Programmeerimise paradigmad

Saab jagada kaheks:

- imperatiivsed e. mis operatsioone teha
  - Lisaks jaotatakse: protseduuraalsed ja objektorienteeritud
- deklaratiivsed e. lahenduse (tõe) kirjeldamine
  - Lisaks jaotatakse: funktsionaalne ja loogiline

Erinevused:

- Imperatiivne kasvas protsessori käsustiku abstaheerimisest.
- Deklaratiivne kasvas välja matemaatikast.

Matemaatik/loogik tahab mõelda kõrgemal tasemel:

- algebralisteststruktuuridest nagu rühmad, monoidid, ring jne.
- funktsioonidest, hulkadest ja relatsioonidest

Funktsionaalne keel

- Matemaatiline funktsioon: tulemus sõltub ainult funktsiooni parameetritest
- Ilma funktsioonideta (meetodite, protseduurideta) ei saa!

  Java: `Math.max(4, 7)`
  Python: `max(4,7)`
  Idris: `max 4 7`

- Sõna „funktsioon“ asemel kasutatakse ka *abstraktsioon*. (Kuna enamasti ei huvita, kuidas `max` on implementeeritud.)
- FP eelistab olemasolevate vahendite üldistamist.

  Näiteks: Samamoodi, kuidas programmeerimiskeeles saab käsitleda andmeid, peab saama käsitleda ka alamprogramme.

  $\Longrightarrow$ `max 3` saab võtta kõigist listi `[2,3,4]` elementidest (3. loeng)

  `map (max 3) [2,3,4]` == `[max 3 2, max 3 3, max 3 4]` == `[3,3,4]`

Puhas keel

- Puhas ehk kõrvaltoimevaba — funktsiooni kutse tulemus sõltub ainult parameetrite väärtustest.

  $\Longrightarrow$ iga funktsiooni saab testida teistest eraldi

- Mittepuhtaid funktsioone (nagu juhuarvude genereerimine) pole võimalik funktsioonidena defineerida

  $\Longrightarrow$ on vaja kasutada näiteks *monaade*

Tugevalt ja staatiliselt tüübitud

- Mida rohkem programmeerimiskeel lubab seda parem?
  - Ei! Näiteks, Portable Document Format (1993) on suuresti vähendatud võimalustega PostScript (1982).
- Piiramise üks võimalus on *tüübisüsteemiga*.
  - Tüüp kirjeldab (ja kitsendab) programmi alamosa ja tüübikontroll otsustab, kas neid osi tohib niimoodi kombineerida.
  - Näiteks: Kui `a` on tüüpi `Int` siis seda saab anda `max` argumendiks.
  - Testimise vajadus mingil määral väiksem?
- Staatiline kontroll — kompileerimise või interpreteerimise käigus
- Tugevalt tüübitud — väljastatakse kohe veateade.

Idris (slaid 1st loengust)

- 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:

```
> idris2 test.idr
```

- ... ja siis väärtustada

```
*Main> c
50.0
```

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: ...
```

Idris väärtused

- Arvutüübid: `Int`, `Integer`, `Nat`, `Double`

```
Main> the Int 10
10
Main> the Double 10
10.0
the Int 1.0
Error: When unifying Double and Int.
```

  - avaldis `the <tüüp> <väärtus>` kitsendab väärtust vastava tüübiga

- Aritmeetika: `+`, `-`, `*`, `/`, `` `div` `` ja teisendus `cast`

```
Main> the Double (2*3+4)
10.0
Main> the Int (cast (the Double 1.0))
1
```

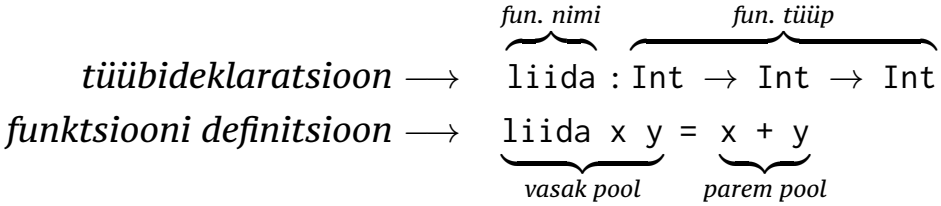
- Tähed `Char` ja sõned `String`

