Programmeerimiskeeled

Kursuse läbinu:

- omandab ülevaatlikud teadmised erinevatest programmeerimise paradigmadest;
- oskab lahendada lihtsaid programmeerimise ülesandeid funktsionaalse programmeerimise abil, sealhulgas saab aru ja oskab kasutada kõrgemat järku polümorfseid funktsioone;
- oskab mitme-paradigma keeles kasutada koos funktsionaalse, imperatiivse ja objekt-orienteeritud paradigma tehnikaid.

Administrativia

- Loengud
  - Kalmer Apinis (kalmera@ut.ee)
- Praksid
  - inf1 – E14-L2-206 – Simmo Saan
  - inf2 – T14-L2-202 – Mirjam Iher
- Loe pikemalt: courses.cs.ut.ee/2019/PK

Hinde kujunemine

- Protsendiskaala 0p .. 100p
- 0p..50p $\rightarrow$ F, 51p..60p $\rightarrow$ E, 61p..70p $\rightarrow$ D, ...
  - ümardame üles (70.1 -> 71 -> C)
- Eksam (40p)
- Loengutestid (10$*$1p, moodle)
- Kodutööd (8$*$2p+2$*$7p = 30p, moodle)
- Baastestid (2$*$10p, loengute lõpul, $<$20p $\rightarrow$ hinne F)
- Kodutööde, loengutestide asemel soovitatav teha oma projekt.
  - Programmeerida midagi Haskellis ja/või Scalas.

Materjalid

1. konspekt: `http://kodu.ut.ee/~kalmera/haskell/`

2. “Learn You a Haskell for Great Good!” (2011)
   - \+ Väga hea raamat intuitsiooni saamiseks
   - \- Pole põhjalik ega täielik

3. “Real World Haskell” (2008)
   - \+ Põhjalik ja selge!
   - \- Mahukas!

4. “Sissejuhatus Funktsionaalsesse Programmeerimisse” (2010)
   - \+ Eestikeelne
   - \- Osati liiga detailne, osati liiga piiratud

(Haskelli on vahepeal uuendatud! Raamatud on kohati vananenud!)

Miks uued programmeerimiskeeled?

- Keel C on vähemalt sama võimas, ükskõik mis teine programmeerimiskeel! (järeldub Church-Turingi teesist)
- Vastus: komponentide ja algoritmide taaskasutus! Keegi ei kirjuta programme “nullist”! Mida lihtsam on erinevaid teeke omavahel ühendada, seda parem.
- Vastus: korrektsuse tõestamise võimalus. Meil on palju koodi aga me ei saa seda usaldada.
- otsime abi teoreetikutelt . . .

Motivatsioon

Taaskasutusel on abiks

- paindlikud modulaarsuse võimalused

Korrektsuse tõestamisel on abiks

- Keel on täpselt defineeritud.
- Range tüübisüsteem.
- Ilmutatud viidatavus ehk saame ignoreerida ebaolulist.
  - Puhas FP – funktsiooni tagastusväärtus sõltub ainul argumentidest.
- Baaskonstruktsioonide hulk on väike.

$\Longrightarrow$ Õpime Funktsionaalset Programmeerimist!

Funktsionaalne Programmeerimine (FP)

- FP motivatsioon
  - Probleem: võimalused on välja arendamata ja üksteisega konfliktis
  - Võimaluste lisamise asemel peaks üldistama olemasolevaid!
  - Aritmeetilised operatsioonid (funktsioonid) on möödapääsmatud!
- Tüüpimise motivatsioon
  - Probleem: Harva(?) esinevad vead erijuhtudest.
  - Võimaldab vältida vigu ja anda edasi kompileerimisaegset infot.
- Puhta FP motivatsioon
  - Selleks et uurida keerulisi struktuure, peab olema võimalus neid keelata!
  - Näide: hajus ja paralleelne arvutus
- Laisa FP motivatsioon
  - Teoreetiliselt paindlikum kui agar väärtustamine (FP magistriaine).
  - Näide: lõpmatute listidega arvutamine

$\Longrightarrow$ Õpime Haskelli!

Loe lisaks: RWH, eessõna

Haskell

- Haskell erineb Pythonist ja Javast väga palju. Varasemad oskused Pythonist või Javast ei ole otse rakendtavad!
- Seetõttu on Haskelli targem õppida kui matemaatikat/algebrat, mitte kui programmeerimist.
- Väga tähtis on, et te töötaksite juba algusest peale kaasa. Alustame väga lihtsate programmidega ja jõuame alles kursuse lõpuks mingile arvestatavale tasemele.

