Programske paradigme

— Funkcionalna paradigma —

Milena Vujošević Janičić

www.matf.bg.ac.rs/~milena

Matematički fakultet, Univerzitet u Beogradu

Pregled

Pregled

Programske paradigme

- Imperativnu paradigmu karakteriše postojanje naredbi - izvršavanje programa se svodi na izvršavanje naredbi
- Izvršavanje programa može se svesti i na evaluaciju izraza.
- U zavisnosti od izraza, imamo logičku paradigmu (izrazi su relacije) i funkcionalnu (izrazi su funkcije)
- Ako je izraz relacija — rezultat true-false.
- Ako je izraz funkcija, rezultat mogu da budu različite vrednosti.
- Funkcije smo sretali i ranije, u drugim programskim jezicima, ali iako nose isto ime, ovde se radi o suštinski različitim pojmovima.

Programske paradigme

- Pomenute paradigme se temelje na različitim teorijskim modelima
  - Formalizam za imperativne jezike —
  - Formalizam za logičke jezike —
  - Formalizam za funkcionalne jezike —

Programske paradigme

- Pomenute paradigme se temelje na različitim teorijskim modelima
  - Formalizam za imperativne jezike — Tjuringova i URM mašina
  - Formalizam za logičke jezike —
  - Formalizam za funkcionalne jezike —

Programske paradigme

- Pomenute paradigme se temelje na različitim teorijskim modelima
  - Formalizam za imperativne jezike — Tjuringova i URM mašina
  - Formalizam za logičke jezike — Logika prvog reda
  - Formalizam za funkcionalne jezike —

Programske paradigme

- Pomenute paradigme se temelje na različitim teorijskim modelima
  - Formalizam za imperativne jezike — Tjuringova i URM mašina
  - Formalizam za logičke jezike — Logika prvog reda
  - Formalizam za funkcionalne jezike — Lambda račun

Programske paradigme

- Pomenute paradigme se temelje na različitim teorijskim modelima
  - Formalizam za imperativne jezike — Tjuringova i URM mašina
  - Formalizam za logičke jezike — Logika prvog reda
  - Formalizam za funkcionalne jezike — Lambda račun
- Funkcionalni jezici su mnogo bliži svom teorijskom modelu nego imperativni jezici pa je poznavanje lambda računa važno i ono omogućava bolje funkcionalno programiranje

Šta možemo sa funkcionalnim programiranjem?

- Ekspresivnost funkcionalnih jezika je ekvivalentna ekspresivnosti lambda računa.
- Ekspresivnost lambda računa je ekvivalentna ekspresivnosti Tjuringovih mašina (1937)
- Ekspresivnost Tjuringovih mašina je ekvivalentna ekspresivnosti imperativnih jezika
- **Svi programi koji se mogu napisati imperativnim stilom, mogu se napisati i funkcionalnim stilom.**
- Lambda račun naglašava pravila za transformaciju izraza i ne zamara se arhitekturom mašine koja to može da ostvari

Funkcionalna paradigma (ili funkcijska paradigma)

- Funkcionalna paradigma (ili funkcijska paradigma) se zasniva na pojmu matematičkih funkcija — otud potiče i njen naziv.
- Osnovni cilj funkcionalne paradigme je da oponaša matematičke funkcije - rezultat toga je pristup programiranju koji je u osnovi drugačiji od imperativnog programiranja.
- Osnovna apstrakcija u imperativnim programskim jezicima je apstrakcija kontrole toka (podrutina), u OO jezicima je objekat, a u funkcionalnim jezicima to je funkcija.

Osnovne aktivnosti

- Funkcionalno programiranje je stil koji se zasniva na izračunavanju izraza kombinovanjem funkcija. Osnovne aktivnosti su:
  1. Definisanje funkcije (pridruživanje imenu funkcije vrednosti izraza pri čemu izraz može sadržati pozive drugih funkcija)
  2. Primena funkcije (poziv funkcije sa zadatim argumentima).
  3. Kompozicija funkcija (navođenje niza poziva funkcija) - kreiranje programa.
- Program u funkcionalnom programiranju je niz definicija i poziva funkcija.
- Izvršavanje programa je evaluacija funkcija.

Primer

- Pretpostavimo da imamo funkciju:

$$max(x, y) = \begin{cases} x & , x > y \\ y & , y \geq x \end{cases}$$

- Prethodnu funkciju možemo koristiti za definisanje novih funkcija. Na primer:
  max3(x,y,z) = max(max(x,y), z);
  Funkcija max3 je kompozicija funkcija max.
- Prethodno definisane funkcije možemo kombinovati na razne načine. Na primer, ako treba izračunati max6(a,b,c,d,e,f), to možemo uraditi na sledeće načine:
  max(max3(a,b,c), max3(d,e,f))
  max3(max(a,b), max(c,d), max(e,f))
  max(max(max(a,b, max(c,d)), max(e,f))

Osnovne funkcije i mehanizmi

- Da bi se uspešno programiralo, treba:
  - ugraditi neke osnovne funkcije u sam programski jezik,
  - obezbediti mehanizme za formiranje novih (kompleksnijih) funkcija
  - obezbediti strukture za prezentovanje podataka (strukture koje se koriste da se predstave parametri i vrednosti koje funkcija izračunava)
  - formirati biblioteku funkcija koje mogu biti korišćene kasnije.
- Ukoliko je jezik dobro definisan, broj osnovnih funkcija i struktura podataka je relativno mali.

Apstrakcija — funkcija

- Osnovna apstrakcija je funkcija i sve se svodi na izračunavanje funkcija.
- Funkcije je ravnopravna sa ostalim tipovima podataka, može biti povratna vrednost ili parametar druge funkcije. Na primer

  ```
  duplo f x = f (f x)
  ```

  Ovo je funkcija višeg reda i to je važna karakteristika funkcionalnih jezika.
- Posebno važne funkcije višeg reda su map, filter i fold (reduce)

Funkcionalna paradigma

- Funkcionalna paradigma nastaje 59. godine prošlog veka — najistaknutiji predstavnik funkcionalne paradigme bio je programski jezik Lisp (LISt Processing—LISP).
- Stagnacija u razvoju funkcionalne paradigme sedamdesetih godina prošlog veka.

