Programske paradigme i jezici

Funkcijsko programiranje i programski jezik Haskell

Jan Šnajder

Sveučilište u Zagrebu
Fakultet elektrotehnike i računarstva

svibanj 2012.

v2.1

Sadržaj

Sadržaj

Deklarativna vs. imperativna paradigma

Programski jezici: imperativni i deklarativni.

Imperativni program izvodi se (slijednim) izvođenjem naredbi, dok se deklarativni program izvodi vrednovanjem izraza.

Deklarativni jezici mogu biti logički (izrazi su relacije) ili funkcijski (izrazi su funkcije).

Imperativni: C, C++, C#, Java, ...

Deklarativni: SML, Ocaml, Haskell, F#, Erlang, Prolog, ...

Ove paradigme temelje se na različitim teorijskim modelima, odnosno modelima izračunljivosti (engl. *computational models*).

Imperativna paradigma temelji se na modelu Turingovog stroja, logička na formalnoj logici, a funkcijska na lambda-računu.

Implicitno vs. eksplicitno stanje (1)

Imperativni jezici imaju implicitno stanje i izvođenje se svodi na postepeno mijenjanje tog stanja izvođenjem pojedinačnih naredbi.

Izmjena implicitnog stanja tipično se ostvaruje naredbom pridruživanja.

Npr. izračun faktorijele broja `x`:

```
fact(x) {
  n = x;
  a = 1;
  while (n > 0) {
    a = a*n;
    n = n-1;
  }
  return a;
}
```

Implicitno vs. eksplicitno stanje (2)

Deklarativni jezici nemaju implicitno stanje.

Izvođenje programa svodi se na evaluaciju izraza i to bez stanja (engl. *state-less*).

Petlja se ostvaruje rekurzijom:

```
fact x = if x==0 then 1
         else x*fact(x-1)
```

Izračun sa stanjem može se ostvariti uporabom eksplicitnog stanja koje onda moramo povlačiti naokolo:

```
fact x = fact' x 1
  where fact' n a = if n>0 then fact' (n-1) (a*n)
                    else a
```

Referencijalna prozirnost (1)

Deklarativni jezici imaju svojstvo tzv. referencijalne prozirnosti: istovrijedni izrazi zamjenjivi su u svim kontekstima.

```
... x+x ...
where x = fact 5
```

Svako pojavljivanje izraza `x` u programu možemo zamijeniti s `fact 5`.

To općenito nije slučaj kod imperativnih jezika, budući da se vrijednost varijable `x` nakon inicializacije može naknadno promijeniti.

Kod imperativnih jezika u obzir moramo uzeti redoslijed izvođenja naredbi i oni stoga nisu referencijalno prozirni.

Popratni efekt

Popratni efekt (engl. *side effect*): svaka promjena implicitnog stanja koja narušava referencijalnu prozirnost programa.

```
foo(x) {
  y = 0;
  return 2*x;
}
```

```
...
y = 2
if (foo(y)==foo(y))
  then ... else ...
```

```
...
y = 5;
x = foo(2);
z = x + y
```

```
...
y = 5;
z = foo(2) + y;
...
```

Funkcija `foo` izvodi popratni efekt (`y=0`) i time narušava referencijalnu prozirnost svih dijelova programa iz kojih se poziva!

Referencijalna prozirnost (2)

Ako u potpunosti želimo zadržati referencijalnu prozirnost, ne smijemo dopustiti nikakve popratne efekte!

Zauzvrat dobivamo programe koji su:

1. formalno koncizni
2. pogodni za formalnu verifikaciju
3. manje podložni pogreškama
4. pogodni za paralelizaciju

Referencijalna prozirnost (3)

Očuvanje referencijalne prozirnosti u praksi je problematično:
neki algoritmi inherentno su temeljeni na stanju, a neke funkcije postoje samo zbog svojih popratnih efekata (`fprintf`, `fscanf`, `rand`).

Zbog praktičnosti, većina deklarativnih jezika dopušta kontrolirane popratne efekte (engl. *declarative in style*).

S druge strane, čisto deklarativni jezici (engl. *purely declarative*) ne dopuštaju baš nikakve popratne efekte.

Takvi jezici koriste dodatne mehanizme kako bi omogućili izračun sa stanjem i istovremeno zadržali referencijalnu prozirnost.

Funkcijsko programiranje

Stil programiranja u kojemu se sve svodi na izračunavanje funkcija.

Funkcija je *građanin prvoga reda*: ravnopravna je ostalim tipovima podataka te može biti povratna vrijednost ili argument druge funkcije – funkcije višeg reda (engl. *higher-order function*).

```
dvaput f x = f (f x)
```

Ostale karakteristike suvremenih funkcijskih jezika su:

- lijena evaluacija (engl. *lazy evaluation*)
- podatkovna apstrakcija
- podudaranje uzorka (engl. *pattern matching*)
- zaključivanje o tipovima (engl. *type inference*)

Funkcijski programski jezici

Funkcijski programski jezici *podržavaju* i *ohrabruju* funkcijski stil programiranja.

Najpoznatiji funkcijski jezici:
Standard ML (SML), Clean, Miranda, Haskell, ...

Moderni višeparadigmatski jezici uključuju funkcijsku paradigmu:
Common Lisp, OCaml, Scala, Python, Ruby, F#, R, Mathematica, JavaScript, Clojure, ...

