

Aritmeetiliste avaldiste väärtustamine

Aritmeetilised avaldised

```
infixl 10 :+:, :-:  
infixl 11 :*, :/  
data Expr = Num Int  
          | (:+:) Expr Expr  
          | (: -:) Expr Expr  
          | (:*:) Expr Expr  
          | (: /:) Expr Expr
```

Näited

```
exp1 : Expr  
exp1 = Num 1 :+: Num 2  
exp2 : Expr  
exp2 = Num 2 :* (Num 4 :-: Num 1)  
exp3 : Expr  
exp3 = Num 4 :* Num 3 :/: Num 2
```

Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

```
eval1 : Expr → Int
eval1 (Num i)      = i
eval1 (e1 :+: e2)  = (eval1 e1) + (eval1 e2)
eval1 (e1 :-: e2)  = (eval1 e1) - (eval1 e2)
eval1 (e1 **: e2)  = (eval1 e1) * (eval1 e2)
eval1 (e1 :/: e2)  = (eval1 e1) `div` (eval1 e2)
```

Aritmeetiliste avaldiste väärtustamine

Proovime järgi ...

```
| Main> eval1 exp2  
| 6
```

Aritmeetiliste avaldiste väärtustamine

Proovime järgi ...

```
| Main> eval1 exp2  
| 6
```

Töötab ...

Aritmeetiliste avaldiste väärtustamine

Proovime järgi ...

```
| Main> eval1 exp2  
| 6
```

```
| Main> eval1 (Num 3 :/: Num 0)  
| let False = True in prim__div_Int x y
```

Töötab ... kuid veatöötlus puudub

Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

```
eval1 : Expr → Int
eval1 (Num i)      = i
eval1 (e1 :+: e2)   = (eval1 e1) + (eval1 e2)
eval1 (e1 :-: e2)   = (eval1 e1) - (eval1 e2)
eval1 (e1 **: e2)   = (eval1 e1) * (eval1 e2)
eval1 (e1 :/: e2)   = (eval1 e1) `div` (eval1 e2)
```

Kas saame teha paremini?

Aritmeetiliste avaldiste väärtustamine

Veatöötusega väärtustaja

```
eval2 : Expr → Maybe Int
eval2 (Num i)      = Just i
eval2 (e1 :+: e2) = case eval2 e1 of
    Nothing ⇒ Nothing
    Just v1  ⇒ case eval2 e2 of
        Nothing ⇒ Nothing
        Just v2  ⇒ Just (v1+v2)
eval2 (e1 :-: e2) = ...
eval2 (e1 **: e2) = ...
```

Aritmeetiliste avaldiste väärtustamine

Veatöötusega väärtustaja (järg.)

```
eval2 (e1 :/: e2) = case eval2 e1 of
  Nothing => Nothing
  Just v1 => case eval2 e2 of
    Nothing => Nothing
    Just v2 =>
      if v2==0
      then Nothing
      else Just (v1 `div` v2)
```


Aritmeetiliste avaldiste väärtustamine

Proovime järgi ...

```
| Main> eval2 exp2  
| Just 6
```

Aritmeetiliste avaldiste väärtustamine

Proovime järgi ...

```
| Main> eval2 exp2  
| Just 6
```

```
| Main> eval2 (Num 3 :/: Num 0)  
| Nothing
```

Töötab nagu vaja!!

Aritmeetiliste avaldiste väärtustamine

Veatöötusega väärtustaja (järg.)

```
eval2 (e1 :/: e2) = case eval2 e1 of  
    Nothing -> Nothing
```

Kuid ...

Palju koodi duplitseerimist

```
    if v2==0  
    then Nothing  
    else Just (v1 `div` v2)
```

Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

```

eval1 : Expr → Int
eval1 (Num i)      = i
eval1 (e1 :+: e2) = (eval1 e1) + (eval1 e2)
eval1 (e1 :-: e2) = (eval1 e1) - (eval1 e2)
eval1 (e1 :*: e2) = (eval1 e1) * (eval1 e2)
eval1 (e1 :/: e2) = (eval1 e1) `div` (eval1 e2)