Funkcionalni programski jezici

- 1924 Kombinatorna logika — Schonfinkel & Curry
- 1930 Lambda račun — Alonzo Church
- 1959 Lisp — McCarthy, MIT
- 1964 SECD — apstraktna mašina i jezik ISWIM (Landin) — prekretnica za razvoj kompajlera i praktičnih rešenja
- 1977 FP — Backus

Funkcionalna paradigma

- Tjuringova nagrada za 1977 godinu je dodeljena John Backus-u za njegov rad na razvoju Fortran-a.
- Svaki dobitnik ove nagrade drži predavanje prilikom formalne dodele, koje se posle štampa u časopisu Communications of the ACM.
- Backus je održao predavanje sa poentom da su čisti funkcionalni programski jezici bolji od imperativnih jer su programi koji se pišu na njima čitljiviji, pouzdaniji i verovatnije ispravni.

Funkcionalna paradigma

- Suština njegove argumentacije bila je da su funkcionalni jezici lakši za razumevanje, i za vreme i nakon razvoja, najviše zbog toga što je **vrednost izraza nezavisna od konteksta u kojem se izraz nalazi**.
- Osnovna karakteristika čistih funkcionalnih programskih jezika je **transparentnost referenci** što kao posledicu ima nepostojanje propratnih (bočnih) efekta.

Funkcionalna paradigma

- U okviru svog predavanja, Backus je koristio svoj funkcionalni jezik FP kao podršku argumentaciji koju izlaže.
- Iako jezik FP nije zaživeo, njegovo izlaganje je motivisalo debate i nova istraživanja funkcionalnih programskih jezika.

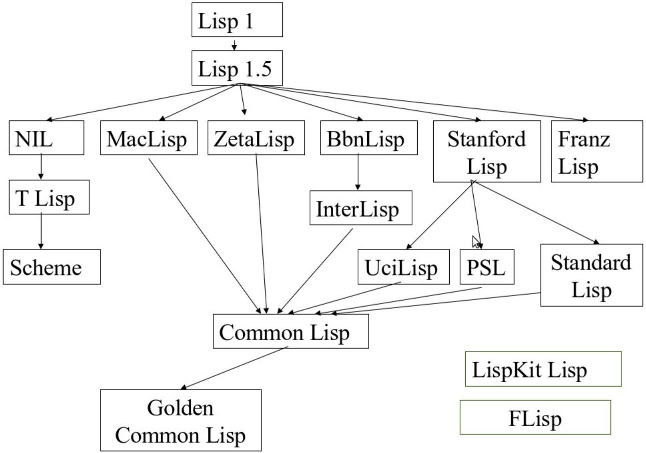
Funkcionalni programski jezici

- 1978 Scheme – Sussman and Steele
- 1978 ML – Robin Milner, University of Edinburgh
- 1985 Miranda – David Turner (ML)
- 1986 Erlang (Ericsson) — 1998 open source
- 1990 SML — Robin Milner, University of Edinburgh
- 1990 Haskell – Haskell Committee (ML, Miranda), Haskell 98, Haskell 2010

Funkcionalni programski jezici

- 1996 OCaml — Xavier Leroy, Jérôme Vouillon, Damien Doligez, Didier Rémy (ML)
- 2002 F# — Don Syme, Microsoft Research, Cambridge (ML)
- 2003 Scala – Martin Odersky, EPFL
- 2012 Elixir - José Valim
- Podrška funkcionalnim konceptima: 2011 C++, JAVA, skript jezici (Phyton)

Lisp (ANSI Common Lisp standard 1994)



Funkcionalna paradigma

- Razvojem i zrelošću jezika kao što su ML, Haskell, OCaml, F# i Scala raste interesovanje i upotreba funkcionalnih programskih jezika.
- Funkcionalni jezici su obično jezici opšte namene i mogu se koristiti u proizvoljnim domenima.
- Na primer, u domenima obrade baza podataka, finansijskog modelovanja, statičke analize programa, bioinformatike... Broj domena upotrebe raste i zapravo ne postoje specifični domeni za funkcionalno programiranje.
- Funkcionalni jezici su pogodni za paralelizaciju, što ih posebno čini popularnim za paralelno i distribuirano programiranje.
- Sve više programskih jezika koristi osnovne funkcionalne koncepte.

Pregled

1. Uvod u funkcionalno programiranje

2. Programski jezik Haskell
   - Razvoj
   - Karakteristike
   - Demonstracija GHC

3. Svojstva funkcionalnih jezika

4. Teorijske osnove — lambda račun

Haskell Brooks Curry

Haskell Brooks Curry 1900–1982 logičar i matematičar

Haskell

- 1987 — međunarodni odbor počinje sa dizajnom novog, zajedničkog funkcionalnog jezika
- 1990 — odbor najavljuje specifikaciju Haskell 1.0
- 1990–1997 — četiri izmene standarda
- 1998 — Haskell 98
- 2010 — Haskell Prime, tj Haskell 2010

Karakteristike Haskella

- `https://www.haskell.org/`
- Čist funkcionalni jezik
- Lenja evaluacija — izbegavaju se nepotrebna izračunavanja
- Haskel ima moćni sistem tipova — tipovi se ne moraju uvek navoditi, tj postoji automatsko zaključivanje tipova
- Statički tipiziran jezik — tipovi se određuju u fazi kompilacije
- Strogo tipiziran jezik — svi se tipovi moraju poklapati, nema implicitnih konverzija
- Podrška za paralelno i distirbuirano programiranje