Neki imperativni jezici eksplicitno podržavaju neke funkcijske koncepte (C#, Java), dok su u drugima oni ostvarivi (C, C++).

Premda mnogi imperativni jezici podržavaju funkcijske koncepte, čisto funkcijski podskup takvih jezika najčešće je vrlo slab!

Povijesni razvoj

- 1924. – kombinatorna logika (Schönfinkel & Curry)
- 1930. – lambda-račun (Church)
- 1958. – jezik Lisp (McCarthy)
- 1964. – jezik ISWIM i apstraktni stroj SECD (Landin)
- 1977. – jezik FP (Backus)
- 1977. – jezik ML (Milner) i Hindley-Milnerov algoritam
- '80 – jezici Miranda (Turner), Hope, Lazy ML, Standard ML, ...
- 1987. – međunarodni odbor započinje s dizajnom novog jezika
- 1990. – odbor objavljuje specifikaciju Haskell 1.0
- 2002. – jezik F#

Sadržaj

Lambda-račun

Lambda-račun (Church, 1930.) je formalan model izračunljivosti funkcija.

Matematika funkciju definira kao relaciju između domene i kodomene: funkcija je "crna kutija" i ne zanima nas kako se izračunava.

Lambda-račun funkcije tretira kao izraze koji se postepeno transformiraju do rješenja – funkcija zapravo definira algoritam.

Lambda-račun smatra se prvim funkcijskim jezikom (premda to nije bila prvotna ideja).

Svi moderni funkcijski jezici zapravo su (samo) "uljepšane" varijante lambda-računa.

Lambda-izraz (1)

Osnovni koncept je lambda-izraz ili funkcijska apstrakcija:

$$\lambda\, \mathit{varijabla}\, .\, \mathit{tijelo}$$

- $\lambda x\, .\, x + 1$
- $\lambda x\, .\, 2 * x + 3$
- $\lambda x\, .\, x$

Lambda-izraz definira argumente i tijelo funkcije, ali ne i njezino ime, pa se također zove anonimna funkcija.

Lambda-izraz (2)

Lambda-izrazi mogu se primjenjivati (aplicirati) na druge izraze:

$$(\lambda\, \textit{varijabla}\, .\, \textit{tijelo})\ \textit{izraz}$$

- $(\lambda x\, .\, x + 1)\, 5$
- $(\lambda x\, .\, x)\, 1$
- $(\lambda x\, .\, 2 * x + 3)\, \big((\lambda x\, .\, x + 1)\, 5\big)$

Aplikacija lambda-izraza odgovara pozivu funkcije.

β-redukcija

Izračunu povratne vrijednosti funkcije odgovara β-redukcija:

$$(\lambda\, \mathit{varijabla}\,.\,\mathit{tijelo})\;\mathit{izraz} \;\rightarrow_\beta\; \mathit{tijelo}[\mathit{varijabla} := \mathit{izraz}]$$

β-redukcija u tijelu lambda-izraza formalni argument zamijenjuje aktualnim te vraća tijelo funkcije.

- $(\lambda x\,.\,x+1)\,5 \;\rightarrow_\beta\; (x+1)[x := 5] = 5+1$
- $(\lambda x\,.\,(2*x+3))\,((\lambda x\,.\,x+1)\,5) \;\rightarrow_\beta\; (\lambda x\,.\,2*x+3)\,(5+1)$

Višestrukom primjenom β-redukcije ($\twoheadrightarrow_\beta$) dolazimo do krajnjeg rezultata:

- $(\lambda x\,.\,x+1)\,5 \;\twoheadrightarrow_\beta\; 6$
- $(\lambda x\,.\,(2*x+3))\,((\lambda x\,.\,x+1)\,5) \;\twoheadrightarrow_\beta\; 15$

Funkcije višeg reda

Funkcije višeg reda prirođene su lambda-računu.

Funkcija prima funkciju: aplikacija u tijelu lambda-izraza.

- $\lambda x\,.\,(x\,2) + 1$
- $(\lambda x\,.\,(x\,2) + 1)\,(\lambda x\,.\,x + 1) \;\rightarrow_\beta\; (\lambda x\,.\,x + 1)\,2 + 1 \;\twoheadrightarrow_\beta\; 4$
- $(\lambda x\,.\,(x\,2) + 1)\,(\lambda x\,.\,x) \;\twoheadrightarrow_\beta\; 3$

Funkcija vraća funkciju: manji lambda-izraz u tijelu većeg lambda-izraza.

- $\lambda x\,.\,(\lambda y\,.\,2 * y + x)$
- $(\lambda x\,.\,(\lambda y\,.\,2 * y + x))\,5 \;\rightarrow_\beta\; \lambda y\,.\,2 * y + 5$

Ovakva je uporaba česta, pa se uvodi pokrata:
umjesto $\lambda x\,.\,(\lambda y\,.\,2 * y + x)$ skraćeno pišemo $\lambda xy\,.\,2 * y + x$

Funkcije s više argumenata

Lambda-izrazi ograničeni su na samo jedan argument!

Kako u lambda-računu definirati funkciju $f(x, y) = x + y$?

Schönfinkel (1924): bilo koja funkcija s više argumenata može se definirati pomoću funkcije od samo jednog argumenta.