Haskell

- Haskelli programm koosneb definitsioonidest
  - Näiteks, loome faili test.hs:

```
a = 40.0
b = 30.0
c = sqrt (a^2 + b^2)
```

- Definitsioone saab lugeda interaktiivsesse keskkonda:

```
> stack ghci test.hs
```

- ... ja siis käivitada

```
*Main> c
50.0
```

- Mida see programm arvutab?
- Mis juhtub, kui muuta definitsioonide järjekorda?

Loe lisaks: RWH, peatükk 1; LYaH, peatükk 2, Starting Out

Tüübid

- Igal defineeritaval nimel on tüüp. Enamasti ei pea tüüpe juurde kirjutama, kuid dokumenteerimise eesmärgil on seda siiski soovitatav aegajalt teha.
- Näiteks:

```
a, b, c :: Float
a = 40
b = 30
c = sqrt (a^2 + b^2)
```

  või

```
a :: Float
a = 40
b :: Float
b = 30
c :: Float
c = sqrt (a^2 + b^2)
```

Loe lisaks: LYaH, peatükk 3, Types and Typeclasses, Believe the type

Funktsioonide defineerimine

- Funktsioone defineeritakse samuti võrdusmärgiga. Võetakse abiks formaalsed parameetrid.
  - näide:

```
pyth :: Float -> Float -> Float
pyth x y = sqrt (x^2 + y^2)
```

- Haskell järgib matemaatilist notatsiooni, kuid on erandeid:
  - "$f(x, y)$" asemel kirjutame "$f\ x\ y$"
- Funktsiooni tüüp on "α -> β", kus α on sisendi tüüp ja β tulemuse tüüp

Faktoriaali näide

- If-avaldisel põhinev faktoriaal

```
fact1 :: Int -> Int
fact1 n = if n==0 then 1 else n * fact1 (n-1)
```

- `fact1` väärtustamine

```
fact1 2
==> if 2 == 0 then 1 else 2 * fact1 (2-1)
==> if 2 == 0 then 1 else 2 * fact1 1
==> 2 * fact1 1
==> 2 * (if 1 == 0 then 1 else 1 * fact1 (1-1))
==> 2 * (if 1 == 0 then 1 else 1 * fact1 0)
==> 2 * (1 * fact1 0)
==> 2 * (1 * (if 0 == 0 then 1 else 0 * fact1 (-1)))
==> 2 * (1 * 1)
==> 2
```

- Näidiste sobitamisel põhinev faktoriaal

```
fact2 :: Int -> Int
fact2 0 = 1
fact2 n = n * fact2 (n-1)
```

- Valvuritel põhinev faktoriaal ...

```
fact3 :: Int -> Int
fact3 n
  | n==0      = 1
  | otherwise = n * fact3 (n-1)
```

- Valvuritel põhinev faktoriaal mis ei lähe tsükklisse

```
fact4 :: Int -> Int
fact4 n
  | n == 0 = 1
  | n >= 1 = n * fact4 (n-1)
```

- Akumulaatorit kasutav, where-konstruktsiooniga

```
fact5 :: Int -> Int
fact5 n = fact5' 1 n
       where fact5' a 0 = a
             fact5' a m = fact5' (a*m) (m-1)
```

- Akumulaatorit kasutav, let-konstruktsiooniga

```
fact6 :: Integer -> Integer
fact6 n =
  let fact6' = \ a n -> case n of
                          0 -> a
                          _ -> fact6' (a*n) (n-1)
  in fact6' 1 n
```

- product funktsiooni ja jada kasutav faktoriaal

```
fact7 :: Int -> Int
fact7 n = product [1..n]
```

Loe lisaks: LYaH, peatükk 4

Vindi ülekeeramine ...

- Püsipunktikombinaatori abil defineeritud faktoriaal

```
fact9 :: Integer -> Integer
fact9 = fixedPt f
  where f g 0     = 1
        f g n     = n * g (n-1)
        fixedPt f = g where g = f g
```

- “The Evolution of a Haskell Programmer”
  - https://www.willamette.edu/~fruehr/haskell/evolution.html

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 välja protsessori käsustiku abstaheerimisest.
- Deklaratiivne kasvas välja matemaatikast.

Matemaatik/loogik tahab mõelda

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

Paljud protsessori instruktsioonid pole lihtsalt modelleeritavad – keerulised erijuhud.

Keerulised erijuhud – näide

Olgu meil Java programm, milles on selline koodirida:

```
int x = Math.abs(y);
```

Kas x on positiivne (, null) või negatiivne?

```
Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE
```

Funktsionaalsed keeled

FP lühiajalugu

- Kombinaatorloogika (M. Schönfinkel 1924, H. Curry 1927)
- Lambda-arvutus (A. Church 1936)
- Lisp (J. McCarthy 1958)
- ML ja polümorfne tüübisüsteem (R. Milner 1978)
- Hope, Sasl, Miranda, . . . (1980 – 85)
- Haskell (1988)

Haskelli ajalugu

- 1987 loodi Haskell'i komitee
- 1988 esimene keelekirjeldus (v. 1.0)
- 1999 Haskell98 (Standard Haskell)
- 2010 Haskell 2010
- 2016 GHC 8 (suur standardteegi puhastus)

Haskell on . . .