Karakteristike Haskella

- Podržava parametarski polimorfizam (višeobličje) i preopterećivanje — što omogućava sažeto i generičko programiranje
- Podržava kompaktan i ekspresivan način definisanja listi kao osnovnih struktura funkcionalnog programiranja

Karakteristike Haskella

- Funkcije višeg reda omogućavaju visok nivo apstrakcije i korišćenja funkcijskih oblikovnih obrazaca (uočavanje obrazaca izračunavanja koje se često sprovode i njihovo izdvajanje u funkcije višeg reda)
- Haskell ima podršku za monadičko programiranje koje omogućava da se propratni efekti izvedu bez narušavanja transparentnosti referenci
- Razrađena biblioteka standardnih funkcija (Standard Library) i dodatnih modula (Hackage)

Standardizacija

- Standaradizaciju sprovodi međunarodni odbor (Haskell Commitee)
- Literatura:
  `https://www.haskell.org/documentation`
  Real world Haskell
  `http://book.realworldhaskell.org/`
- Haskell se koristi u: `https://wiki.haskell.org/Haskell_in_industry`

Interpreteri i prevodioci

- GHC — Glasgow Haskell Compiler — interaktivni interpreter i kompajler `https://www.haskell.org/ghc/`
- Kompajler koji proizvodi optimizovan kôd koji se može upotrebljvati za industrijske primene
- Haskell se kotira vrlo visoko po brzini izvršavanja (na nekim primerima čak i u nivou sa C-om i C++)
- Ekstenzija za Haskell kôd .hs, npr

  ghc hello.hs

- Interpreter: ghci
- Razvojno okruženje za Haskell: `https://wiki.haskell.org/IDEs`

GHCi

```
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

- Biblioteka Prelude definiše osnovne funkcije

GHCi — help komanda

:help, :?, :h — prikazivanje komandi interpretera
:quit, :q — izlazak iz interpretera

```
Prelude> :help
 Commands available from the prompt:

   <statement>                 evaluate/run <statement>
   :                           repeat last command
   :{\n ..lines.. \n:}\n       multiline command
   :add [*]<module> ...        add module(s) to the current target set
   :browse[!] [[*]<mod>]       display the names defined by module <mod>
                               (!: more details; *: all top-level names)
   :cd <dir>                   change directory to <dir>
   :cmd <expr>                 run the commands returned by <expr>::IO String
   :ctags[!] [<file>]          create tags file for Vi (default: "tags")
                               (!: use regex instead of line number)
   :def <cmd> <expr>           define a command :<cmd>
   :edit <file>                edit file
```

Zdravo svima!

```
Prelude> putStrLn "Zdravo svima!"
Zdravo svima!
```

Kalkulator

Demonstracija naprednog kalkulatora

```
Prelude> 2+3-1
4
Prelude> 9/2
4.5
Prelude> div 9 2
4
Prelude> it^5
1024
```

GHC kompajler

- Jednostavni primeri mogu da se isprobaju u interpreteru
- Za pisanje programa u Haskelu koristi se kompajler
- Kôd pišemo u datoteci sa ekstenzijom .hs

GHC kompajler

- Ukoliko želimo da napravimo izvršnu verziju, potrebno je da definišemo main funkciju od koje će početi izračunavanje
- Prevođenje
  `ghc 1.hs`
  Pokretanje `./1`
- Možemo napisati i kôd koji sadrži samo definicije funkcija koje učitamo u interpreter kao modul i koristimo
- `Prelude> :load 1.hs`
  Izlazak sa
  `*Main> :module`

Pregled

Strukture podataka

- Funkcionalni jezici imaju osnovne tipove podataka (celobrojna vrednosti, realne vrednosti, logičke vrednosti, stringovi)
- Osnovna struktura podataka koja se javlja u svim funkcionalnim jezicima je lista
- Funkcionalni jezici podržavaju i torke koje mogu da sadrže elemente različitih tipova
- Često: torke zauzimaju kontinualni prostor u memoriji (slično kao nizovi) dok su liste impelementirane preko povezanih listi
- Izbor odgovarajuće strukture zavisi od problema (da li je broj elemenata fiksiran)

Osnovni tipovi u Haskelu

- Osnovni tipovi: Bool, Char, String, Int, Integer, Float

```
Prelude> let x = 3
Prelude> :type x
x :: Integer
Prelude> let x = 'a'
Prelude> :type x
x :: Char
Prelude> let x="pera"
Prelude> :type x
x :: [Char]
```

Torke u Haskelu

- Torke su kolekcije fiksiranog broja vrednosti ali potencijalno različitih tipova

```
n::(t1, ..., tn)
n = (e1, e2, ... en)
```

- Funkcije selektori za parove fst i snd

```
t::(Float, Integer)
t = (2.4, 2)
-- fst t -> 2.4
-- snd t-> 2
```

- Torke mogu biti parametri i povratne vrednosti funkcija

Liste u Haskelu

- Liste su kolekcije proizvoljnog broja vrednosti istog tipa

```
n :: [t]
n = [e1, e2, ... en]
```

- Liste mogu biti parametri i povratne vrednosti funkcija
- Veliki broj funkcija za rad sa listama

Primer: torka i lista u Haskelu

- Polinom drugog stepena

```
p2 :: (Float, Float, Float)
p2 = (1, 2, 3)
```

- Polinom proizvoljnog stepena

```
pn :: [Float]
pn = [1, 2, 3]
```

Liste u Haskelu

```
Prelude> [1,2,3]
[1,2,3]
Prelude> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [1,3..40]
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39]
Prelude> [1,6..90]
[1,6,11,16,21,26,31,36,41,46,51,56,61,66,71,76,81,86]
Prelude> [1,6..] --beskonačna lista!
```

Liste u Haskelu

```
Prelude> ['A'..'F']
"ABCDEF"
Prelude> ['A'..'z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz"
Prelude> [5..1]
[]
Prelude> [5,4..1]
[5,4,3,2,1]
```

Liste u Haskelu

Funkcije za rad sa listama

