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 $\longrightarrow$ $\overbrace{\texttt{liida}}^{\textit{fun. nimi}} \texttt{ :} \overbrace{\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}}^{\textit{fun. tüüp}}$

funktsiooni definitsioon $\longrightarrow$ $\underbrace{\texttt{liida x y}}_{\textit{vasak pool}} \texttt{ = } \underbrace{\texttt{x + y}}_{\textit{parem pool}}$

Funktsioonide defineerimine

tüübideklaratsioon ⟶ $\overbrace{\texttt{liida}}^{\textit{fun. nimi}} : \overbrace{\texttt{Int} \to \texttt{Int} \to \texttt{Int}}^{\textit{fun. tüüp}}$

funktsiooni definitsioon ⟶ $\underbrace{\texttt{liida x y}}_{\textit{vasak pool}} = \underbrace{\texttt{x + y}}_{\textit{parem pool}}$

- Funktsiooni rakendus on vasakassotsiatiivne:

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

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

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

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
fromInteger 2 + fromInteger 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:

```
[1, 2, 3] ==      ::
                 /  \
               1     ::
                    /  \
                  2     ::
                       /  \
                     3     []
```

Järjendid e. listid

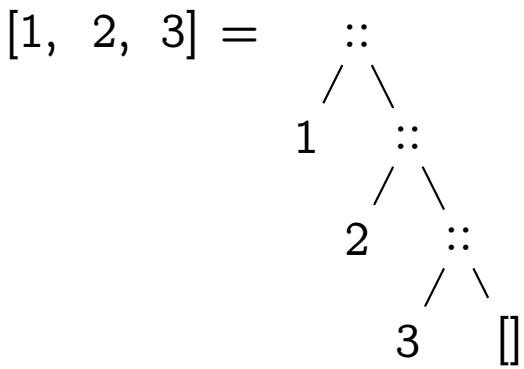
- Järjendi tüübiks on “List a”, kus “a” on elementide tüüp.
  - Näiteks: `List Int`, `List Char`, `List (List Float)`
- Järjendeid saab kirjutada näiteks nii:
  - `[1, 2, 3], [3], []`
  - `[True, False, False], [False], []`
- Idrise listid on arvuti mälus esindatud puudena:

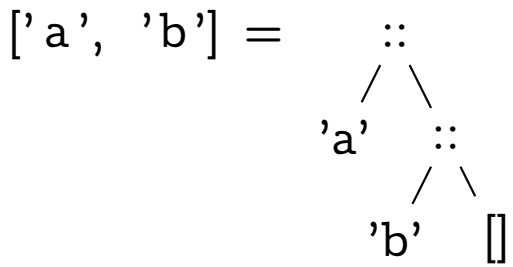
```
[1, 2, 3] ==      ::
                 /  \
               1     ::
                    /  \
                  2     ::
                       /  \
                     3     []
```

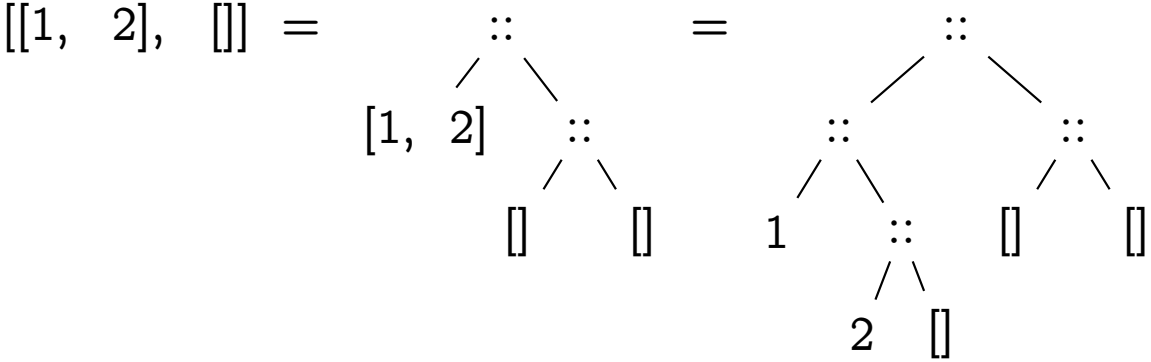
- Järjendi loomiseks on kaks konstruktorit, mis vastavad (list-tüüpi) puu tippudele.
  - `[] : List a`
  - `(::) : a → List a → List a`

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
```

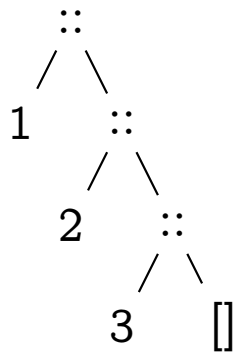
või

```
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)

Tüübid on kasulikud töövahendid, mitte nuhtlus!

- Nagu Javas on levinud „testipõhine arendus“ on Idrises „tüübipõhine arendus“.

- Kõigepealt kirjuta tüübid ja implementeeri kasutades auke.

```
fact2 : Int → Int
fact2 0 = ?esimene
fact2 n = ?teine
```

- Seejärel täita kõik augud (näiteks) alustades keerulisemast. Vajadusel lisada väiksemaid auke.

```
fact2 : Int → Int
fact2 0 = ?esimene
fact2 n = n * fact2 ?argument
```

- Aukude konteksti saab lasta välja trükkida.

```
Main> :m
2 holes: Main.argument, Main.esimene

Main> :t argument
   n : Int
--------------------------------
argument : Int
```