Curryjev postupak (engl. *currying*) ili Schönfinkelizacija

Funkcija oblika

$$f(x_1, x_2, \ldots, x_n) = \textit{tijelo}$$

u lambda-računu definira se kao

$$\lambda x_1 \,.\, (\lambda x_2 \,.\, (\cdots (\lambda x_n \,.\, \textit{tijelo}) \cdots))$$

odnosno skraćeno kao

$$\lambda x_1 x_2 \cdots x_n \,.\, \textit{tijelo}$$

Npr. $((\lambda xy \,.\, x + y)\, 5)\, 3 \;\rightarrow_\beta\; (\lambda y \,.\, 5 + y)\, 3 \twoheadrightarrow_\beta\; 8$

Normalan oblik

Višestrukom β-redukcijom postepeno izračunavamo vrijednost izraza, a zaustavljamo se tek kada daljnja β-redukcija više nije moguća.

Tako dobiveni lambda-izraz naziva se normalan oblik (engl. *normal form*) i odgovara vrijednosti polaznog izraza.

Neki izrazi nemaju normalan oblik. Npr.: $(\lambda x\,.\,x\,x)\,(\lambda x\,.\,x\,x)$

Bilo bi dobro da je normalan oblik jedinstven te da ga, ako postoji, možemo izvesti. To nije sasvim očito budući da postoje izrazi za koje postoji više (divergentnih) mogućnosti β-redukcije.

- $(\lambda x\,.\,2 * x)\,((\lambda x\,.\,x + 1)\,3) \;\rightarrow_\beta\; (\lambda x\,.\,2 * x)\,(3 + 1)$
- $(\lambda x\,.\,2 * x)\,((\lambda x\,.\,x + 1)\,3) \;\rightarrow_\beta\; 2 * ((\lambda x\,.\,x + 1)\,3)$

Church-Rosserov teorem

Church-Rosserov teorem

Ako se lambda-izraz E može reducirati na dva (različita) izraza M i N, onda postoji treći izraz Z do kojeg se može doći i iz M i iz N.

$$(E \twoheadrightarrow_\beta M \ \wedge \ E \twoheadrightarrow_\beta N) \ \rightarrow \ \exists Z(M \twoheadrightarrow_\beta Z \ \wedge \ N \twoheadrightarrow_\beta Z)$$

Svojstvo Church-Rossera ili svojstvo stjecišta (engl. *confluence*).

Korolar: svaki lambda-izraz ima najviše jedan normalan oblik, tj. normalan oblik je jedinstven.

Dakle nije bitno kako smo došli do normalnog oblika – postupak redukcije na normalan oblik je deterministič an.

Ostaje pitanje: koji je redoslijed redukcija optimalan?

Teorem standardizacije

Postoje dvije osnovne strategije β-redukcije:

1. Aplikativan poredak (engl. *applicative-order*):
   β-redukcije uvijek reduciraju najdesniji unutarnji izraz
   Npr. $(\lambda x\,.\,2 * x)\,(3+1) \;\to_\beta\; (\lambda x\,.\,2 * x)\,4 \;\twoheadrightarrow_\beta\; 8$
   Ovo odgovara evaluaciji call-by-value.
2. Normalan poredak (engl. *normal-order*):
   β-redukcije uvijek reduciraju najljeviji vanjski izraz
   Npr. $(\lambda x\,.\,2 * x)\,(3+1) \;\to_\beta\; 2 * (3+1) \;\twoheadrightarrow_\beta\; 8$
   Odgovara evaluaciji call-by-name odnosno call-by-need.

Teorem standardizacije

Ako je Z normalan oblik lambda-izraza E, onda postoji slijed redukcija u normalnom poretku koji vodi od E do Z.

Lijena evaluacija

Normalan poredak redukcije omogućava lijenu evaluaciju (engl. *lazy evaluation*): izrazi se evaluiraju tek onda kada je to potrebno.

Lijena evaluacija izbjegava nepotrebne izračune:

- AP: $(\lambda x\,.\,1)\,(8 * 7) \;\to_\beta\; (\lambda x\,.\,1)\,56 \to_\beta 1$
- NP: $(\lambda x\,.\,1)\,(8 * 7) \;\to_\beta\; 1$

Lijena evaluacija jamči završetak izračuna kad god je on moguć:

- AP: $(\lambda x\,.\,1)\,((\lambda x\,.\,x\,x)(\lambda x\,.\,x\,x)) \;\to_\beta\; \cdots\; \bot$ (nije izračunljivo)
- NP: $(\lambda x\,.\,1)\,((\lambda x\,.\,x\,x)(\lambda x\,.\,x\,x)) \;\to_\beta 1$

Lijena evaluacija vodi računa o tome da se izračuni ne ponavljaju (tehnikom redukcije grafa pokazivačima se povezuju identični izrazi).

- $(\lambda x\,.\,x + x)(8 * 7) \;\to_\beta\; \bullet \underbrace{+}_{8*7} \bullet \;\to_\beta\; 56 + 56$

Ekspresivnost lambda-računa

Sukladno Church-Turingovoj tezi, funkcije koje su efektivno izračunljive upravo su one koje se mogu definirati lambda-računom.

U smislu ekspresivnosti, lambda-račun ekvivalentan je Turingovom stroju.

Moderni funkcijski programski jezici temelje se na tipiziranom lambda-računu (lambda račun proširen s tipovima).

