Substitutsioon

- λ-arvutuse alusoperatsiooniks on formaalsete parameetrite asendamine tegelike argumentidega.
- Avaldise $(\lambda x.\, e_1)\, e_2$ väärtustamiseks tuleb termis e_1 asendada muutuja x kõik vabad esinemised termiga e_2.
- Substitutsiooni tähistame $e_1[x \mapsto e_2]$.
- Peab vältima vabade muutujate "vangistamist".
- Näited:

$$\begin{array}{lcl} (\lambda x.\, y\, x)[y \mapsto \lambda z.\, z] & = & \lambda x.\, (\lambda z.\, z)\, x \\ (\lambda x.\, y\, x)[x \mapsto \lambda z.\, z] & = & \lambda x.\, y\, x \\ (\lambda x.\, y\, x)[y \mapsto \lambda z.\, x] & \neq & \lambda x.\, (\lambda z.\, x)\, x \end{array}$$

Substitutsioon

Substitutsiooni definitsioon:

$$
\begin{aligned}
y[x \mapsto e] &= \begin{cases} e & \text{if } x = y \\ y & \text{otherwise} \end{cases} \\
(e_1\ e_2)[x \mapsto e] &= (e_1[x \mapsto e])\ (e_2[x \mapsto e]) \\
(\lambda y.\, e_1)[x \mapsto e] &= \begin{cases} \lambda y.\, e_1 & \text{if } x = y \\ \lambda y.\, e_1[x \mapsto e] & \text{if } y \notin \mathrm{FV}(e) \\ \lambda z.\, e_1[y \mapsto z][x \mapsto e] & \text{otherwise} \end{cases}
\end{aligned}
$$

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
```

Anonüümsed funktsioonid

- λ-arvutus: $\lambda\ x\ y\ z.\ e$
- Idrises: `\ a, b, c => e`
  (slaididel \ ja => asemel kirjutame λ ja $\Rightarrow$)

- Idris kasutab seda ka siseselt.
  Defineerides funktsiooni

```
f : ...
f x y z = ...
```

  teisendab kompilaator koodi selliseks

```
f : ...
f = λ x, y, z ⇒ ...
```

Kõrgemat järku funktsioon

... on funktsioon, mis võtab argumendiks või tagastab funktsiooni.

FP keskne mõte: arvutada saab ka arvutustega (e. funktsioonidega).

Näiteks map:

```
map : (a → b) → List a → List b
map f []      = []
map f (x::xs) = f x :: map f xs
```

Funktsiooni map illustreerib järgmine võrdus:

$$\texttt{map}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

Näide:

```
inverses : List Double → List Double
inverses xs = map (λ x ⇒ 1 / x) xs
```

```
Main> inverses [1,2,4,8]
[1.0, 0.5, 0.25, 0.125]
```

Kõrgemat järku funktsioon: foldr

```
foldr : (a → b → b) → b → List a → b
foldr f b []      = b
foldr f b (x::xs) = f x (foldr f b xs)
```

Funktsiooni foldr illustreerib järgmine võrdus:

$$\texttt{foldr}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = x_1 + (x_2 + (\ldots + (x_n + b)))$$

Funktsiooni map saab defineerida läbi foldr-i

```
map : (a → b) → List a → List b
map f xs = foldr g [] xs
    where g : a → List b → List b
          g x ys = f x :: ys
```

ehk

$$\begin{aligned}\texttt{foldr}\ g\ []\ [x_1, x_2, \ldots, x_n] &= x_1\texttt{`}g\texttt{`}(x_2\texttt{`}g\texttt{`}(\ldots\texttt{`}g\texttt{`}(x_n\texttt{`}g\texttt{`}[]))) = \\ &= f\ x_1 :: f\ x_2 :: \ldots :: f\ x_n :: [] = \\ &= [f\ x_1, f\ x_2, \ldots, f\ x_n]\end{aligned}$$

Kõrgemat järku funktsioon: foldl

```
foldl : (b → a → b) → b → List a → b
foldl f b []     = b
foldl f b (x::xs) = foldl f (f b x) xs
```

Funktsiooni foldl illustreerib järgmine võrdus:

$$\texttt{foldl}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = (((b + x_1) + x_2) + \ldots) + x_n$$

Funktsiooni reverse saab defineerida läbi foldl-i

```
reverse : List a → List a
reverse xs = foldl g [] xs
   where g : List a → a → List a
         g x y = y :: x
```

ehk

$$\begin{aligned}\texttt{foldl}\ g\ []\ [x_1, x_2, \ldots, x_n] &= ((([]\ `g`\ x_1)\ `g`\ x_2)\ `g`\ \ldots)\ `g`\ x_n = \\ &= x_n :: \ldots :: x_2 :: x_1 :: [] = \\ &= [x_n, \ldots, x_2, x_1]\end{aligned}$$

Kõrgemat järku funktsioonid kui disainimustrid

```
f : List Double → Double
f (x::xs) = x + f xs
f []      = 0
```

vs.

```
f = foldr (+) 0
```

- Lihtsate funktsioonide puhul on ka rekursiivne lahendus selge.
- Pikema ülesande puhul on hea eraldada listi töötlemine.
- S.t. mitte *bittide*-tasemel vaid abstraktsemalt.
  Kumb definitsioon on selgem:

```
f : List Double → Double
f (0::_)  = 0
f (x::xs) = x + f xs
f []      = 0
```

  või

```
f : List Double → Double
f = sum ∘ takeWhile (≠0)
```

Funktsiooni osaline rakendamine

- Selle asemel, et siduda kõik argumendid muutujatega

```
uusFunktsioon x y z = olemasolevFunktsioon (x+1) y z
```

  saame Idrises võtta osa argumente ja tagastada funktsiooni

```
uusFunktsioon x = olemasolevFunktsioon (x+1)
```

- Pane tähele, kuidas töötavad koos aplikatsiooni vasakassotsiatiivsus ja funktsiooni tüübi paremassotsiatiivsus.

```
f : a → (b → (c → d))
f   x :   b → (c → d)
(f  x)    y :   c → d
((f x)    y)    z :  d
```

ehk

```
f : a → b → c → d
f   x :  b → c → d
f   x    y :  c → d
f   x    y    z :  d
```

- funktsioone saame `(.)`-ga komponeerida, ehk:

```
f xs = sum (takeWhile (≠0) xs)
```

  asemel

```
f = sum ∘ takeWhile (≠0)
```

- Funktsiooni `(a,b) → c` *karritud* kuju on `a → b → c`.
- Kirjuta funktsioone nii, et neid oleks võimalik osaliselt rakendada!