Funkcionális programozás
1. előadás

MÁRTON Gyöngyvér

Sapientia Egyetem, Matematika-Informatika Tanszék
Marosvásárhely, Románia
mgyongyi@ms.sapientia.ro

2023, tavaszi félév

Miről lesz szó?

- követelmények
- könyvészet
- programozási módszerek összehasonlítása: imperatív, funkcionális, logikai
- Haskell, történelmi háttér, telepítés, használat, bevezető fogalmak, az első program
- Haskell alaptípusok
- feladatok: faktoriális számítás

Követelmények, vizsgajegy

- vizsgajegy = (laborjegy1 + laborjegy2)/2, ahol a második jegy és az átlag el kell érje a 4.5-t
- a 8. és 14. oktatási héten kerül sor laborórán egy-egy laborvizsgára, ezek képezik a laborjegy1, illetve laborjegy2 értékeket
- a laborjegy2 kiváltható egy kisebb alkalmazás elkészítésével, ezt a szándékot a 9.ik hétig kell jelezni, úgy hogy elküldenek emailben, egy féloldalas bemutató anyagot (mit fog csinálni az alkalmazás)
- az alkalmazást egyénileg kell elkészíteni, és a kész alkalmazást a 14. héten kell elküldeni, majd a vizsgaidőszak első hetében, egy 10 perces bemutatóban el kell magyarázni. Az alkalmazásban, megjegyzésekkel kell minden függvényt ellátni: mit csinál, mi a szerepe, hol hívódik meg és hogyan.

Könyvészet


Simon Thompson, Haskell, The Craft of Functional Programming, Addison-Wesley.


Graham Hutton, Programming in Haskell, Cambridge University Press, 2007.


Nyékyné Gaizler Judit, Programozási nyelvek, Kiskapu Budapest, 2003.


Márton Gyöngyvér, *Funkcionális programozás - Haskell alapismeretek*, Scientia, Kolozsvár, 2021.


http://learnyouahaskell.com/


http://learn.hfm.io/


Prognyelvek portál: http://nyelvek.inf.elte.hu/leirasok/Haskell/index.php

Programozási módszerek: imperatív programozás

- programozási nyelvek: assembly, Java, C, C++, stb.,
- alapvezérlési szerkezete: a feltételes utasítás, a ciklus utasítás
- jellemzők:
  - felszólító mód: parancsok, utasítások sorozata, amelyek tetszőleges módon változtathatják az eltárolt adat (változó) értékét,
  - a lényeg az algoritmus megtalálása (hogyan oldjuk meg a feladatot)
  - pontosan követhetők a végrehajtott lépések.

Programozási módszerek: funkcionális programozás

- programozási nyelvek: Lisp, Haskell, Clean,
- alapeszköze: a függvénykiértékelés, a rekurzív függvényhívás,
- a változó: nincs előző értéket megsemmisítő értékadásra lehetőség,
- a programkód tömör, rövid,
- kezdete az 1930-as években megjelenő lambda kalkulusra vezethető vissza,
- funkcionális programozási stílusra imperatív nyelvek esetében is lehetőség van.

Programozási módszerek: logikai programozás

- programozási nyelvek: Prolog, SQL,
- alapeszköze: a reláció-tények, szabályok megadása, ezek rekurzív alkalmazása,
- gyökerei hasonlóan a lambda kalkulusban találhatóak,
- legfontosabb alkalmazási területe a mesterséges intelligencia.

Példaprogram

A faktoriáls függvény C nyelvben megírt változatai:

1. változat:

```
int fakt1(int n) {
    int i, res;
    if (n < 0) return -1;
    if (n == 0) return 1;
    for (i = 1, res = 1; i <= n; i++)
        res *= i;
    return res;
}
```

2. változat:

```
int fakt2(int n) {
    if (n < 0) return -1;
    if (n == 0) return 1;
    else return n * fakt2(n - 1);
}
```

3. változat:

```
int fakt3(int res, int n) {
    if (n < 0) return -1;
    if (n == 0) return res;
    return fakt3(n * res, n - 1);
}
```

a függvényhívások:

```
X = fakt1 (10);
X = fakt2 (10);
X = fakt3 (1, 10);
```

Példaprogram

A faktoriáls függvény Haskell nyelvben megírt változtai:

1. változat:

```
fakt1 :: Int -> Int
fakt1 0 = 1
fakt1 n = n * fakt1 (n-1)
```

2. változat:

```
fakt2 :: Int -> Int
fakt2 n
    | n < 0 = -1
    | n == 0 = 1
    | otherwise = n * fakt2 (n-1)
```

3. változat:

```
fakt3 :: Int -> Int -> Int
fakt3 res n
    | n < 0 = -1
    | n == 0 = res
    | otherwise = fakt3 (n * res) (n - 1)
```

Haskell, történelmi háttér

- 1930-ban Alonzo Church kifejlesztette a lambda kalkulust, matematikai függvények vizsgálata céljából,
- 1950-ben megjelenik a LISP (LISt Processor), az első funkcionális programozási nyelv,
- 1970-ben megjelenik az FP (Functional Programming), amely először vezeti be a magasabb rendű függvényeket,
- 1987-ben egy nemzetközi bizottság egy új, lusta kiértékelési stratégiával működő funkcionális programozási nyelv fejlesztése mellett dönt,
- a nyelv neve Haskell lesz, Haskell Curry (amerikai matematikus) neve után,
- 2003-ban jelenik meg az első igazán megbízható Haskell verzió.

A Haskell programozási nyelv

- a Haskell telepítése a *https://www.haskell.org/ghcup/* oldalon keresztül történik
- Windows alá, a Haskell telepítése előtt, fel lehet tenni a `chocolatey`-t: *https://chocolatey.org/install*. Vigyázzunk, hogy a parancssort, azaz a PowerShell-t admin módban indítsuk!
- ha sikerült telepíteni a chocolatey-t, akkor: `> choco install haskell-dev`
- kódszerkesztőnek használhatjuk a Visual Studio Code-t (VSC): *https://code.visualstudio.com/download*
- VSC-ben egy terminal elindításával, a `ghci` parancsot kiadva, lehet elindítani a Haskell-t, ami után a terminálban megjelenik a `Prelude>` prompt
- VSC-ben további kiterjesztések telepíthetők, amelyek a kódolást, fordítást, futtatást segítik:
  - Haskell language support powered by the Haskell Language Server: hibakezelések, típus kezelések és sok más nyelvi elem automatikus támogatása
  - Haskell Syntax Highlighting: szintaxis elemek kiemelése (kulcsszavak, behúzások, zárójelek, stb.)
  - Code Runner: különböző programozási nyelveken írt kódok futtatását teszi lehetővé

A Haskell programozási nyelv

- több implementációja is ismert:
  - Hugs, amely egy interpreter, leginkább oktatásban használják
  - GHC (Glasgow Haskell Compiler), valós alkalmazások fejlesztéséhez használják, natív kódra kompilál, biztosítja a párhuzamos végrehajtást, a debuggolást, a hatékonyság elemzését
- a GHC komponensei:
  - ghci: az interaktív interpreter és debugger
  - ghc: a kompilátor, ami a natív kódot generálja
  - runghc: a Haskell programokat futtatja, mint szkripteket, anélkül hogy előbb kompilálni kellene őket

A Haskell programozási nyelv

- sikeres telepítés után mentsük le a korábban megadott faktorialis függvényeket egy állományba, legyen ez `eload1.hs` (végezhetjük ezt egy akármilyen szövegszerkesztővel)
- az op rendszerből indítsunk el egy command ablakot, válasszuk ki azt a könyvtárat ahová az `eload1.hs` állományt mentettük majd adjuk ki a következő parancsot: `> ghci eload1.hs`
- ennek hatására elindul a Haskell és bármelyik függvény meghívható lesz a `eload1.hs` állományból pl:

```
> fakt1 10
> fakt2 10
> fakt3 1 10
```

- a Haskell-ből a `> :q` paranccsal lépjünk ki

A Haskell programozási nyelv

- ha futtatható kódot akarunk akkor az `eload1.hs` állományhoz írjuk még hozzá a következő kódsorokat

```
main :: IO ()
main = do
    putStr "fakt1 10: "
    print (fakt1 10)
    putStr "fakt2 10: "
    print (fakt2 10)
    putStr "Fakt3 10: "
    print (fakt3 1 10)
    getLine
    return ()
```

- mentsük el, majd ha beírjuk egy terminálba a következő parancsot, akkor létrejön a futtaható állomány

```
> ghc eload1.hs
> .\eload1.hs
```