Sadržaj

Razvoj Haskella

- 1987. – međunarodni odbor počinje s dizajnom novog, zajedničkog funkcijskog jezika
- 1990. – odbor objavljuje specifikaciju Haskell 1.0. Jezik je nazvan u čast logičara Haskella Curryja
- 1990–1997. – četiri izmjene standarda
- 1998. – odbor objavljuje Haskell 98 Report, trenutno važeći standard
- 2003. – Haskell report
- 2009. – odbor nastavlja rad na novom standardu nazvanom Haskell Prime

Karakteristike (1)

Čisto funkcijski jezik: popratni efekti nisu mogući i programi zato zadržavaju referencijalnu prozirnost.

Koristi lijenu evaluaciju: izbjegavaju se nepotrebni izračuni i ohrabruje se modularizacija.

Programi su koncizni i redovito 2 do 10 puta kraći od istih programa u drugim programskim jezicima.

Moćan tipski sustav (engl. *type system*) kojim se velik broj pogrešaka može otkriti prije samog izvođenja programa.

Tipski sustav istovremeno podržava polimorfizam (parametarsko višeobličje) i preopterećenje (engl. *overloading*).

Podržava sažet zapis listi (engl. *list comprehension*): kompaktan i ekspresivan način definiranja listi kao osnovnih podatkovnih struktura.

Karakteristike (2)

Naglašava uporabu rekurzije te je nadopunjuje mehanizmima podudaranja uzoraka i tzv. čuvarima (engl. *guards*).

Funkcije višeg reda omogućavaju visoku razinu apstrakcije i korištenje funkcijskih oblikovnih obrazaca.

Monadičko programiranje omogućava da se (ponekad neizbježni) popratni efekti izvode bez narušavanja referencijalne prozirnosti.

Razrađena biblioteka standardnih funkcija (*Standard Library*) i dodatnih modula (*Hackage*).

Haskell razvija i podržava velika zajednica znanstvenika i entuzijasta, a standardizaciju provodi međunarodni odbor (*Haskell Committee*).

Primjer koda u Haskellu

```
import qualified Data.Set as S

bfSearch :: (Ord a) => a -> (a -> [a]) -> (a -> Bool) -> [a]
bfSearch s0 succ goal = step [(s0,[])] S.empty
  where step [] v = []
        step (x@(s,ss):xs) v
          | goal s        = reverse $ s:ss
          | S.member s v = step xs v
          | otherwise     = step (xs ++ expand x)
                                 (S.insert s v)
        expand (s,ss) = [(t,s:ss) | t <- succ s]

showStep :: State -> State -> String
showStep s1 s2 = show s1 ++ "␣-->␣" ++ show s2

solve :: IO ()
solve = putStr . unlines $ zipWith showStep ss (tail ss)
  where ss = bfSearch initState nextState isGoal
```

Interpreteri i prevodioci

- HUGS – interaktivni interpreter
  `http://www.haskell.org/hugs`
  Pogodan za edukacijske svrhe i razvoj prototipa
- GHC (Glasgow Haskell Compiler) – interaktivni interpreter i prevodioc
  `http://www.haskell.org/ghc`
  Prevodi u optimizirani C kôd, prikladan za "real-world" primjene

Interakcija s interpreterom (1)

```
$ ghci
GHCi, version 6.8.2: http://www.haskell.org/ghc/
:? for help
Loading package base ... linking ... done.
Prelude>
```

Biblioteka Prelude definira osnovne funkcije.

Interaktivni interpreter možemo koristiti kao "napredni kalkulator":

```
Prelude> 2+(3-1)
4
Prelude> 9 `div` 2
4
Prelude> it^5
1024
```

Interakcija s interpreterom (2)

Aplikacija funkcije piše se bez zagrada (kao u lambda-računu):

```
Prelude> sqrt 2
1.414213562373095
```

Zagrade su potrebne isključivo za grupiranje argumenta:

```
Prelude> sqrt (abs (-2))
1.414213562373095
Prelude> sqrt abs (-2)

<interactive>:1:0:
    No instance for (Floating (a -> a))
      arising from a use of ‘sqrt’ at <interactive>:1:0-12
    Possible fix: add an instance declaration for
      (Floating (a -> a))
    In the expression: sqrt abs (- 2)
    In the definition of ‘it’: it = sqrt abs (- 2)
```

Interakcija s interpreterom (3)

Ako funkcija uzima više argumenata, koristi se Curryjev postupak:

```
Prelude> (max 3) 10
10
```

Aplikacija funkcije je lijevo asocijativna, pa pišemo jednostavnije:

```
Prelude> max 3 10
10
Prelude> max (sqrt 625) 10
25.0
```

Vrijednosti možemo pohranjivati u varijable:

```
Prelude> let x = sqrt 625
Prelude> max x 10
25.0
```

Interakcija s interpreterom (4)

Operacije nad listama:

```
Prelude> head [1,2,3,4,5]
1
Prelude> length [1,2,3,4,5]
5
Prelude> take 2 [1,2,3,4,5]
[1,2]
Prelude> sum [1,2,3,4,5]
15
```

U interaktivnom načinu možemo definirati (kraće) funkcije:

```
Prelude> let uvecaj x = x + 1
Prelude> uvecaj 5
6
Prelude> let dvaput f x = f (f x)
Prelude> dvaput uvecaj 5
7
```

