Idris

- Raamat: *Type-Driven development with Idris*, Edwin Brady
- Idrise programm sisaldab definitsioone;
  igal defineeritaval nimel on tüüp.
  - Näiteks, loome faili test.idr:

```
a : Double
a = 40.0

b : Double
b = 30.0

c : Double
c = sqrt (a*a + b*b)
```

- Definitsioone saab lugeda interaktiivsesse keskkonda:

```
> rlwrap idris2 test.idr
```

- ... ja siis väärtustada

```
*Main> c
50.0
```

Funktsioonide defineerimine

tüübideklaratsioon ⟶ `liida : Int → Int → Int` (*fun. nimi*: `liida`; *fun. tüüp*: `Int → Int → Int`)

funktsiooni definitsioon ⟶ `liida x y = x + y` (*vasak pool*: `liida x y`; *parem pool*: `x + y`)

Funktsioonide defineerimine

tüübideklaratsioon ⟶ `liida : Int → Int → Int` (*fun. nimi*: `liida`; *fun. tüüp*: `Int → Int → Int`)

funktsiooni definitsioon ⟶ `liida x y = x + y` (*vasak pool*: `liida x y`; *parem pool*: `x + y`)

- Saab osaliselt rakendada (e. anda vähem argumente).
- Defineerides ei pea kõiki argumente kirjutama.

```
liidaKaks : Int → Int
liidaKaks = liida 2
```

- Saab anda rohkem argumente

```
liidaKaks 3
```

Funktsiooni rakendus on vasakassotsiatiivne:

```
liida 2 3 == ((liida 2) 3)
```

Tüübid Idrises!

- Tüübituletus interaktiivselt:

```
Main> :t False
Prelude.False : Bool

Main> :t (++)
Prelude.List.++ : List a → List a → List a
Prelude.String.++ : String → String → String
```

Tüübid Idrises!

- Tüübituletus interaktiivselt:

```
Main> :t False
Prelude.False : Bool

Main> :t (++)
Prelude.List.++ : List a -> List a -> List a
Prelude.String.++ : String -> String -> String
```

- ... aga see pole alati intuitiivne:

```
Main> :t 2+2
2 + 2 : Integer
```

- Sellisel juhul saab kontrolli teha ka `the` funktsiooniga:

```
Main> the Double (2+2)
4.0
Main> the Int (2+2)
4
Main> the Bool (2+2)
Error: ...
```

- Tüübimuutujad – saab asendada suvalise teise tüübiga

```
Main> :t id
Prelude.id : a → a

Main> the Int (id 3)
3
Main> the (Bool → Bool) id
id
Main> the (Bool → Int) id
Error: ...
```

Konstruktorid ja mustrisobitus

Tõväärtused on Idrises tüübiga `Bool`.

- Konstruktoriteks `True` ja `False`.
- Tõeväärtuse info saab kätte mustrisobitusega:

```
f : Bool → Bool
f True  = False
f False = True
```

Väärtustest saab luua paare ja muid ennikuid.

- Näiteks: `(1,'a')`, `((1.1,8,'x'),False)`

Konstruktorid ja mustrisobitus

Tõväärtused on Idrises tüübiga `Bool`.

- Konstruktoriteks `True` ja `False`.
- Tõeväärtuse info saab kätte mustrisobitusega:

```
f : Bool → Bool
f True  = False
f False = True
```

Väärtustest saab luua paare ja muid ennikuid.

- Näiteks: `(1,'a')`, `((1.1,8,'x'),False)`
- Enniku tüüp konstrueeritakse komponentide tüüpidest:
  `(1, 'a', False) : (Int,Char,Bool)`

Konstruktorid ja mustrisobitus

Tõväärtused on Idrises tüübiga `Bool`.

- Konstruktoriteks `True` ja `False`.
- Tõeväärtuse info saab kätte mustrisobitusega:

```
f : Bool → Bool
f True  = False
f False = True
```

Väärtustest saab luua paare ja muid ennikuid.

- Näiteks: `(1,'a')`, `((1.1,8,'x'),False)`
- Enniku tüüp konstrueeritakse komponentide tüüpidest:
  `(1, 'a', False) : (Int,Char,Bool)`
- Info saab kätte mustrisobitusega:

```
f : (Int, Char, String) → Int
f (x,c,ys) = x + 1
```

Järjendid e. listid

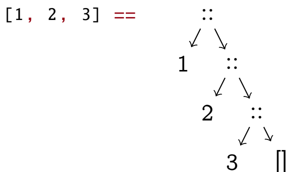
- Järjendi tüübiks on “List a”, kus “a” on elementide tüüp.
  - Näiteks: `List Int`, `List Char`, `List (List Float)`

Järjendid e. listid

- Järjendi tüübiks on “List a”, kus “a” on elementide tüüp.
  - Näiteks: `List Int`, `List Char`, `List (List Float)`
- Järjendeid saab kirjutada näiteks nii:
  - `[1, 2, 3], [3], []`
  - `[True, False, False], [False], []`

