Abstraktsed masinad ja transleerimine Haskellis

- Keele While abstraktne süntaks:

```
type Var    = String
type OpName = String

data Expr = Var Var
          | Num Int
          | Op1 OpName Expr
          | Op2 OpName Expr Expr

data Stmt = Assign Var Expr
          | Skip
          | Seq Stmt Stmt
          | If Expr Stmt Stmt
          | While Expr Stmt
```

Abstraktne masin — AM1

- Instruktsioonid:

```
type Code  = [Instr]

data Instr = Push Int
           | Add | Mul | Sub | Div | Mod | Neg
           | Eq  | NEq | LE  | LEq | GE  | GEq
           | And | Or  | Not
           | Fetch Var | Store Var
           | Noop
           | Branch Code Code
           | Loop Code
```

Abstraktne masin — AM1

- Konfiguratsioonid:

```
type Stack  = [Int]
type State  = [(Var,Int)]
type AMConf = (Code, Stack, State)
```

- Abifunktsioonid:

```
fetch :: Var -> State -> Int
fetch x s = case lookup x s of
              Nothing -> error ("Unbound variable: " ++ x)
              Just v  -> v

store :: Var -> Int -> State -> State
store x v []     = [(x,v)]
store x v (y:ys) | x == fst y = (x,v):ys
                 | otherwise  = y : store x v ys
```

Abstraktne masin — AM1

- Struktuurne semantika:

```
amStep :: AMConf -> Maybe AMConf
amStep (Push n  : c,     e, s)   = Just (c, n : e, s)
amStep (Fetch x : c,     e, s)   = Just (c, v : e, s)
       where v = fetch x s
amStep (Store x : c, z : e, s)   = Just (c, e, s')
       where s' = store x z s
amStep (Add : c, z1 : z2 : e, s) = Just (c, v : e, s)
       where v = z1 + z2
...
amStep (Eq : c, z1 : z2 : e, s) = Just (c, v : e, s)
       where v = if z1 == z2 then 1 else 0
...
```

Abstraktne masin — AM1

- Struktuurne semantika (järg):

```
amStep (Noop : c, e, s) = Just (c, e, s)
amStep (Branch c1 c2 : c, z : e, s)
       | z /= 0    = Just (c1 ++ c, e, s)
       | otherwise = Just (c2 ++ c, e, s)
amStep (Loop c1 : c, z : e, s)
       | z /= 0    = Just (c1 ++ Loop c1 : c, e, s)
       | otherwise = Just (c, e, s)
amStep _ = Nothing

amRun :: Code -> State -> [AMConf]
amRun c s = initConf : unfoldr step initConf
   where initConf = (c,[],s)
         step = fmap (λ nc -> (nc,nc)) . amStep
```

Abstraktne masin — AM1

- Operaatorid:

```
op1s, op2s :: [(OpName, Instr)]
op1s = [("!", Not),  (", Neg)]
op2s = [("+", Add),  (", Sub)
       ,("*", Mul),  ("/", Div), ("%", Mod)
       ,("==", Eq),  ("!=", NEq)
       ,(«",  LE),  («=", LEq)
       ,(»",  LE),  (»=", LEq)
       ,("&&", And), (|", Or)]

op2instr op table
     = case lookup op table of
         Nothing -> error ("Unknown operator: " ++ op)
         Just i  -> i
```

Abstraktne masin — AM1

- Avaldiste transleerimine:

```
transE :: Expr -> Code
transE (Var x)          = [Fetch x]
transE (Num n)          = [Push n]
transE (Op1 op e)       = transE e  ++ [i]
   where i = op2instr op op1s
transE (Op2 op e1 e2) = transE e1 ++ transE e2 ++ [i]
   where i = op2instr op op1s
```

- NB! Väga ebaefektiivne kuna instruktsioone lisatakse koodijada lõppu.

Abstraktne masin — AM1

- Transleerimisfunktsioonide signatuur:

```
class Trans a where
   trans     :: a -> Code
   transPrec :: a -> Code -> Code

   trans x       = transPrec x []
   transPrec x c = trans x ++ c
```

- Avaldiste transleerimine:

```
instance Trans Expr where
  transPrec (Var x)         c  = Fetch x : c
  transPrec (Num n)         c  = Push n : c
  transPrec (Op1 op e)      c  = transPrec e  (i:c)
       where i = op2instr op op1s
  transPrec (Op2 op e1 e2) c  = transPrec e2
                                      (transPrec e1 (i:c))
       where i = op2instr op op2s
```