```

Veatöõtlusega väärtustaja

```

eval2 : Expr → Maybe Int
eval2 (Num i)      = Just i
eval2 (e1 :+: e2) = case eval2 e1 of
  Nothing ⇒ Nothing
  Just v1 ⇒ case eval2 e2 of
    Nothing ⇒ Nothing
    Just v2 ⇒ Just (v1+v2)
eval2 (e1 :-: e2) = case eval2 e1 of
  Nothing ⇒ Nothing
  Just v1 ⇒ case eval2 e2 of
    Nothing ⇒ Nothing
    Just v2 ⇒ Just (v1-v2)
eval2 (e1 :*: e2) = case eval2 e1 of
  Nothing ⇒ Nothing
  Just v1 ⇒ case eval2 e2 of
    Nothing ⇒ Nothing
    Just v2 ⇒ Just (v1*v2)
eval2 (e1 :/: e2) = case eval2 e1 of
  Nothing ⇒ Nothing
  Just v1 ⇒ case eval2 e2 of
    Nothing ⇒ Nothing
    Just v2 ⇒
      if v2==0
      then Nothing
      else Just (v1 `div` v2)

```

Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

```
eval1 : Expr → Int
eval1 (Num i)      = i
eval1 (e1 :+: e2) = (eval1 e1) + (eval1 e2)
eval1 (e1 :-: e2) = (eval1 e1) - (eval1 e2)
eval1 (e1 :*: e2) = (eval1 e1) * (eval1 e2)
eval1 (e1 :/: e2) = (eval1 e1) `div` (eval1 e2)
```

Veatöõtlusega väärtustaja

```
eval2 : Expr → Maybe Int
eval2 (Num i)      = Just i
eval2 (e1 :+: e2) = case eval2 e1 of
  Nothing ⇒ Nothing
  Just v1 ⇒ case eval2 e2 of
    Nothing ⇒ Nothing
    Just v2 ⇒ Just (v1+v2)
eval2 (e1 :-: e2) = case eval2 e1 of
  Nothing ⇒ Nothing
  Just v1 ⇒ case eval2 e2 of
    Nothing ⇒ Nothing
    Just v2 ⇒ Just (v1-v2)
eval2 (e1 :*: e2) = case eval2 e1 of
  Nothing ⇒ Nothing
  Just v1 ⇒ case eval2 e2 of
    Nothing ⇒ Nothing
    Just v2 ⇒ Just (v1*v2)
eval2 (e1 :/: e2) = case eval2 e1 of
  Nothing ⇒ Nothing
  Just v1 ⇒ case eval2 e2 of
    Nothing ⇒ Nothing
    Just v2 ⇒
      if v2==0
      then Nothing
      else Just (v1 `div` v2)
```

Mis on sarnased koodimustrid, mida saame refaktoriseerida?

Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

```
eval1 : Expr → Int
eval1 (Num i)    = i
eval1 (e1 :+: e2) = (eval1 e1) + (eval1 e2)
eval1 (e1 :-: e2) = (eval1 e1) - (eval1 e2)
eval1 (e1 :*: e2) = (eval1 e1) * (eval1 e2)
eval1 (e1 :/: e2) = (eval1 e1) / (eval1 e2)
```

Veatöõtlusega väärtustaja

Paneme tähele, et:

- alamavaldiste väärtustamine toimub järjestikku;

```
Just v1 => case eval2 e2 of
  Nothing => Nothing
  Just v2 =>
    if v2==0
    then Nothing
    else Just (v1 `div` v2)
```

Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

```
eval1 : Expr → Int
eval1 (Num i)    = i
eval1 (e1 :+: e2) = (eval1 e1) :+: (eval1 e2)
eval1 (e1 :-: e2) = (eval1 e1) :-: (eval1 e2)
eval1 (e1 :*: e2) = (eval1 e1) :*: (eval1 e2)
eval1 (e1 :/: e2) = (eval1 e1) :/: (eval1 e2)
```

Veatöötusega väärtustaja

Paneme tähele, et:

- alamavaldiste väärtustamine toimub järjestikku;
- kui üks ebaõnnestub, siis ebaõnnestub kogu avaldise väärtustamine;

```
Just v1 => case eval2 e2 of
  Nothing => Nothing
  Just v2 =>
    if v2==0
    then Nothing
    else Just (v1 `div` v2)
```

Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

```
eval1 : Expr → Int
eval1 (Num i)    = i
eval1 (e1 :+: e2) = (eval1 e1) :+: (eval1 e2)
eval1 (e1 :-: e2) = (eval1 e1) :-: (eval1 e2)
eval1 (e1 :*: e2) = (eval1 e1) :*: (eval1 e2)
eval1 (e1 :/: e2) = (eval1 e1) :/: (eval1 e2)
```

Veatöötusega väärtustaja

Paneme tähele, et:

- alamavaldiste väärtustamine toimub järjestikku;
- kui üks ebaõnnestub, siis ebaõnnestub kogu avaldise väärtustamine;
- õnnestumise korral antakse tulemväärtus edasi järgnevale arvutusele.

```
Just v1 ⇒ case eval2 e2 of
  Nothing ⇒ Nothing
  Just v2 ⇒
    if v2 == 0
    then Nothing
    else Just (v1 `div` v2)
```


Aritmeetiliste avaldiste väärtustamine

Arvutuste järjestikustamine

```
evSeq : Maybe Int → (Int → Maybe Int) → Maybe Int
evSeq ma f = case ma of
    Nothing ⇒ Nothing
    Just a  ⇒ f a
```

Aritmeetiliste avaldiste väärtustamine

Veatöötusega väärtustaja

```
eval3 : Expr → Maybe Int
eval3 (Num i)      = Just i
eval3 (e1 :+: e2)  = eval3 e1 `evSeq` λv1 ⇒
                      eval3 e2 `evSeq` λv2 ⇒
                      Just (v1+v2)
eval3 (e1 :-: e2)  = ...
eval3 (e1 :*: e2)  = ...
eval3 (e1 :/: e2)  = eval3 e1 `evSeq` λv1 ⇒
                      eval3 e2 `evSeq` λv2 ⇒
                      if v2 == 0
                        then Nothing
                        else Just (v1 `div` v2)
```

Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

```
eval1 : Expr → Int
eval1 (Num i)      = i
eval1 (e1 :+: e2) = (eval1 e1) + (eval1 e2)
eval1 (e1 :-: e2) = (eval1 e1) - (eval1 e2)
eval1 (e1 :*: e2) = (eval1 e1) * (eval1 e2)
eval1 (e1 :/: e2) = (eval1 e1) `div` (eval1 e2)
```

Veatöõtlusega väärtustaja

```
eval3 : Expr → Maybe Int
eval3 (Num i)      = Just i
eval3 (e1 :+: e2) = eval3 e1 `evSeq` λv1 ⇒
                    eval3 e2 `evSeq` λv2 ⇒
                    Just (v1+v2)
eval3 (e1 :-: e2) = eval3 e1 `evSeq` λv1 ⇒
                    eval3 e2 `evSeq` λv2 ⇒
                    Just (v1-v2)
eval3 (e1 :*: e2) = eval3 e1 `evSeq` λv1 ⇒
                    eval3 e2 `evSeq` λv2 ⇒
                    Just (v1*v2)
eval3 (e1 :/: e2) = eval3 e1 `evSeq` λv1 ⇒
                    eval3 e2 `evSeq` λv2 ⇒
                    if v2 == 0
                    then Nothing
                    else Just (v1 `div` v2)
```

Aritmeetiliste avaldiste väärtustamine

Lihtne v **Üldistus**

Tüüp **Maybe** **a** esitab **potentsiaalselt ebaõnnestuvaid arvutusi**.

```
eval1 : E
eval1 (Num
eval1 (el
eval1 (el
eval1 (el
eval1 (el
eval1 (el
```

```
else Just (v1 `div` v2)
```

Aritmeetiliste avaldiste väärtustamine

Lihtne v **Üldistus**

Tüüp **Maybe a** esitab **potentsiaalselt ebaõnnestuvaid arvutusi**.

- Arvutus võib **õnnestuda** tagastades tüüpi **a** väärtuse.

```
eval1 : E
eval1 (Num
eval1 (el
eval1 (el
eval1 (el
eval1 (el
eval1 (el
```

```
else Just (v1 `div` v2)
```

Aritmeetiliste avaldiste väärtustamine

Lihtne v **Üldistus**

Tüüp **Maybe a** esitab **potentsiaalselt ebaõnnestuvaid arvutusi**.

- Arvutus võib **õnnestuda** tagastades tüüpi **a** väärtuse.
- Arvutus võib **ebaõnnestuda** ilma väärtust tagastamata.

```
else Just (v1 `div` v2)
```

Aritmeetiliste avaldiste väärtustamine

Lihtne v **Üldistus**

Tüüp **Maybe a** esitab **potentsiaalselt ebaõnnestuvaid arvutusi**.

