tail a) prg
  interpretiere (Slide i j : xs) a prg = interpretiere xs (take i a ++ drop (j
+ i) a) prg

  {-
  Testfälle:
  run test1a                  `shouldBe` [2]
```

```haskell
  run [PushK 1]               `shouldBe` [1]
  run [PushK 1, Push 0, Add]  `shouldBe` [2]
  run [PushK 2, PushK 3, Sub] `shouldBe` [-1]
  -}

  -- Zum Testen von Aufgabenteil a).
  test1a =
   [PushK 11
   ,PushK 13
   ,Push 1
   ,Sub
   ,Push 0
   ,Mark 1
   ,Branchz 2
   ,Slide 1 1
   ,PushK 2
   ,PushK 4
   ,Mult
   ,PushK 8
   ,Sub
   ,Jump 1
   ,Mark 2]

  -- =
  -- =
  --
================================================================================
=
```