Varijable

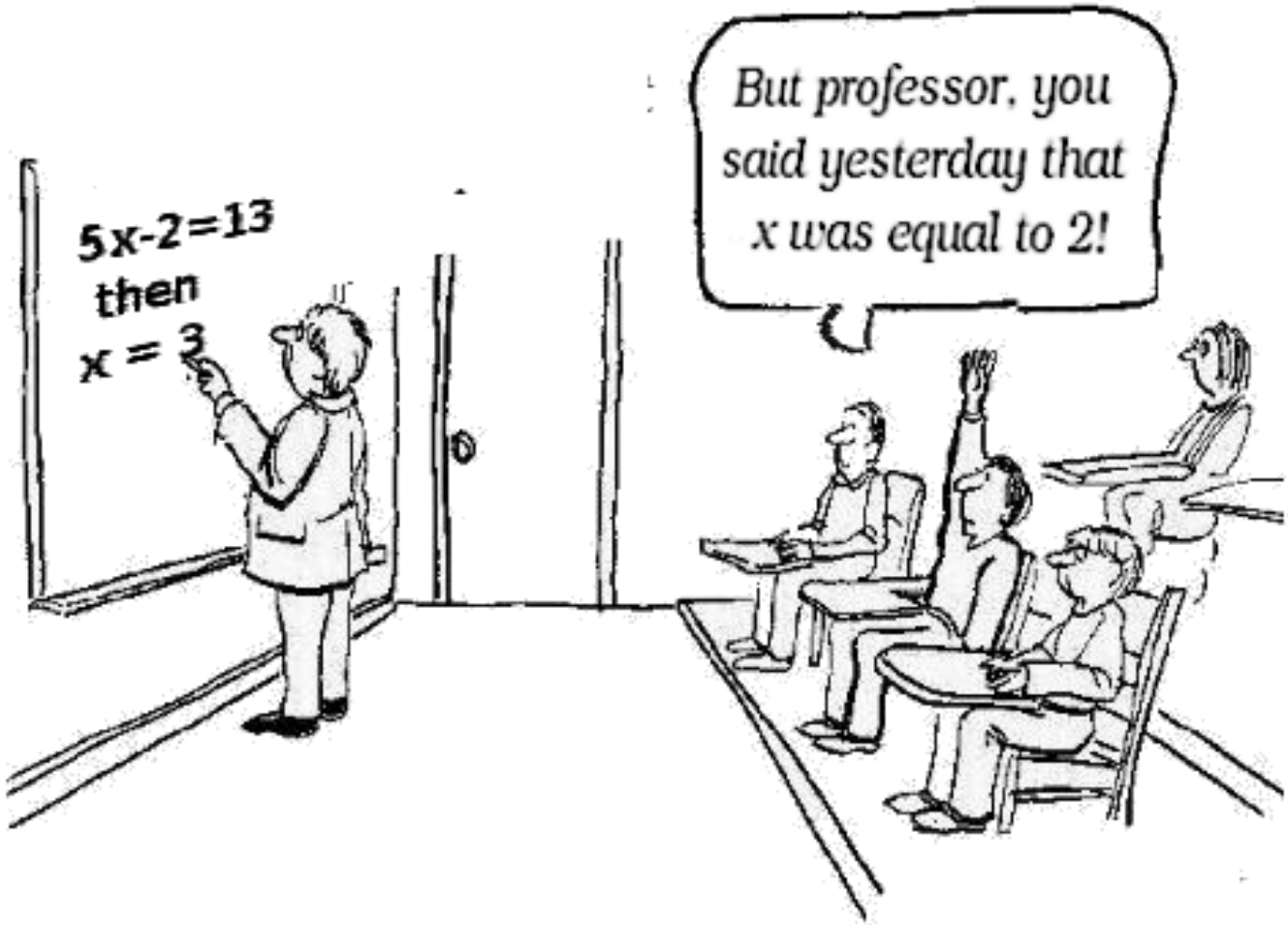
Varijable u Haskellu služe imenovanju izraza, a ne za pohranu vrijednosti u određenu memorijsku lokaciju!

Jednom kada se varijabla veže na neki izraz, njena vrijednost više se ne može promijeniti jer bi to uzrokovalo popratni efekt.

Pokušaj da se varijabli u programu promijeni vrijednost rezultira pogreškom *Multiple declarations*.

U interaktivnom načinu:

```
Prelude> let x = sqrt 625; y = max x 10
Prelude> y
25.0
Prelude> let x = 0    -- stvara novu varijablu!
Prelude> y
25.0
```


5x-2=13
then
x = 3
But professor, you said yesterday that x was equal to 2!

Tipovi podataka

Haskell je strogo i statički tipiziran (engl. *strongly and statically typed*): program mora ispavno kombinirati tipove, a tipovi moraju biti poznati pri prevođenju programa.

Ovakav sustav ima velike prednosti, premda u nekim situacijama otežava pisanje kôda.

Tipski sustav oslanja se na mehanizam zaključivanja o tipovima (engl. *type inference*) i temelji se na Hindley-Milnerovu algoritmu.

Hindley-Milnerov algoritam nalazi najopćenitiji mogući tip izraza.