```
Prelude> head [1,2,3,4,5]
1
Prelude> head ['a', 'b', 'c']
'a'
Prelude> length [1..5]
5
Prelude> length ['a', 'b', 'c']
3
Prelude> take 2 [1,2,3,4,5,6]
[1,2]
Prelude> sum [1,6..90]
```

Liste u Haskelu

Funkcije za rad sa listama

```
Prelude> [0..10] !! 5
5
Prelude> [0,2..] !! 50
100
```

Lenjo izračunavanje — lista jeste beskonačna, ali nama treba 50-ti element i lista će samo dotle biti sračunata

First class citizen

First class citizen

U okviru programskog jezika za neki gradivni element se kaže da je **građanin prvog reda** ako u okviru jezika ne postoje restrikcije po pitanju njegovog kreiranja i korišćenja.

- Građani prvog reda mogu da se čuvaju u promenljivama, da se prosleđuju funkcijama, da se kreiraju u okviru funkcija i da se vrate kao povratna vrednost funkcija.
- U dinamički tipiziranim jezicima (tj gde se tipovi određuju u fazi izvršavanja programa), građani prvog reda imaju tip koji se proverava u fazi izvršavanja
- U funkcionalnom programiranju **funkcije su građani prvog reda**

First class citizen — primeri

- Navedite primere građana prvog reda u programskom jeziku C
- Da li su funkcije građani prvog reda u programskom jeziku C?
- Navedite primere pojmova koji nisu građani prvog reda u programskom jeziku C

Primena funkcije

- Primena funkcije je izračunavanje vrednosti funkcije za neke konkretne argumente
- Primena funkcije je određena uparivanjem imena funkcije sa određenim elementom iz domena.
- Rezultat se izračunava evaluiranjem izraza koji definiše preslikavanje funkcije.

Funkcije višeg reda

Funkcije višeg reda

Funkcije višeg reda (funkcijske forme) imaju jednu ili više funkcija kao parametre ili imaju funkciju kao rezultat, ili oba.

- Primer: kompozicija funkcija
  - Kompozicija funkcija ima dve funkcije kao parametre i rezultat koji je takođe funkcija.
  - Na primer, $f(x) = x + 5$ i $g(x) = 2 \cdot x$, kompozicija funkcija f i g je $h = f \circ g = f(g(x)) = (2 \cdot x) + 5$

Funkcije višeg reda

- Primer: α funkcija (`apply to all` ili `map`)
  - Ova funkcija ima jedan parametar koji je funkcija, i kada se primeni na listu parametara, kao rezultat se dobija lista vrednosti koja se izračunava primenom funkcije parametra na listu parametra.
  - Na primer, $f(x) = 2 \cdot x$ i $\alpha(f, (1, 2, 3)) = (2, 4, 6)$

Funkcija map u Haskelu

```
Prelude> map (+1) [1,5,3,1,6]
[2,6,4,2,7]
Prelude> map (++ "!") ["Zdravo", "Dobar dan", "Cao"]
["Zdravo!", "Dobar dan!", "Cao!"]
Prelude> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
Prelude> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
Prelude> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

Funkcije višeg reda

- Primer: ϕ funkcija (`filter`)
  - Ova funkcija ima jedan parametar koji je funkcija (povratna vrednost true/false), i kada se primeni na listu parametara, kao rezultat se dobija lista vrednosti za koje je ispunjen uslov funkcije.
  - Na primer, $f(x) = odd(x)$ i $\phi(f, (1, 2, 3)) = (1, 3)$

Funkcija filter u Haskelu

```
Prelude> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
Prelude> filter (==3) [1,2,3,4,5]
[3]
Prelude> filter even [1..10]
[2,4,6,8,10]
```

Funkcije višeg reda

- Primer: ρ funkcija (`reduce`)
  - Ova funkcija ima jedan parametar koji je funkcija, i kada se primeni na listu parametara, kao rezultat se dobija odgovarajuća vrednost.
  - Na primer, $f(x, y) = x + y$ i $\rho(f, (1, 2, 3)) = 6$

Funkcije foldl i foldr u Haskelu

```
Prelude> foldl (+) 0 [1,2,3]
6
Prelude> foldr (+) 0 [1,2,3]
6
Prelude> foldl (-) 0 [1,2,3]
-6
Prelude> foldr (-) 0 [1,2,3]
2
-- 1-(2-(3-0))
```

Matematičke osnove

- Matematička funkcija je preslikavanje elemenata jednog skupa (domena) u elemente drugog skupa (kodomena).
- Definicija funkcije uključuje zadavanje domena, kodomena i preslikavanja.
- Preslikavanje može da bude zadato izrazom ili tabelom.
- Funkcije se primenjuju na pojedinačne elemente iz domena, koji se zadaju kao parametri (argumenti) funkcija.
- Domen može da bude Dekartov proizvod različitih skupova, tj funkcija može da ima više od jednog parametra (ili se funkcija sa više argumenata može svesti na primene funkcija sa po jednim argumentom)

Definicija matematičke funkcije

- Definicija funkcije obično uključuje ime funkcije za kojom sledi lista parametara u zagradama, a zatim i izraz koji zadaje preslikavanje. Na primer:
  $kub(x) \equiv x \cdot x \cdot x$ za svaki realni broj x.
- U okviru ove definicije, domen i kodomen su skupovi realnih brojeva, dok se znak $\equiv$ koristi sa značenjem "se definiše kao".
- Parametar x može da bude bilo koji član domena, ali se fiksira kada god se evaluira u okviru izraza funkcije.
- Na ovaj način se promenljive u matematičkoj funkciji razlikuju od promenljivih u imperativnim jezicima. Na primer, za kub(x) imamo $kub(2.0) = 2.0 \cdot 2.0 \cdot 2.0 = 8$ Parametar x je vezan sa vrednošću 2.0 tokom evaluacije, x je konstanta i njena vrednost se ne može promeniti za vreme evaluacije.