- Arvutus võib **õnnestuda** tagastades tüüpi **a** väärtuse.
- Arvutus võib **ebaõnnestuda** ilma väärtust tagastamata.

Arvutusi saab komponeerida **järjestikku**.

```
else Just (v1 `div` v2)
```

Aritmeetiliste avaldiste väärtustamine

Lihtne v Üldistus

```
eval1 : E
eval1 (Num
eval1 (el
eval1 (el
eval1 (el
eval1 (el
eval1 (el
```

Tüüp **Maybe a** esitab **potentsiaalselt ebaõnnestuvaid arvutusi**.

- Arvutus võib **õnnestuda** tagastades tüüpi **a** väärtuse.
- Arvutus võib **ebaõnnestuda** ilma väärtust tagastamata.

Arvutusi saab komponeerida **järjestikku**.

- Kui üks alamarvutus ebaõnnestub, siis ebaõnnestub kogu arvutus!

```
else Just (v1 `div` v2)
```


Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

```
eval1 : Expr → Int
eval1 (Num i)      = i
eval1 (e1 :+: e2) = (eval1 e1) + (eval1 e2)
eval1 (e1 :-: e2) = (eval1 e1) - (eval1 e2)
eval1 (e1 :*: e2) = (eval1 e1) * (eval1 e2)
eval1 (e1 :/: e2) = (eval1 e1) / (eval1 e2)
```

Veatöötusega väärtustaja

```
eval3 : Expr → Maybe Int
eval3 (Num i)      = Just i
eval3 (e1 :+: e2) = eval3 e1 `evSeq` λv1 ⇒
                    ⇒
                    eval3 e2 `evSeq` λv2 ⇒
                    ⇒
                    Just (v1+v2)
eval3 (e1 :-: e2) = eval3 e1 `evSeq` λv1 ⇒
                    ⇒
                    eval3 e2 `evSeq` λv2 ⇒
                    ⇒
                    Just (v1-v2)
eval3 (e1 :*: e2) = eval3 e1 `evSeq` λv1 ⇒
                    ⇒
                    eval3 e2 `evSeq` λv2 ⇒
                    ⇒
                    Just (v1*v2)
eval3 (e1 :/: e2) = eval3 e1 `evSeq` λv1 ⇒
                    ⇒
                    eval3 e2 `evSeq` λv2 ⇒
                    ⇒
                    if v2 == 0
                    then Nothing
                    else Just (v1 `div` v2)
```

Ebaõnnestumine on kõrvalefekt, mis kaudselt mõjutab kõiki järgnevaid arvutusi!

Potensiaalselt ebaõnnestuvad arvutused

Maybe monaad

```
mReturn      : a → Maybe a
mReturn a    = Just a

mFail        : Maybe a
mFail        = Nothing

mBind        : Maybe a → (a → Maybe b) → Maybe b
mBind ma f   = case ma of
                Nothing ⇒ Nothing
                Just a   ⇒ f a
```

Potentsiaalselt ebaõnnestuvad arvutused

Veatöötlusega väärtustaja

```
eval4 : Expr → Maybe Int
eval4 (Num i)      = mReturn i
eval4 (e1 :+: e2)  = eval4 e1 `mBind` λv1 ⇒
                      eval4 e2 `mBind` λv2 ⇒
                      mReturn (v1+v2)

eval4 (e1 :-: e2)  = ...
eval4 (e1 :*: e2)  = ...
eval4 (e1 :/: e2)  = eval4 e1 `mBind` λv1 ⇒
                      eval4 e2 `mBind` λv2 ⇒
                      if v2 == 0
                        then mFail
                        else mReturn (v1 `div` v2)
```

Puu lehtede loendamine

Kahendpuud

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
```

Näited

```
tree1  : Tree Int
tree1  = Branch (Leaf 7) (Leaf 3)

tree2  : Tree Char
tree2  = Branch (Branch (Leaf 'A')
                     (Branch (Leaf 'B')
                              (Leaf 'C')))
          (Branch (Leaf 'D')
                   (Leaf 'E'))
```

Puu lehtede loendamine

Loendamine kasutades ilmutatud loendurit