Osnovni tipovi

Osnovni tipovi

`Bool`	Logičke vrijednosti	`True`, `False`
`Char`	Pojedinačni znakovi	`'a'`, `'\n'`
`String`	Nizovi znakova	`"abc"`
`Int`	Cijeli brojevi fiksnog raspona	123
`Integer`	Cijeli brojevi proizvoljnog raspona	95367431640625
`Float`	Brojevi s pomičnim zarezom	1.4142135

GHC može automatski izvesti ove tipove (naredba ‘`:t`’)

```
Prelude> let x = 5
Prelude> :t x
x :: Integer
Prelude> :t uvecaj x
uvecaj x :: Integer
```

Liste i n-torke

`[Integer]`	Lista cijelih brojeva	`[1,2,3]`
`[Char]`	Lista znakova (string)	`['a','b']` ili `"ab"`
`(Integer,Integer)`	Par cijelih brojeva	`(1,2)`
`[(Integer,String)]`	Lista parova (broj,string)	`[(1,"aa")]`
⋮		

```
Prelude> let [x,y,z] = [(1,2),(3,2),(0,0)]
Prelude> :t x
x :: (Integer, Integer)
Prelude> :t [x,z]
[x,z] :: [(Integer, Integer)]
Prelude> let lista = [(x,"objekt1"),(y,"objekt2")]
Prelude> :t lista
lista :: [((Integer, Integer), [Char])]
```

Liste

Lista je temeljni tip podataka većine deklarativnih jezika.

Lista je definirana rekurzivno:
(1) prazna lista `[]` je lista
(2) lista se sastoji od glave (prvog elementa) i repa (ostalih elemenata), pri čemu je rep opet lista

Infiksni operator ‘`:`’ razdvaja glavu od repa:

```
1:(2:(3:[]))      ⇒  [1,2,3]
1:2:3:[]          ⇒  [1,2,3]
[1]:[2]:[3,4]:[]  ⇒  [[1],[2],[3,4]]
```

Notacija s uglatim zagradama u stvari je samo sintaksni šećer.

Preferira se notacija s uglatim zagradama, osim kada je potrebno izravno pristupiti glavi ili repu.

Dodatne notacije za liste

Skraćeni način definiranja liste:

```
[1..10]    ⇒  [1,2,3,4,5,6,7,8,9,10]
[1,3..10]  ⇒  [1,3,5,7,9]
[1,3..]    ⇒  [1,3,5,7,9,...        beskonačna lista!
```

Sažet zapis liste (engl. list comprehension):

```
Prelude> [ x^2 | x <- [1..5]]
[1,4,9,16,25]

Prelude> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

Prelude> [(x,y) | x <- [1..10], y <- [1..10], x+y == 10]
[(1,9),(2,8),(3,7),(4,6),(5,5),(6,4),(7,3),(8,2),(9,1)]
```

Tipovi funkcija

Funkcija argumente jednog tipa preslikava u argumente nekog drugog tipa:

`Bool -> Bool`	Uzima i vraća logičku vrijednost
`Int -> Int`	Uzima i vraća cijeli broj
`[Char] -> Int`	Uzima niz znakova i vraća cijeli broj
`(Int,Int) -> Int`	Uzima par cijelih brojeva i vraća cijeli broj
⋮	

```
zbroji :: (Int, Int) -> Int
zbroji (x,y) = x + y

Prelude> :t zbroji
zbroji :: (Int, Int) -> Int
Prelude> :t zbroji (1,2)
zbroji (1,2) :: Int
```

Curryjev postupak

Za funkcije od više argumenata uglavnom se ne koriste n-torke, nego Curryjev postupak:

```
zbroji :: (Int, Int) -> Int
zbroji' :: Int -> (Int -> Int)     -- Curryjev oblik
```

(`zbroji'` uzima prvi broj i vraća funkciju koja uzima drugi broj)

Operator '->' je desno asocijativan, pa možemo pisati:

```
zbroji' :: Int -> Int -> Int
```

Isto vrijedi i za funkcije s više od dva argumenta. Npr.:

```
pomnozi :: Int -> Int -> Int -> Int
pomnozi x y z = x*y*z
```

Djelomična aplikacija

Curryjeve funkcije argumente uzimaju jedan po jedan, što ih čini vrlo fleksibilnima.

Konkretno, te se funkcije mogu djelomično aplicirati tako da ih se aplicira samo na dio njihovih argumenata:

```
Prelude> :t zbroji'
zbroji' :: Int -> Int -> Int
Prelude> :t zbroji' 1
zbroji' 1 :: Int -> Int
Prelude> let dodajJedan = zbroji' 1
Prelude> dodajJedan 2
3
Prelude> dodajJedan it
4
```

Djelomičnom aplikacijom fiksiraju se (lijevi) argumenti funkcije.

Polimorfizam

Mnoge funkcije dovoljno su općenite da ih se može primijeniti na različite tipove (npr. funkcija `length`).

Takve funkcije uobičajeno su polimorfnog tipa:

`id :: a -> a`	Funkcija identiteta
`length :: [a] -> Int`	Duljina liste
`reverse :: [a] -> [a]`	Obrtanje liste
`head :: [a] -> a`	Uzima glavu liste
`concat :: [[a]] -> [a]`	Konkatenira podliste
`take :: Int -> [a] -> [a]`	Uzima prvih `n` članova liste
`fst :: (a,b) -> a`	Vraća prvi element dvojke
`snd :: (a,b) -> b`	Vraća drugi element dvojke
⋮	

Varijable `a` i `b` su tzv. tipske varijable (engl. *type variables*).

Preopterećenje (1)

Polimorfizam pokriva slučajeve kada različite vrijednosti imaju sličnu podatkovnu strukturu, dok preopterećenje (engl. *overloading*) pokriva slučajeve kada su nad različitim strukturama definirane slične operacije.