Primeri definisanja matematičkih funkcija

Funkcija abs

- Domen: svi realni brojevi,
- Kodomen: pozitivni realni brojevi
- Preslikavanje se zadaje izrazom:

$$abs(x) \equiv \begin{cases} x & , x \geq 0 \\ -x & , x < 0 \end{cases}$$

Primeri definisanja matematičkih funkcija

Funkcija faktorijel

- Domen: prirodni brojevi
- Kodomen: prirodni brojevi
- Preslikavanje se zadaje izrazom:

$$n! \equiv \begin{cases} 1 & , n = 0 \\ n * (n-1)! & , n > 0 \end{cases}$$

Zadavanje preslikavanja

Jedna od osnovnih karakteristika matematičkih funkcija je da se izračunavanje preslikavanja kontroliše **rekurzijom i kondicionalnim izrazima**, a ne sa sekvencom i iteracijom (kao što je to kod imperativnih jezika).

Karakteristike matematičkih funkcija

- Matematičke funkcije: **vrednosti preslikavanja elemenata iz domena su uvek isti elementi iz kodomena** (jer ne postoje propratni (bočni) efekti i funkcije ne zavise od drugih spoljašnjih promenljivih)
- Kod imperativnih jezika vrednost funkcije može da zavisi od tekućih vrednosti različitih globalnih ili nelokalnih promenljivih.

```
int x = 0;
int f() { return ++x; }
...
cout << f() << " " << f() << endl;
```

Za razumevanje prethodnog koda potrebno je poznavanje tekućeg **stanja** programa.

Stanje programa

Stanje - osnovna karakteristika imperativnih jezika

Stanje programa čine sve vrednosti u memoriji kojima program u toku izvršavanja ima pristup.

- Imperativni jezici imaju implicitno stanje i izvršavanje programa se svodi na postepeno menjanje tog stanja izvođenjem pojedinačnih naredbi.
- Ovo stanje se predstavlja programskim promenljivama

Stanje programa

```
int x = 0, a = 0;
x = x + 3;
a++;
```

```
int x = 0;
...
int f() { return ++x; }
...
cout << f() << " " << f() << endl;
```

- Promena stanja se najčešće izvršava naredbom dodele (eksplicitna, npr =, ili skrivena, npr ++).
- Naziva se implicitno stanje jer znate da se dodelom menja memorija, ne razmišljate o samom stanju, ono se na neki način podrazumeva
- Potrebno je razumeti korišćenje promenljivih i njihove promene tokom izvršavanja da bi se razumeo program, što je veoma zahtevno za velike programe.

Stanje programa - primeri bočnih efekata

```
//Globalna promenljiva y
int y;

//Definicija funkcije foo:
int foo(int x){
y=0;
return 2*x;
}

//Upotreba funkcije foo
...
y=foo(2);
if(y==foo(2)) ...
...
```

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

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

Stanje u funkcionalnim jezicima

Nepostojanje stanja

Funkcionalni jezici nemaju implicitno stanje, čime je razumevanje efekta rada funkcije značajno olakšano.

- Izvođenje programa svodi se na evaluaciju izraza i to bez stanja.
- Posebno, nepostojanje naredbe dodele i promenljivih u imperativnom smislu (tj nepostojanje promenljivih koje menjaju stanje) ima za posledicu da **iterativne konstrukcije nisu moguće** pa se ponavljanje ostvaruje kroz **rekurziju**.

Rekurzija

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

$$n! = \begin{cases} 1 & , n = 0 \\ n * (n-1)! & , n > 0 \end{cases}$$

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

- Rekurzija je u imperativnim jezicima manje prirodna.
- U funkcionalnim jezicima rekurzija je prirodna i ona je zapravo jedino rešenje za ponavljeno izvršavanje.
- U modernom funkcionalnom programiranju, rekurzija je često skrivena kroz upotrebu osnovnih funkcija višeg reda

Eksplicitno stanje

- Neki problemi se ne mogu rešiti bez stanja (ili se mogu rešiti komplikovano).
- Nepostojanje implicitnog stanja nije nedostatak već se upotreba stanja može ostvariti na drugi način, korišćenjem eksplicitnog stanja.
- Eksplicitno stanje se može "napraviti" po potrebi i koristiti
- Na primer

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

(Ovakva upotreba nije preporučljiva)

Transparentnost referenci

Transparentnost referenci

Vrednost izraza je svuda jedinstveno određena: ako se na dva mesta referencira na isti izraz, onda je vrednost ta dva izraza ista.

- Na primer

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

Svako pojavljivanje izraza x u programu možemo da zamenimo sa fact 5.

Transparentnost referenci

- Svojstvo transparentnosti referenci govori da **redosled naredbi nije bitan**.
- U funkcionalnom programiranju neobavezan je eksplicitni redosled navođenja funkcija — funkcije možemo kombinovati na razne načine, bitno je da se definiše izraz koji predstavlja rešenje problema.
- To naravno nije slučaj kod imperativnih jezika gde je bitan redosled izvođenja operacija, x se nakon inicijalizacije može promeniti, pa stoga imperativni jezici nemaju transparentnost referenci. (Naredba dodele ima propratni efekat, a ona je u osnovi imperatvinog programiranja.)
- Propratni efekat je svaka ona promena implicitnog stanja koja narušava transparentnost referenci programa.

Transparentnost referenci

- Propratni efekti nemaju smisla sa stanovišta matematičkih izračunavanja.
- Transparentnost referenci ima niz pogodnosti i korisna je i u drugim paradigmama.
- Programi sa transparentnim referencama su **formalno koncizni**, prikladni za **formalnu verifikaciju**, **manje podložni greškama** i lakše ih je transformisati, optimizovati i **paralelizovati**.
- Paralelizacija je moguća zbog transparentnosti referenci jer se mogu delovi izraza sračunati nezavisno, a onda se združiti naknadno.
- Međutim, transparentnost referenci ima cenu.