Abstraktne masin — AM1

- Lausete transleerimine:

```
instance Trans Stmt where
  transPrec (Assign x e) c  = transPrec e (Store x : c)
  transPrec Skip         c  = Noop : c
  transPrec (Seq s1 s2)  c  = transPrec s1
                                   (transPrec s2 c)
  transPrec (If e s1 s2) c  = transPrec e
                                   (Branch (trans s1)
                                           (trans s2) : c)
  transPrec (While e s)  c  = c1 ++ Loop c2 : c
       where c1 = trans e
             c2 = transPrec s c1
```

Abstraktne masin — AM2

- Instruktsioonid:

```
data Instr = Push Int
           | Load | Store
           | Add | Mul | Sub | Div | Mod | Neg
           ....
```

- Konfiguratsioonid:

```
type Stack  = [Int]
type AMConf = (Code, Stack)

store :: Int -> Int -> Stack -> Stack
store v i e = e1 ++ v : e2
    where (e1, _:e2) = splitAt i e
```

Abstraktne masin — AM2

- Struktuurne semantika:

```
amStep :: AMConf -> Maybe AMConf
amStep (Push n  : c,   e)  = Just (c, n : e)
amStep (Load : c,  z : e)  = Just (c, v : e)
       where v = e !! z
amStep (Store : c, z1 : z2 : e)  = Just (c, e')
       where e' = store z2 z1 e
amStep (Add : c, z1 : z2 : e) = Just (c, v : e)
       where v = z1 + z2
....

amRun :: Code -> Stack -> [AMConf]
amRun c e = initConf : unfoldr step initConf
   where initConf = (c,e)
         step = fmap ( nc -> (nc,nc)) . amStep
```

Abstraktne masin — AM2

- Transleerimisfunktsioonide signatuur:

```
trans     :: a -> Env -> Int -> Code
transPrec :: a -> Env -> Int -> Code -> Code

trans x env h       = transPrec x env h []
transPrec x env h c = trans x env h ++ c
```

- Avaldiste transleerimine:

```
instance Trans Expr where
  transPrec (Var x)    env h c = Push n : Load : c
       where n = h - getLoc x env
  transPrec (Num n)    env h c = Push n : c
  transPrec (Op1 op e) env h c = transPrec e env h (i:c)
       where i = op2instr op op1s
  transPrec (Op2 op e1 e2) env h c
                     = transPrec e2 env h
                           (transPrec e1 env (h+1) (i:c))
       where i = op2instr op op2s
```

Abstraktne masin — AM2

- Lausete transleerimine:

```
instance Trans Stmt where
  transPrec (Assign x e) env h c
                    = transPrec e env h (Push n : Store : c)
       where n = h - getLoc x env
  transPrec Skip env h c = Noop : c
  transPrec (Seq s1 s2)  env h c
                    = transPrec s1 env h
                          (transPrec s2 env h c)
  transPrec (If e s1 s2) env h c
                    = transPrec e env h
                           (Branch (trans s1 env h)
                                   (trans s2 env h) : c)
  transPrec (While e s)  env h c
                    = c1 ++ Loop c2 : c
       where c1 = trans e env h
             c2 = transPrec s env h c1
```

Lokaalsed muutujad

- Abstraktne süntaks:

```
data Stmt = ....
          | Block [Decl] Stmt

data Decl = Decl Var Expr
```

- Plokkstruktuuride transleerimine:

```
instance Trans Stmt where
  ....
  transPrec (Block ds s) env h c
                    = transBlock ds s env h c
       where transBlock [] s env h c
                    = transPrec s env h c
             transBlock (Decl x e : ds) s env h c
                    = transPrec e env h
                              (transBlock ds s env' (h+1) c)
                    where env' = (x,h+1) : env
```

Tsüklikatkestus

- Abstraktne süntaks:

```
data Stmt = ....
          | Break
```

- Instruktsioonid:

```
data Instr = ....
           | Brk
```

- Tsüklikatkestuste transleerimine:

```
instance Trans Stmt where
  ....
  transPrec Brake env h c  = Brk : c
```

Abstraktne masin — AM3

- Konfiguratsioonid:

```
type Stack  = [Int]
type CStack = [(Code,Code)]
type AMConf = (Code, Stack, CStack)
```

- Struktuurne semantika:

```
amStep :: AMConf -> Maybe AMConf
amStep (Push n : c, e, cs) = Just (c, n : e, cs)
....
amStep (Loop c1 : c, z : e, cs)
       | z /= 0    = Just (c1, e, (c1,c):cs)
       | otherwise = Just (c, e, cs)
amStep ([], z : e, (c1,c):cs)
       | z /= 0    = Just (c1, e, (c1,c):cs)
       | otherwise = Just (c, e, cs)
amStep (Brk : _, e, (_,c):cs) = Just (c, e, cs)
amStep _ = Nothing
```