Ostvaruje se tipskim razredima (engl. *type classes*): definira koje funkcije neki tip mora implementirati, a da bi postao instancom tog razreda.

Osnovni tipski razredi

`Eq`	tipovi s jednakošću	`==`, `/=`
`Ord`	tipovi s uređajem (nasljeđuje `Eq`)	`<`, `>`, `<=`, `>=`, ...
`Num`	numerički tipovi (nasljeđuje `Eq`)	`+`, `-`, `*`, ...
`Integral`	cjelobrojni tipovi (nasljeđuje `Num`)	`div`, `mod`
`Fractional`	razlomački tipovi (nasljeđuje `Num`)	`/`, `recip`

Preopterećenje (2)

Tipskim razredima ograničava se polimorfizam funkcija. Npr.:

```
(+) :: (Num a) => a -> a -> a
sum :: (Num a) => [a] -> a
elem :: (Eq a) => a -> [a] -> Bool
max :: (Ord a) => a -> a -> a
```

Npr.:

```
Prelude> max "abc" "bcd"
"bcd"
Prelude> sum "bcd"

<interactive>:1:0:
  No instance for (Num Char)
    arising from a use of ‘sum’ at <interactive>:1:0-8
  Possible fix: add an instance declaration for (Num Char)
  In the expression: sum "bcd"
  In the definition of ‘it’: it = sum "bcd"
```

Definiranje funkcija

Uvjetno izvođenje može se ostvariti konstruktom if-then-else:

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

Kad god je moguće, preferira se uporaba čuvara (engl. *guards*):

```
abs :: Int -> Int
abs n | n >= 0    = n
      | otherwise = -n
```

Lokalne definicije uvode se (uvučeno!) u bloku `where`:

```
prepolovi :: [a] -> ([a],[a])
prepolovi xs = (take pola xs,drop pola xs)
  where pola = length xs `div` 2
```

Podudaranje uzoraka

Podudaranje uzoraka (engl. *pattern matching*):

- Funkcija se raspisuje po jednadžbama, od specifičnijih k općenitijim
- Izračunava se prva jednadžba čiji se uzorak podudara s pozivnom vrijednošću funkcije

```
fst :: (a,b) -> a   -- vraca prvi element para
fst (x,_) = x

head :: [a] -> a    -- vraca glavu liste
head (x:_) = x

foo [x]   = x       -- koji je tip ove funkcije?
foo [x,y] = x-y
foo _     = 0
```

Lambda-izrazi

Lambda-izrazima definiraju se jednostavne funkcije koje nam trebaju samo na jednom mjestu (inače ne mogu biti anonimne).

$\lambda x\,.\,x+1 \quad \Rightarrow \quad$ `\x -> x + 1`

$\lambda xy\,.\,x+y \quad \Rightarrow \quad$ `\x y -> x + y`

```
Prelude> (\x -> x+1) 1
2
Prelude> let uvecaj = (\x -> x+1)    -- vrlo netipicno!
Prelude> uvecaj 1
2
Prelude> (\(x:_) -> x) [1,2,3]       -- lambda s uzorkom
1
```

Rekurzivne funkcije

Rekurzivne funkcije obično imaju jednu ili više nerekurzivnih (osnovnih) jednadžbi te jednu ili više rekurzivnih jednadžbi:

```
fact :: (Num a) => a -> a
fact 0 = 1
fact n = n * fact (n-1)
```

Funkcija koja ima više argumenata također može biti rekurzivna:

```
prod :: (Num a) => a -> a -> a
prod _ 0 = 0
prod m n = m + prod m (n-1)
```

Rekurzija nad listom (1)

Rekurzija se često provodi nad podatkovnim strukturama (strukturna rekurzija), i to najčešće nad listom:

```
sum :: (Num a) => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs

length :: [a] -> Int    -- zasto ovdje nema ogranicenja?
length []     = 0
length (_:xs) = 1 + length xs

elem :: (Eq a) => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) | x==y       = True
              | otherwise = elem x ys
```

Rekurzija nad listom (2)

```
udvostruci :: (Num a) => [a] -> [a]
udvostruci []     = []
udvostruci (x:xs) = 2*x : udvostruci xs

uvecaj :: (Num a) => [a] -> [a]
uvecaj []     = []
uvecaj (x:xs) = x+1 : uvecaj xs

Prelude> udvostruci [1,2,3]
[2,4,6]
Prelude> uvecaj [1,2,3]
[2,3,4]
Prelude> 6 `elem` (uvecaj (udvostruci [1,2,3]))
False
```

Rekurzija nad listom (3)

Funkcija `zip` stapa dvije liste u jednu listu parova:

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _  = []
zip _  [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

Prelude> zip ['a','b','c'] [1,2,3,4,5]
[('a',1),('b',2),('c',3)]
```

Tipičan primjer višestruke rekurzije:

```
qSort :: (Ord a) => [a] -> [a]
qSort []     = []
qSort (x:xs) = qSort ys ++ [x] ++ qSort zs
  where ys = [y | y <- xs, y <= x]
        zs = [z | z <- xs, z > x]
```

Lijena evaluacija

Lijenom evaluacijom izbjegava se nepotrebno izračunavanje. Npr.:

```
Prelude> let inf = 1 + inf
inf
*** Exception: stack overflow
Prelude> fst (1,inf)
1
```

Lijena evaluacija omogućava rad s beskonačnim strukturama podataka. Npr.:

```
Prelude> let ones = 1 : ones
Prelude> take 5 ones
[1,1,1,1,1]
Prelude> zip ['a','b','c'] [1..]
[('a',1),('b',2),('c',3)]
```

Funkcijski obrasci

Osnovna prednost funkcija višeg reda jest što mogu apstrahirati tipične funkcijske obrasce.

Tipični funkcijski obrasci jesu `map`, `filter` i `fold`.

Razmotrite sljedeće dvije definicije:

```
udvostruci :: (Num a) => [a] -> [a]
udvostruci []     = []
udvostruci (x:xs) = 2*x : udvostruci xs

uvecaj :: (Num a) => [a] -> [a]
uvecaj []     = []
uvecaj (x:xs) = x+1 : uvecaj xs
```

Primjećujete li neki obrazac?

Funkcija map

Funkcija map aplicira danu funkciju na svaki element liste:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Često se funkcija koju primjenjujemo definira lambda-izrazom:

```
uvecaj :: (Num a) => [a] -> [a]
uvecaj xs = map (\x -> x+1) xs

udvostruci :: (Num a) => [a] -> [a]
udvostruci xs = map (\x -> 2*x) xs
```

Uporabom funkcijskih obrazaca kôd postaje sažetiji, pregledniji i svima razumljiv (idiomatski).

Funkcija `filter`

Funkcija `filter` probire elemente liste koji zadovoljavaju dani predikat:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

Npr.:

```
Prelude> filter even [1..20]
[2,4,6,8,10,12,14,16,18,20]
```

```
Prelude> filter (\x -> x>5 && odd x) [1..20]
[7,9,11,13,15,17,19]
```

Kompozicija funkcija

Funkcija '.' (definirana kao infiksni operator) daje novu funkciju koja je kompozicija dviju funkcija:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Npr.:

```
Prelude> (map (\x -> x+1) . filter (\x -> x>5)) [1..20]
[7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]
```

Još sažetije:

```
Prelude> map (+1) . filter (>5) $ [1..20]
[7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]
```

Monade

Monade su vrlo važan koncept svojstven Haskellu.

Monade

Monade apstrahiraju različite načine izračuna: izračun sa stanjem, nedeterministički izračun, izračun koji može ne uspjeti i sl. Jedan primjer monade je IO-monada, pomoću koje se izvode sve ulazno-izlazne operacije.

Monade omogućavaju izvođenje ulazno-izlaznih operacija bez narušavanja referencijalne prozirnosti (kod je i dalje čisto funkcijski!).

Primjer monadičkog koda:

```
main = do
  putStrLn "Upisi␣broj"
  x <- getLine
  putStrLn $ "Upisao␣si␣broj" ++ x
```

Primjeri (1)

Pitagorine trojke:

```
pythagoreanTriplets =
  [(a,b,c) | c <- [1..10], a <- [1..c], b <- [1..a],
             a^2 + b^2 == c^2]
```

Prebrojavanje elemenata u listi:

```
counts :: Ord a => [a] -> [(a,Int)]
counts xs = [ (y,length ys) | ys@(y:_) <- group $ sort xs]
```

Zajednički prefiks dviju lista:

```
commonPrefix :: Eq a => [a] -> [a] -> [a]
commonPrefix xs =
  map fst . takeWhile (uncurry (==)) . zip xs
```

Camel-casing znakovnog niza:

```
camelCase s = concat [toUpper g : r | (g:r) <- words s]
```

Primjeri (2)

"Hello world" program:

```
main = putStrLn "hello, world!"
```

Filtriranje praznih linija:

```
main :: IO ()
main = interact (unlines . filter (not . null) . lines)
```

Ispis sadržaja datoteke s prefiksiranim brojem linije:

```
cat :: String -> IO ()
cat f = withFile f ReadMode $ \h -> do
 s <- hGetContents h
 putStr . unlines .
 zipWith (\i l -> show i ++ ": " ++ l) [0..] $ lines s
```

Primjeri (3)

Definicija strukture stabla:

```
data Tree a = Null | Node a (Tree a) (Tree a)
  deriving (Show,Eq)
```

Zbrajanje brojeva u stablu:

```
sumTree :: Tree Int -> Int
sumTree Null          = 0
sumTree (Node x l r) = x + sumTree l + sumTree r
```

Umetanje u sortirano stablo:

```
treeInsert :: Ord a => a -> Tree a -> Tree a
treeInsert x Null = Node x Null Null
treeInsert x t@(Node y l r)
  | x < y     = Node y (treeInsert x l) r
  | x > y     = Node y l (treeInsert x r)
  | otherwise = t
```

Dalje...?

Haskell je predivan, ali složen. Ovime smo tek zagrebli površinu!

Imate pitanja? Pridružite nam se na grupi:

`http://groups.google.hr/group/haskell-fer`

Želite naučiti Haskell i svom mozgu priuštiti nezaboravan doživljaj? Upišite Programiranje u Haskellu:

`http://www.fer.unizg.hr/predmet/puh`

On-line materijali

Haskell homepage

- http://www.haskell.org

Tutorials

- Haskell Wikibook
  http://en.wikibooks.org/wiki/Haskell
- Learn You A Haskell for Great Good!
  http://learnyouahaskell.com/
- Yet Another Haskell Tutorial
  http://darcs.haskell.org/yaht/yaht.pdf

Books

- Real World Haskell
  http://book.realworldhaskell.org/