Čisti funkcionalni jezici

- Ukoliko želimo da imamo u potpunosti transparentnost referenci, ne smemo dopustiti nikakve propratne efekte.
- U praksi je to veoma teško jer postoje neki algoritmi koji se suštinski temelje na promeni stanja (npr `random`) dok neke funkcije postoje samo zbog svoji propratnih efekata (npr `scanf`, tj. funkcije za ulaz/izlaz).
- Zbog toga, većina funkcionalnih programskih jezika **dopušta kontrolisane propratne efekte** (tj. imperativnost je prisutna u većoj ili manjoj meri i na različite načine).
- Većina funkcionalnih jezika uključuju naredne imperativne osobine: promenljive (mutable variables) i konstrukte koji se ponašaju kao naredbe dodele.

Čisti funkcionalni jezici

- U zavisnosti od prisutnosti imperativnih osobina, postoji podela na čiste funkcionalne jezike (bez propratnih efekata) i na one koji nisu u čisti.
- Čisto funkcionalni jezici ne dopuštaju baš nikakve propratne efekte, takvi jezici koriste dodatne mehanizme kako bi omogućili izračunavanje sa stanjem, a istovremeno zadržali transparentnost referenci.
- Jako je mali broj čistih funkcionalnih jezika (Haskel, Clean, Miranda).

Funkcionalni jezici

- Funkcionalni jezici su jezici koji podržavaju i ohrabruju funkcionalni stil programiranja, npr SML, čisti jezici Clean, Haskell, Miranda
- Moderni višeparadigmatski programski jezici su Common Lisp, OCaml, Scala, Python, Ruby, F#, Clojure...
- Neki imperativni jezici eksplicitno podržavaju neke funkcijske koncepte (C#, Java) dok su u drugima oni ostvarivi (C, C++).
- Iako mnogi imperativni jezici podržavaju osnovne koncepte funkcionalnog programiranja, čisto funkcionalni podskup takvih jezika je najčešće vrlo slab i nećemo ih nazvati funkcionalnim jezicima.

Neke karakteristike sistema tipova (slika: Nevena Marković)

Zaključivanje tipova

- Zaključivanje tipova može biti statičko (u fazi kompilacije) i dinamičko (u fazi izvršavanja)
- Statičko zaključivanje tipova je manje fleksiblno ali efikasnije
- Dinamičko zaključivanje tipova je fleksiblnije ali manje efikasno
- Funkcionalni jezici mogu imati statičko (npr Haskell) ili dinamičko zaključivanje tipova (npr Elixir)

Zaključivanje tipova kod statički tipiziranih jezika

- Funkcionalni programski jezici su najčešće jako tipizirani jezici — svi tipovi moraju da se poklapaju i nema implicitnih konverzija
- S druge strane, nije neophodno navoditi sve tipove — kompajler je u stanju da često automatski sam zaključi tipove
- Kôd koji se piše najčešće sam po sebi polimorfan, a zaključuju se najopštiji mogući tipovi na osnovu tipskih razreda
- Tipski razredi — tip sa jednakošću, tipovi sa uređenjem, numerički tipovi, celobrojni tipovi, realni tipovi...

Polimorfnost u Haskelu

- Definisanje funkcije

```
Prelude> let uvecaj x = x+1
Prelude> uvecaj 5
6
Prelude> uvecaj 55.5
56.5
Prelude> uvecaj 1234567890123456789012345678901234567890
1234567890123456789012345678901234567891
```

Tipovi u Haskelu

- Tipovi funkcija: funkcija argumente jednog tipa preslikava u argumente drugog tipa npr:

```
Bool -> Bool
Int -> Int
[Char] -> Int
(Int, Int) -> Int
...
```

- Tipovi se mogu navesti prilikom definicije funkcije

Tipske promenljive u Haskelu

- Mogu se koristiti i tipske promenljive a, b... Na primer:
  `length :: [a] -> Int` — ovo označava da a može da bude bilo koji tip
  `reverse :: [a] -> [a]` — tip prve i druge liste moraju da budu iste
- Većina standardnih funkcija je definisano na ovaj način, tj za razne tipove

```
Prelude> length [1,2,3]
3
Prelude> length "abcde"
5
Prelude> :type length
length :: [a] -> Int
```

Tipski razredi u Haskelu

- Prethodno smo upotrebljavali kada imamo različite tipove nad istom strukturom podataka (npr tip Int ili Char, struktura podataka lista)
- Preopterećivanje koristimo kada su nad različitim strukturama podataka definisane iste operacije (na primer, + za cele i realne brojeve)
- Preopterećivanje se ostvaruje preko tipskih razreda
- Tipski razred definiše koje funkcije neki tip mora da implementira da bi pripadao tom razredu
  - EQ Tipovi sa jednakošću $==$ $/=$
  - ORD Tipovi sa uređenjem (nasleđuje EQ) $<, >, <= \dots$
  - NUM Numerički tipovi (nasleđuje ORD) $+, -, *, \dots$
  - INTEGRAL Celobrojni tipovi (nasleđuje NUM) div, mod
  - FRACTIONAL razlomački tipovi (nasleđuje NUM) /, recip

Tipski razredi u Haskelu

- Tipskim razredima se ograničavaju funkcije
- Na primer, `sum :: Num a => [a] -> a`
  Sumiranje može da se definiše samo nad numeričkim tipovime

```
Prelude> sum [1,2,3]
6
Prelude> sum ['a','b','c']

<interactive>:52:1:
    No instance for (Num Char)
      arising from a use of `sum'
    Possible fix: add an instance declaration for (Num Char)
    In the expression: sum ['a', 'b', 'c']
    In an equation for `it': it = sum ['a', 'b', 'c']
```