```
Main> the String "tekst"
"tekst"
Main> the Char 'A'
'A'
```

- Tõeväärtused `Bool` ja loogika

```
Main> True && False
False
Main> True || False
True
Main> 3 > 2
True
Main> 100 == 99
False
Main> 100 ≠ 99
True
```

Funktsioonide defineerimine



- 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
```

Infix operaatorid

- Infix operaatorid koosnevad sümbolitest: `:+-*\/=.?|&><!@$%~^#`
  - Näiteks: `5+2`, `2*pi*r*r`, `Float → Int`
- Infix operaatori kasutamiseks prefix-vormis tuleb see panna sulgudesse. Näiteks: `3 + 4 == (+) 3 4`
- Infix operaatoril on prioriteet (i.k. precedence)
  - Näiteks: `3*4+5 == (3*4)+5`, kuna $*$ on tasemel 9 ja $+$ tasemel 8
- Infix operaator võib olla parem- või vasakassotsiatiivne:
  - näiteks, `a && b && c == a&&(b&&c)`, `2 - 3 - 4 == (2-3)-4`
- Neid seatakse nn. fixity deklaratsioonidega:

```
infixl 8 +, -
infixl 9 *, /
infixr 5 &&
```

- Fixity deklaratsioone saab vaadata:
  - Näiteks käsuga “:doc (+)”
- Prefix operaatoreid saab rakendada infix-selt tagurpidi-ülakomade (`) vahel
  - Näiteks: `5 `div` 2`

Ennikud ja *Unit* tüüp

- 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:

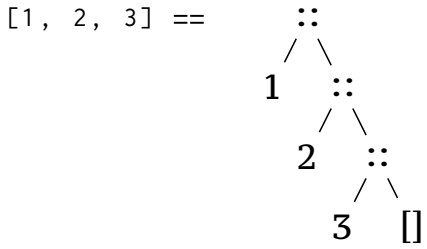
```
f : (Int, Char, String) → Int
f (x,c,ys) = x + 1
```

Kõige triviaalsem väärtus Idrises on `()`.

- Selle tüüp on samuti `()` ehk `() : ()`
- Kasutatakse juhtudel, kui pole vaja informatsiooni edastada.
- Paljudes keeltes kasutatakse selle asemel tüüpi `void`

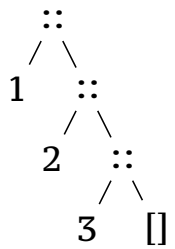
Järjendid e. listid

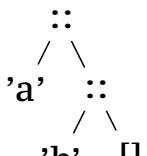
- Järjendi tüübiks on “List a”, kus “a” on elementide tüüp.
  - Näiteks: `List Int`, `List Char`, `List (List Float)`
- Järjendeid saab kirjutada näiteks nii:
  - `[1, 2, 3], [3], []`
  - `[True, False, False], [False], []`
- Idrise listid on arvuti mälus esindatud puudena:

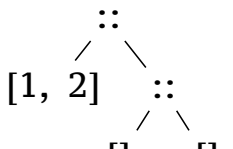


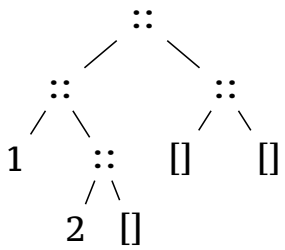
- Järjendi loomiseks on kaks konstruktorit, mis vastavad (list-tüüpi) puu tippudele.
  - `[] : List a`
  - `(::) : a → List a → List a`
  - Näide: `[3,2,1] == 3 :: (2 :: (1 :: []))`

Listi näited

[1, 2, 3] = 

['a', 'b'] = 

[[1, 2], []] = 

= 

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
```

Listide näited

- Korrutis

```
product : List Int → Int
product []     = 1
product (x::xs) = x * product xs
```

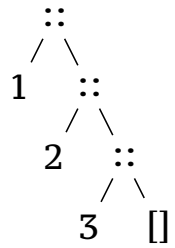
- Listi ümberpööramine

```
reverse : List a → List a
reverse []     = []
reverse (x::xs) = reverse xs ++ [x]
```

- Järjest asetsevate võrdsete elementide loendamine

```
countEq : List Int → Int
countEq (x::y::xs) =
  if x==y then
    1 + countEq (y::xs)
  else
    countEq (y::xs)
countEq _  =  0
```

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)