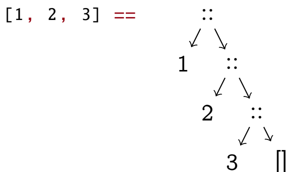
Järjendid e. listid

- Järjendi tüübiks on “List a”, kus “a” on elementide tüüp.
  - Näiteks: `List Int`, `List Char`, `List (List Float)`
- Järjendeid saab kirjutada näiteks nii:
  - `[1, 2, 3], [3], []`
  - `[True, False, False], [False], []`
- Idrise listid on arvuti mälus esindatud puudena:



Järjendid e. listid

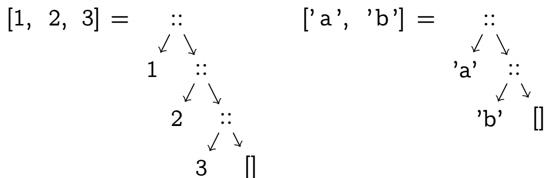
- Järjendi tüübiks on “List a”, kus “a” on elementide tüüp.
  - Näiteks: `List Int`, `List Char`, `List (List Float)`
- Järjendeid saab kirjutada näiteks nii:
  - `[1, 2, 3], [3], []`
  - `[True, False, False], [False], []`
- Idrise listid on arvuti mälus esindatud puudena:

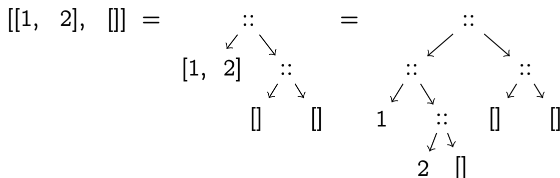


- Järjendi loomiseks on kaks konstruktorit, mis vastavad (list-tüüpi) puu tippudele.
  - `[] : List a`
  - `(::) : a → List a → List a`

Näide: `[3,2,1] == 3 :: (2 :: (1 :: []))`

Listi näited





Järjendid e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi. Näiteks

```
length : List a → Int
length []     = 0
length (x::xs) = 1 + length xs
```

või

```
length : List a → Int
length xs =
      case xs of
        []      ⇒ 0
        (x::xs) ⇒ 1 + length xs
```

Järjendid e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi. Näiteks

```
length : List a → Int
length []      = 0
length (x::xs) = 1 + length xs
```

  või

```
length : List a → Int
length xs =
      case xs of
        []      ⇒ 0
        (x::xs) ⇒ 1 + length xs
```

- Mustreid sobitatakse „ülevalt alla“, kuni esimese sobiva leidmiseni!
- Arvutuskäik:

```
length [1,2,3]
```

Järjendid e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi. Näiteks

```
length : List a → Int
length []      = 0
length (x::xs) = 1 + length xs
```

  või

```
length : List a → Int
length xs =
      case xs of
        []      ⇒ 0
        (x::xs) ⇒ 1 + length xs
```

- Mustreid sobitatakse „ülevalt alla“, kuni esimese sobiva leidmiseni!
- Arvutuskäik:

```
   length [1,2,3]
== length (1 :: (2 :: (3 :: [])))
```

Järjendid e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi. Näiteks

```
length : List a → Int
length []      = 0
length (x::xs) = 1 + length xs
```

  või

```
length : List a → Int
length xs =
      case xs of
        []      ⇒ 0
        (x::xs) ⇒ 1 + length xs
```

- Mustreid sobitatakse „ülevalt alla“, kuni esimese sobiva leidmiseni!
- Arvutuskäik:

```
length [1,2,3]
== length (1 :: (2 :: (3 :: [])))
⟹ 1 + length (2 :: (3 :: []))
```

Järjendid e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi. Näiteks

```
length : List a → Int
length []      = 0
length (x::xs) = 1 + length xs
```

  või

```
length : List a → Int
length xs =
       case xs of
         []      ⇒ 0
         (x::xs) ⇒ 1 + length xs
```

- Mustreid sobitatakse „ülevalt alla“, kuni esimese sobiva leidmiseni!
- Arvutuskäik:

```
length [1,2,3]
== length (1 :: (2 :: (3 :: [])))
⟹ 1 + length (2 :: (3 :: []))
⟹ 1 + (1 + length (3 :: []))
```

Järjendid e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi. Näiteks

```
length : List a → Int
length []      = 0
length (x::xs) = 1 + length xs
```

või

```
length : List a → Int
length xs =
      case xs of
        []      ⇒ 0
        (x::xs) ⇒ 1 + length xs
```

- Mustreid sobitatakse „ülevalt alla“, kuni esimese sobiva leidmiseni!
- Arvutuskäik:

```
length [1,2,3]
== length (1 :: (2 :: (3 :: [])))
⟹ 1 + length (2 :: (3 :: []))
⟹ 1 + (1 + length (3 :: []))
⟹ 1 + (1 + (1 + length []))
```