Tipski razredi u Haskelu

```
Prelude> elem 45 [1,3..] --za 46 se ne bi zaustavilo
True
Prelude> :t elem
elem :: Eq a => a -> [a] -> Bool

Prelude> :t max
max :: Ord a => a -> a -> a
Prelude> max "abc" "cde"
"cde"
```

Tipski razredi u Haskelu

```
map :: (a->b) -> [a] -> [b]

head :: [a] -> a

fst :: (a, b) -> a

inc :: Num a => a -> a

(==) :: (Eq a) => a -> a -> Bool

(>) :: (Ord a) => a -> a -> Bool
```

Polimorfnost konstanti u Haskelu

```
:t 20
20 :: (Num t) => t

ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

Sintaksa

- Prvi funkcionalni jezik, Lisp, koristi sintaksu koja je veoma drugačija od sintakse koja se koristi u imperativnim jezicima.
- **Lisp (lots of irritating silly parentheses)**

```
(define factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Primer - Lisp

Rekurzivno rešenje:

```
(define (factorial n)
        (if (= n 0)
            1
            (* n (factorial (- n 1) ))
        ))
(display (factorial 7))
```

Iterativno rešenje (tj. imitacija imperativnog rešenja):

```
(define (factorial n)
        (define (iter product counter)
                (if (> counter n)
                       product
                       (iter (* counter product) (+ counter 1))
                ))
        (iter 1 1))
(display (factorial 7))
```

Primer

Scheme

```
(define (factorial n)
   (cond ((< n 0) #f)
           ((<= n 1) 1)
           (else (* n (factorial (- n 1))))))
```

Sintaksa

- Lisp, kao multiparadigmatski jezik, dozvoljava i upotrebu konstrukcija koje odgovaraju petljama u imperativnim jezicima

```
(loop for i from 0 to 16
   do (format t "~D! = ~D~%" i (factorial i)))
```

Sintaksa

- Noviji funkcionalni jezici koriste sintaksu koja je slična sintaksi imperativnih jezika.
- *Pattern matching* — poklapanje obrazaca
  U poklapanju obrazaca, pokušavamo da poklopimo vrednost prema obrascu, i, po potrebi, da vežemo promenljivu sa uspešnim poklapanjem.
- *Comprehensions* — skraćenice

Poklapanje obrazaca — Haskell

- Izrazi mogu da se uparuju sa obrascima, rezultat je izraz koji odgovara prvom uparenom obrascu

```
case exp of
    p1 -> e1
    p2 -> e2
    ...
    pn -> en
    _ -> e
```

Sa uparenim obrascem vrši se vezivanje, ako vezivanje nije potrebno koristi se wildecard _

Poklapanje obrazaca — Haskell

```
gcd :: Integer -> Integer -> Integer
gcd x y = case y of
    0 -> x
    _ -> gcd y (x 'mod' y)
```

Drugi način (korišćenjem parametara kao obrasca)

```
gcd :: Integer -> Integer -> Integer
gcd x 0 = x
gcd x y = gcd y (x 'mod' y)
```

Poklapanje obrazaca — Haskell

Paterni se posebno koriste za rad sa listama, pri čemu je

```
[]           prazna lista
[x]          lista sa jednim elementom
[x1, x2]     lista sa tacno dva elementa
...
h : t       neprazna lista
x1 : x2 : t lista sa bar dva elementa
...
```

Poklapanje obrazaca — Haskell

Primer

```
length :: [a] -> Integer
length l = case l of
    [] -> 0
    h:t -> 1 + length t
```

Parametar kao obrazac

```
length :: [a] -> Integer
length [] = 0
length (h:t) = 1 + length t
```

Poklapanje obrazaca — Haskell

Primer

```
PrviPlusTreci :: [Integer] -> Integer
PrviPlusTreci l = case l of
    [] -> 0
    [x1] -> x1
    [x1, _] -> x1
    x1:_:x3:_ -> x1+x3
```

Parametar kao obrazac

```
PrviPlusTreci :: [Integer] -> Integer
PrviPlusTreci [] = 0
PrviPlusTreci [x1] =  x1
PrviPlusTreci [x1, _] = x1
PrviPlusTreci (x1 : _ : x3 : _) = x1 + x3
```

Skraćenice

- Skraćen način zapisa nekih čestih konstrukata - to je sintaksni dodatak zarad produktivnijeg programiranja
- Postoje različite skraćenice u različitim (funkcionalnim) jezicima
- Često su vezane za definisanje listi na način blizak matematičkim definicijama

```
[ 2*x    | x <- [0..], x^2 > 10 ]
[ izraz | generator, filter ]
```

Skraćenice

```
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)]
```

Skraćenice

```
a = [(x,y) | x <- [1..5], y <- [3..5]]
-- [(1,3),(1,4),(1,5),(2,3),(2,4) ...

b = [(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)]

c = [x+2*x+x/2 | x <- [1,2,3,4]]
-- [3.5,7.0,10.5,14.0]

deljiviSaTri a b = [x | x<-[a..b], x 'mod' 3 == 0]
```

Semantika

- Semantika programskog jezika opisuje proces izvršavanja programa na tom jeziku.
- Semantika može da se opiše formalno i neformalno
- Uloga semantike:
    - programer može da razume kako se program izvršava pre njegovog pokretanja kao i šta mora da obezbedi prilikom kreiranja kompilatora
    - razumevanje karakteristika programskog jezika
    - dokazivanje svojstava određenog programskog jezika
- Mogu se razmatrati različita semantička svojstva jezika

Striktna i nestriktna semantika

- Jedno interesantno semantičko svojstvo je striktnost semantike
- Striktna i nestriktna semantika (izračunavanje vođeno potrebama).
- Izraz je striktan ako nema vrednost kad bar jedan od njegovih operanada nema vrednost.
- Izraz je nestriktan kad može da ima vrednost čak i ako neki od njegovih operanada nema vrednost.