```
labAux1          : Tree a → Int → (Tree Int, Int)
labAux1 (Leaf x)  c = (Leaf c, c+1)
labAux1 (Branch t1 t2) c = let (t1', c1) = labAux1 t1 c
                             (t2', c2) = labAux1 t2 c1
                             in (Branch t1' t2', c2)

labelTree1       : Tree a → Tree Int
labelTree1 t     = fst (labAux1 t 0)
```

Puu lehtede loendamine

Loendamine kasutades ilmutatud loendurit

```
labAux1      : Tree a → Int → (Tree Int, Int)
labAux1 (Leaf x)    c = (Leaf c, c+1)
labAux1 (Branch t1 t2) c = let (t1', c1) = labAux1 t1 c
                             (t2', c2) = labAux1 t2 c1
                             in (Branch t1' t2', c2)

labelTree1    : Tree a → Tree Int
labelTree1 t = fst (labAux1 t 0)
```

Paneme tähele, et ...

- algul loendur algväärtustatakse;

Puu lehtede loendamine

Loendamine kasutades ilmutatud loendurit

```
labAux1      : Tree a → Int → (Tree Int, Int)
labAux1 (Leaf x)    c = (Leaf c, c+1)
labAux1 (Branch t1 t2) c = let (t1', c1) = labAux1 t1 c
                             (t2', c2) = labAux1 t2 c1
                             in (Branch t1' t2', c2)

labelTree1    : Tree a → Tree Int
labelTree1 t = fst (labAux1 t 0)
```

Paneme tähele, et ...

- algul loendur algväärtustatakse;
- siis antakse ilmutatult alamarvutustele edasi;

Puu lehtede loendamine

Loendamine kasutades ilmutatud loendurit

```
labAux1      : Tree a → Int → (Tree Int, Int)
labAux1 (Leaf x)    c = (Leaf c, c+1)
labAux1 (Branch t1 t2) c = let (t1', c1) = labAux1 t1 c
                             (t2', c2) = labAux1 t2 c1
                             in (Branch t1' t2', c2)

labelTree1    : Tree a → Tree Int
labelTree1 t = fst (labAux1 t 0)
```

Paneme tähele, et ...

- algul loendur algväärtustatakse;
- siis antakse ilmutatult alamarvutustele edasi;
- ning teda sisuliselt kasutatakse vaid lehtedes.

Puu lehtede loendamine

Loendamine kasutades ilmutatud loendurit

```
labAux1                : Tree a → Int → (Tree Int, Int)
labAux1 (Leaf x)        c = (Leaf c, c+1)
labAux1 (Branch t1 t2) c = let (t1',c1) = labAux1 t1 c
                             (t2',c2) = labAux1 t2 c1
                             in (Branch t1' t2', c2)
```

```
labelTree1
labelTree1 t
```

Väga vigadealdis, kuna väga lihtne on edastada loenduri valet versiooni!

Paneme tähele, et ...

- algul loendur algväärtustatakse;
- siis antakse ilmutatult alamarvutustele edasi;
- ning teda sisuliselt kasutatakse vaid lehtedes.

Puu lehtede loendamine

Loendamine kasutades ilmutatud loendurit

```
labAux1                : Tree a → Int → (Tree Int, Int)
labAux1 (Leaf x)        c = (Leaf c, c+1)
labAux1 (Branch t1 t2) c = let (t1',c1) = labAux1 t1 c
                             (t2',c2) = labAux1 t2 c1
                             in (Branch t1' t2', c2)
```

```
labelTree1
labelTree1 t
```

Imperatiivsetes keeltes kasutaksime
globaalset muutujat!

Paneme tähele, et ...

- algul loendur algväärtustatakse;
- siis antakse ilmutatult alamarvutustele edasi;
- ning teda sisuliselt kasutatakse vaid lehtedes.

Puu lehtede loendamine

Loendamine "imperatiivses keeles"