- kiadhatjuk > `runghc eload1` parancsot, ekkor fordítás nélkül fogja a Haskell futtatni a kódunkat

A Haskell programozási nyelv

- a Haskell indítása történhet tehát parancssorból, vagy a Visual Studio Code-t használva, vagy más környezetet
- a `Prelude>` prompt megjelenése azt jelzi, hogy sikeresen betöltődött a standard könyvtár,
- a `Prelude>` prompt után parancsok, kifejezések stb. adhatóak meg
- további könyvtárcsomagok is betölthetőek
- a prompt után kifejezések írhatóak, amelyeket a Haskell azonnal kiértékel, tehát használható számológépként
- a Haskell kifejezéseket, függvényeket, a kódsorokat hs kiterjesztésű állományokba is írhatjuk
- egy állományba több Haskell függvényt is írhatunk, amelyek fordítás után külön-külön kiértékelhetők
- Haskell állományokban számos beépített, vagy utólag telepített könyvtárcsomag függvényeit is használhatjuk
- Haskell projekteket is írhatunk, amelyek több állományból állhatnak

Haskell, mint számológép

```
Prelude> 10 + 3
13

Prelude> (+) 3 + 10
13

Prelude> 7.5 * 4.3
32.25

Prelude> (*) 7.5 4.3
32.25
```

```
Prelude> div 10 3
3

Prelude> 10 `div` 3
3

Prelude> mod 13 7
6

Prelude> 13 `mod` 7
6
```

A hatványozáshoz, aszerint, hogy egész, vagy valós számokon végezzük, más-más műveleti jelet használunk, ily módon a négyzetgyök, a köbgyök stb. értékét könyvtárfüggvény nélkül is meg tudjuk határozni:

```
Prelude> 10 ** 3
1000.0

Prelude> 10 ^ 3
1000
```

```
Prelude> 2 ** 0.5
1.4142135623730951

Prelude> 2 ** (1/3)
1.2599210498948732
```

Haskell, mint számológép

Az `Integer` típus bevezetésével a Haskell képes tetszőlegesen nagy számokat kezelni, például a hatványozó operátor, anélkül, hogy bármiféle könyvtárcsomagot importálnánk, helyesen határozza meg a következő számítások eredményét:

```
Prelude> 2 ^ 100
1267650600228229401496703205376

Prelude> 2 ** 100
1.2676506002282294e30
```

Ha a kifejezések megadása során hibát követünk el, akkor azokat a Haskell nem fogja kiértékelni, helyette hibaüzenetet ad. Például a következő kifejezést a Haskell nem tudja kiértékelni, mert a ^ operátor csak egész típusú értékekre alkalmazható:

```
Prelude>  2 ^ 0.5
... error:
Could not deduce (Integral b0) arising from a use of '^'
```

Haskell, mint számológép

```
> "Hello " ++ "vilag" ++ "!"
"Hello vilag!"

> length "Hello vilag!"
12

> length [15, 8, 10, 3, 4]
5

> sum [1, 2, 3, 4, 5]
15

> sum [1..10]
55
```

Haskell, mint számológép

```
> madarak = ["rigo", "cinke", "harkaly"]
madarak :: [[Char]]

> "pinty" : madarak
["pinty", "rigo", "cinke", "harkaly"]

> ("Hello " ++ "vilag!") == "Hello vilag!"
True

> "Hello " ++ "vilag!" == "Hello vilag!"
True

> import Data.List
> sort [1, 6, 5, -10, 7]
[-10, 1, 5, 6, 7]
```

Haskell, az első lépések

- a prompt után függvényeket is definiálhatunk, majd meghívhatjuk:

```
> terulet r = if r < 0 then error "Rossz benmenet!" else r * r * pi

> terulet 5
78.53981633974483
```

- a függvényt állományba is írhatjuk, amelynek kiterjesztése hs kell legyen
- legyen az első Haskell állományunk `elso.hs`, amelybe írjuk be:

```
terulet :: Double -> Double
terulet r = if r < 0 then error "Rossz benmenet!" else r * r * pi
```

Haskell, az első lépések

Ha a VSC-ben történt a kódolás, és ha az állomány neve: `elso.hs`, akkor a következőképpen lehet meghívni a `terulet` függvényt:

- válasszuk ki a `File/Open Folder`-el a munkamappát,
- egy `Terminal` ablakba indítsuk el a `gchi`-vel a Haskell-t,
- írjuk be a prompt után `:l elso.hs`,
- ekkor minden függvény, amely az állományban van kiértékelhetővé válik, hívjuk meg tehát a függvényt, pl: `> terulet 10`
- az állomány betöltéséhez még a következőképpen is eljárhatunk: írjuk be a prompt után `:l`, majd válasszuk ki a `Terminal/Run Active File` menüpontot,
- ha változtatást végzünk a kódsoron, akkor le kell menteni az állományt, majd a `:r` Haskell paranccsal újra be lehet tölteni a módosított file-t.

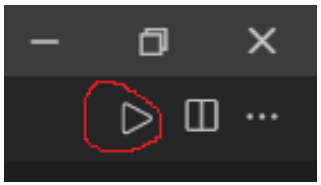
Haskell, az első lépések

A Haskell állományt a következőképpen is futtathatjuk:

- az állományt egészítsük ki egy `main` függvénnyel, pl:

```
main :: IO ()
main = do
    print (terulet 20.75)
```

- ha sikeresen telepítettük a `Code Runner` Haskell kiterjesztést, akkor a programunk futtatható lesz a szerkesztő jobb felső sarkában található gomb lenyomásával:



- egy állomány természetesen nem tartalmazhat két `main` függvényt
- mielőtt futtatnánk az állományt ne felejtsük el lementeni

GHC parancsok

lehet rövidített formában is használni őket, a parancs kezdőbetűjével:

- `:load fnev.hs` az fnev.hs nevű állomány betöltése,
- `:reload fnev.hs` az fnev.hs nevű állomány újrabetöltése, rövidítve: `:r`
- `:type kif` a kif kifejezése típusának a lekérdezése, rövidítve: `:t kif`
- `:?` az összes GHC parancs lekérdezése,
- `:quit` kilépés a GHC-ből,
- `:set +t` a kiértékelés után a kifejezés típusa is megjelenik
- `:unset +t` az előző beállítás visszavonása
- `:set +s` a kiértékelés után megjelenik az eltelt idő és a lefoglalt bájtok száma
- `:!cd` az aktuális könyvtár (directory) állapot lekérdezése
- `:cd C:\Diak` könyvtár (directory) változtatás
- ...

Alaptípusok

- Bool: logikai típus, két értékkel: False, True,
- Char: egyetlen (Unicode) karakter eltárolására alkalmas, idézőjel közé kell írni az értéket:

  `'A', '\n', '+'`
- String: karakterláncok kezelésére alkalmas, macskakörröm közé kell írni az értékeket: `"Hello Haskell", "10 * 0 = 0"`,
- Int: rögzített pontosságú egész számok, -2^{31} közötti 2^{31} értékek kezelésére alkalmas,
- Integer: tetszőleges pontosságú egész számok kezelésére alkalmas. Az Int és Integer közötti választást elsősorban a hatékonyság alapján kell eldönteni,
- Float, Double: valós számok kezelésére alkalmas. A Double 64-bites lebegőpontos ábrázolású, használata ajánlott, ellentétben a Float-tal, aminek használata kevésbé ajánlott.

Típusosztályok

Egy függvény argumentumai nem csak egy adott típushoz tartozó értékek lehetnek. A Haskell bevezeti a "típusváltozó" fogalmát, amit azt jelenti hogy a függvényargumentumok specifikációjakor típusosztályokat is megadhatunk. Jelölésükre az angol ábécé kis betűit használjuk.

A `terulet` függvény szignatúráját a következőképpen módosíthatjuk:

```
terulet :: (Ord t, Floating t) => t -> t
terulet r = if r < 0 then error "Rossz benmenet!" else r * r * pi
```

A `sort` függvény szignatúráját a következőképpen kérdezhetjük le:

```
> import Data.List
> :t sort
sort :: Ord a => [a] -> [a]
```

Ez azt jelenti, hogy a `sort` függvény minden olyan típusú adatra meghívható, amely az `Ord` típusosztályba tartozik. Például egész számok rendezése mellett karakterláncokat is rendezhetünk:

```
> sort ["pinty", "rigo", "cinke", "harkaly"]
```