- tugevalt ning staatiliselt tüübitud,
- laisk ja puhas
- funktsionaalne keel.

Funktsionaalne keel

- Ilma funktsioonideta (meetodite, protseduurideta) ei saa!

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

- Sõna ”funktsioon” asemel kasutatakse ka *abstraktsioon*.

- FP eelistab olemasolevate vahendite üldistust.
  Näiteks *if*-lause saame ise defineerida:

```
kui True t f  = t
kui False t f = f
```

- Turingi masin: programm on staatiline ja andmed dünaamilised
- FP: Samamoodi, kuidas programmeerimiskeeles saab käsitleda andmeid, peab saama käsitleda ka alamprogramme.
- $\Longrightarrow$ kõrgemat järku funktsioonid

  `map (max 3) [2,3,4]` $\rightsquigarrow$ `[max 3 2, max 3 3, max 3 4]` $\rightsquigarrow$ `[3,3,4]`
- Kõrgemat järku funktsioonid võimaldavad meil programmi osadest mõelda kõrgemal abstraktsiooni tasemel ehk siis suuremate tükkidena.

Tugevalt ja staatiliselt tüübitud

- Mida rohkem programmeerimiskeel lubab seda parem?
  - PostScript (1982) vs. Portable Document Format (1993)
- Piiramise üks võimalus on *tüübisüsteemiga*.
- Staatiline tüübikontroll — kompileerimise (või interpreteerimise) käigus
- Tugevalt tüübitud keele puhul väljastatakse kohe veateade.
- Nõrgalt tüübitud keele võib püüda näiteks sõnet arvuks teisendada.
- Testimise vajadus mingil määral väiksem?

Puhas keel

- Puhas — kõrvaltoimevaba ehk 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 *monaade*

Laisk keel

Olgu meil selline kood:

```
tagastaViis x = 5
topelt x = x+x
```

- *agaras keeles* arvutatakse parameeter enne funktsiooni kutset

  `tagastaViis (1+1)` ⇝ `tagastaViis 2` ⇝ `5`

  `topelt (1+1)` ⇝ `topelt 2` ⇝ `2+2` ⇝ `4`

- *laisas keeles* tehakse funktsioonikutse asendus enne

  `tagastaViis (1+1)` ⇝ `5`

  `topelt (1+1)` ⇝ `(1+1)+(1+1)` ⇝ `2+(1+1)` ⇝ `2+2` ⇝ `4`

- Haskell teeb tegelikult hoopis nii

  `topelt (1+1)` ⇝ `x+x where x=1+1` ⇝ `x+x where x=2` == `2+2` ⇝ `4`

Infix Operaatorid I

- Infix operaatorid (näiteks +) ja tüübid (näiteks ->) kirjutatakse argumentide vahele, mitte argumentide ette.
  - Näiteks: `5 + 2`, `2*pi*r^2`, `Float -> Int`

- Infixoperaatori kasutamiseks prefix-vormis tuleb see panna sulgudesse. Näiteks: `3 + 4 == (+) 3 4`

- Infixoperaatoreid saab ise juurde defineerida:

```
(@) :: Int -> Int -> Int
x @ y = 2*x+y
```

  või

```
(@) :: Int -> Int -> Int
(@) x y = 2*x+y
```

- Mõni kord eeldame, et funktsioonirakendused on prefixsed.

Infix Operaatorid II

- Infixoperaatoril on prioriteet (i.k. precedence)
  - Näiteks: `3*4+5 == (3*4)+5`, kuna `*` on tasemel 7 ja `+` tasemel 6
- Infixoperaatoril võib olla assotsiatiivsus
  - näiteks, `2**3**4 == 2**(3**4)`, `2 - 3 - 4 == (2-3)-4`
- Neid seatakse nn. fixity deklaratsioonidega:

```
infixl 7 *
infixl 6 +
infixl 6 -
infixr 8 **
```

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

Eeldefineeritud operaatorite fixity (Haskell 98)

Prec-edence	Left associative operators	Non-associative operators	Right associative operators
9	!!		.
8			^, ^^, **
7	*, /, ‘div‘, ‘mod‘, ‘rem‘, ‘quot‘		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=, ‘elem‘, ‘notElem‘	
3			&&
2			||
1	>>, >>=		
0			$, $!, ‘seq‘

(Funktsioonirakenduse *pretsedence* on 10.)