Järjendid e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi. Näiteks

```
length : List a → Int
length []       = 0
length (x::xs) = 1 + length xs
```

või

```
length : List a → Int
length xs =
      case xs of
        []       ⇒ 0
        (x::xs)  ⇒ 1 + length xs
```

- Mustreid sobitatakse „ülevalt alla“, kuni esimese sobiva leidmiseni!
- Arvutuskäik:

```
length [1,2,3]
== length (1 :: (2 :: (3 :: [])))
⟹ 1 + length (2 :: (3 :: []))
⟹ 1 + (1 + length (3 :: []))
⟹ 1 + (1 + (1 + length []))
⟹ 1 + (1 + (1 + 0))
```

Järjendid e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi. Näiteks

```
length : List a → Int
length []      = 0
length (x::xs) = 1 + length xs
```

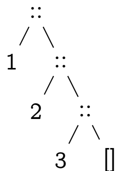
või

```
length : List a → Int
length xs =
      case xs of
        []      ⇒ 0
        (x::xs) ⇒ 1 + length xs
```

- Mustreid sobitatakse „ülevalt alla“, kuni esimese sobiva leidmiseni!
- Arvutuskäik:

```
length [1,2,3]
== length (1 :: (2 :: (3 :: [])))
⟹ 1 + length (2 :: (3 :: []))
⟹ 1 + (1 + length (3 :: []))
⟹ 1 + (1 + (1 + length []))
⟹ 1 + (1 + (1 + 0))
⟹ 3
```

Tähelepanekud



- Elemendi lisamine listi algusse kiire!
  - kiire == konstante aeg ja mälu
  - Põhjendus: argumente ei ole vaja kopeerida.
- Viimase elemendi selekteerimine aeglane!
  - aeglane == lineaarselt listi pikkusega
- Järjendi lõppu lisamine aeglane!
  - aeglane == aeg ja mälu lineaarne listi pikkusega (tuleb kopeerida)

Lambda-arvutus

- λ-termide süntaks:

$$
\begin{array}{rcll}
e & ::= & x & \text{muutuja} \\
  & | & (e_1\ e_2) & \text{aplikatsioon} \\
  & | & (\lambda x.\ e) & \text{abstraktsioon}
\end{array}
$$

- Sulgudest hoidumine:

$$
\begin{array}{lcl}
e_1\ e_2\ \ldots\ e_n & \equiv & ((\ldots(e_1\ e_2)\ldots)e_n) \\
\lambda x.\ e_1\ e_2\ \ldots\ e_n & \equiv & (\lambda x.\ (e_1\ e_2\ \ldots\ e_n)) \\
\lambda x_1\ x_2\ \ldots\ x_n.\ e & \equiv & (\lambda x_1.\ (\lambda x_2.\ (\ldots(\lambda x_n.\ e)\ldots)))
\end{array}
$$

- Näited:

$$\lambda x.\ x \qquad\qquad ((\lambda x.\ (\lambda f.\ f\ x))\ y)(\lambda z.\ z)$$

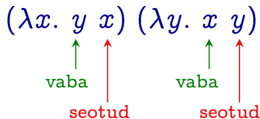
Lambda-arvutus

- Puhas λ-arvutus „räägib“ ainult funktsioonidest.
- Arve ja teisi andmetüüpe eraldi ei ole, kuna neid saab defineerida λ-termidena.
- Mugavuse pärast kasutame siiski arve ja binaarseid operaatoreid nagu nad oleksid keelde sisseehitatud.
- Samuti kasutame makro-definitsioone (peavad olema mitterekursiivsed).
- Näited:

$$\begin{array}{lcl}
\mathsf{add} & \equiv & \lambda x\, y.\ x + y \\
\mathsf{dbl} & \equiv & \lambda x.\ 2 * x \\
\mathsf{I} & \equiv & \lambda x.\ x \\
\mathsf{K} & \equiv & \lambda x\, y.\ x \\
\mathsf{S} & \equiv & \lambda f\, g\, x.\ f\, x\, (g\, x)
\end{array}$$

Vabad ja seotud muutujad

- Muutuja x on *vaba* λ-termis e (so. $x \in \mathrm{FV}(e)$), kui ta ei asu sümboli λx mõjupiirkonnas.
- Vastasel korral on x *seotud*.



Vabad ja seotud muutujad

- Vabade muutujate induktiivne definitsioon:

$$
\begin{array}{lcl}
\mathrm{FV}(x) & = & \{x\} \\
\mathrm{FV}(e_1\ e_2) & = & \mathrm{FV}(e_1) \bigcup \mathrm{FV}(e_2) \\
\mathrm{FV}(\lambda x.\ e) & = & \mathrm{FV}(e) - \{x\}
\end{array}
$$

- Ilma vabade muutujateta λ-termid on kinnised.
- Näited:

$$
\begin{array}{lcl}
\mathrm{FV}(\lambda x\, y.\ x\ \ y) & = & \emptyset \\
\mathrm{FV}(\lambda x.\ (\lambda y.x)\,(\lambda z.y)) & = & \{y\}
\end{array}
$$