```
global counter := 0
```

```
labelTree      : Tree a → Tree Int
labelTree (Leaf x) = t := Leaf counter
                  counter += 1
                  return t
labelTree (Branch t1 t2) = t1' := labelTree t1
                          t2' := labelTree t2
                          return (Branch t1' t2')
```

Puu lehtede loendamine

Üldistus

Olekust sõltuvaid arvutusi saame esitada **oleku teisendajatena**.

Loend

global

labelT

labelT

labelT

Puu lehtede loendamine

Üldistus

Olekust sõltuvaid arvutusi saame esitada **oleku teisendajatena**.

- S.o. funktsioonidena, mis seavad sisendolekule vastavusse väärtuse koos väljundolekuga.

Loend

global

labelT

labelT

labelT

Puu lehtede loendamine

Üldistus

Olekust sõltuvaid arvutusi saame esitada **oleku teisendajatena**.

- S.o. funktsioonidena, mis seavad sisendolekule vastavusse väärtuse koos väljundolekuga.
- Kui arvutus ei vaja olekut, siis on sisend- ja väljundolekud samad.

Loend

global

labelT

labelT

labelT

Puu lehtede loendamine

Üldistus

Olekust sõltuvaid arvutusi saame esitada **oleku teisendajatenä**.

- S.o. funktsioonidena, mis seavad sisendolekule vastavusse väärtuse koos väljundolekuga.
- Kui arvutus ei vaja olekut, siis on sisend- ja väljundolekud samad.
- Järjestikkompositsiooni korral antakse esimese alamarvutuse väljundolek teisele alamarvutusele sisendolekuks.

Puu lehtede loendamine

Üldistus

Olekust sõltuvaid arvutusi saame esitada **oleku teisendajatenä**.

- S.o. funktsioonidena, mis seavad sisendolekule vastavusse väärtuse koos väljundolekuga.
- Kui arvutus ei vaja olekut, siis on sisend- ja väljundolekud samad.
- Järjestikkompositsiooni korral antakse esimese alamarvutuse väljundolek teisele alamarvutusele sisendolekuks.
- Arvutused, mis ei tagasta väärtust kuid muudavad olekut, on **kõrvalefektid**.

Olekust sõltuvad arvutused

Olekuteiendajad

```
IntSt      :    Type → Type
IntSt a    =    Int → (a, Int)

sReturn    :    a → IntSt a
sReturn x  =    λs ⇒ (x, s)

inc        :    IntSt Int
inc        =    λs ⇒ (s, s+1)

sBind      :    IntSt a → (a → IntSt b) → IntSt b
m `sBind` f =    λs ⇒ let (x1, s1) = m s
                        (x2, s2) = f x1 s1
                        in (x2, s2)
```

Olekust sõltuvad arvutused

Loendamine kasutades olekuteisendajaid

```
labAux2      : Tree a → IntSt (Tree Int)
labAux2 (Leaf x) = inc `sBind` λi ⇒
                  sReturn (Leaf i)
labAux2 (Branch t1 t2) = labAux2 t1 `sBind` λt1' ⇒
                          labAux2 t2 `sBind` λt2' ⇒
                          sReturn (Branch t1' t2')

labelTree2    : Tree a → Tree Int
labelTree2 t = fst (labAux2 t 0)
```

Olekust sõltuvad arvutused

Loend Võrdlus

labAux2
labAux2
labAux2

Mõlemal juhul kasutatakse efektidega arvutuste selgemaks esitamiseks järgmisi abstraktsioone:

- efektidega arvutusi esitav tüüp;
- kombinaator, mis esitab puhtaid väärtusi;
- kombinaator arvutuste järjestikkomponeerimiseks.

labelT1
labelT1

Olekust sõltuvad arvutused

Loend Võrdlus

labAux2
labAux2
labAux2

Mõlemal juhul kasutatakse efektidega arvutuste selgemaks esitamiseks järgmisi abstraktsioone:

- efektidega arvutusi esitav tüüp;
- kombinaator, mis esitab puhtaid väärtusi;
- kombinaator arvutuste järjestikkomponeerimiseks.

labelT2
labelT2

Vastava struktuuriga arvutusi kutsutakse **monaadideks**.

Monaadidest Idrises

Idrisis in monaad on ühe muutujaga tüübipere, mille jaoks on defineeritud järgnevad funktsioonid:

```
interface Functor f where
  map : (a → b) → f a → f b
interface Functor f ⇒ Applicative f where
  pure  : a → f a
  (<*>) : f (a → b) → f a → f b
interface Applicative m ⇒ Monad m where
  (>=>) : m a → (a → m b) → m b
```

- Saab kasutada **do**-süntaksit:

```
main : IO ()
main = do
  x ← ranomRIO (1,10)
  print x
```