Striktna i nestriktna semantika

- Od konkretnog izraza može da zavisi da li će i na osnovu koliko poznatih argumenata biti izračunata njegova vrednost.
- Na primer, $a\&b$ ima vrednost $\perp$ ako a ima vrednost $\perp$ - tj nije važna vrednost izraza b (može se razmatrati nestriktna semantika). U striktnoj semantici, ako vrednost za b nije poznata, vrednost konjunkcije se ne može izračunati.
- Razmotrite naredne primere: $a||b$, $a+b$, $a-b$, $a \cdot b$, a/b...

Striktna i nestriktna semantika

- Kod striktne semantike, prvo se izračunaju vrednosti svih operanada, pa se onda izračunava vrednost izraza.
- Kod nestriktne semantike izračunavanje operanada se odlaže sve dok te vrednosti ne budu neophodne. Ta strategija je poznata kao zadržano ili lenjo izračunavanje (lazy evaluation).
- Nestriktna semantika omogućava kreiranje beskonačnih struktura i izraza.

Striktna i nestriktna semantika

- Tehnika prenosa parametara koja podržava striktnu semantiku je prenos parametara po vrednosti.
- Kod nestriktne semantike izračunavanje argumenata funkcije se odlaže sve dok te vrednosti ne budu neophodne tako da to odgovara prenosu parametara po potrebi (*call by need*).

Striktna i nestriktna semantika

- Semantika je striktna ako je svaki izraz tog jezika striktan.
- Semantika nije striktna ukoliko se dozvole nestriktni izrazi .
- Većina funkcionalnih jezika ima striktnu semantiku, npr Lisp, OCaml, Scala, ali Miranda i Haskel imaju nestriktnu semantiku.

Primer rešenja sa beskonačnom listom

```
Prelude> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Primer rešenja sa beskonačnom listom

```
Prelude> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

166650

Suma svih neparnih kvadrata brojeva manjih od 10000

Implementacija jezika

- Funkcionalnim programiranjem se definišu izrazi čije se vrednosti evaluiraju, vrši se oponašanje matematičkih funkcija koje je na visokom nivou apstrakcije i to je veoma udaljeno od konkretnog hardvera računara na kojem se program izvršava
- Bez obzira odakle krećemo, moramo stići do asemblera i izvršnog koda
- Izvršni program uvek odgovara hardveru računara, a asembler je po prirodi imperativni jezik
- Proces kompilacije funkcionalnih jezika je zbog toga veoma složen, tj veliki je izazov napraviti kompajler koji podržava transparentnost referenci, beskonačne strukture podataka, nestriktnu semanitiku i sve pomenute koncepte

Implementacija jezika

- Zbog izazova da se napravi efikasan kompajler, funkcionalni jezici su dugo bili najčešće interpretirani jezici
- Realizacija prvog kompajlera kroz SECD mašine, ali se to više ne korsiti
- Realizacija Haskel kompajlera se zasniva na konceptu G-mašine

Sakupljač otpadaka

- U funkcionalnim jezicima ne postoje promenljive pa zato ne možemo ni da razmataramo njihov životni vek
- Ipak, interno, podaci moraju da se čuvaju, prostor u memoriji se alocira na hipu
- O dealokaciji memorije brine se sakupljač otpadaka
- Postoje razne vrste sakupljača otpadaka, Haskel koristi generacijski sakupljač otpadaka (objekti u memoriji se dele po starosti a očekivanje je da će mlađi objekti biti brže iskorišćeni i uklonjeni)

Prednosti i mane funkcionalnog programiranja

Postoji opšta debata na temu prednosti i mana funkcionalnog programiranja. Naredni razlozi se često navode kao prednosti funkconalnog programiranja

- Stanje i propratni efekti — za velike programe teško je pratiti stanje i razumeti propratne efekte, lakše je kada imamo uvek isto ponašanje funkcija,

- Pralelno programiranje — jednostavno i bezbedno konkurentno programiranje je posledica transparentnih referenci,

Prednosti i mane funkcionalnog programiranja

Postoji opšta debata na temu prednosti i mana funkcionalnog programiranja. Naredni razlozi se često navode kao prednosti funkconalnog programiranja ali istovremeno i kao mane:

- Stanje i propratni efekti — za velike programe teško je pratiti stanje i razumeti propratne efekte, lakše je kada imamo uvek isto ponašanje funkcija, ali svet koji nas okružuje je pun promena i različitih stanja i nije prirodno da koncepti jezika budu u suportnosti sa domenom koji se modeluje.
- Pralelno programiranje — jednostavno i bezbedno konkurentno programiranje je posledica transparentnih referenci, međutim, rad sa podacima koji se ne menjaju dovodi do mogućeg rada sa podacima koji su umeđuvremenu izmenjeni, dok rad sa podacima koji se menjaju jeste komplikovaniji i zahteva kodiranje kompleksne logike ali omogućava rad sa podacima koji se menjaju (šta je bitnije?)

Prednosti i mane funkcionalnog programiranja

Naredni razlozi se često navode kao prednosti funkconalnog programiranja

- Stil programiranja — programi su često kraći i lakši za čitanje,
- Produktivnost programera — produktivnost je veća,

Prednosti i mane funkcionalnog programiranja

Naredni razlozi se često navode kao prednosti funkconalnog programiranja ali istovremeno i kao mane:

- Stil programiranja — programi su često kraći i lakši za čitanje, treba znati pročitati funkcionalni kôd.
- Produktivnost programera — produktivnost je veća, ali većina programera ne gradi nove sisteme već rade na održavanju starih koji su pisani u drugim (imperativnim) jezicima.

Diskusije na temu funkcionalnog programiranja

- Efikasnost se dugo navodila kao mana, ali zapravo efikasnost odavno nije problematična. Na primer, merenja pokazuju da je Haskel na nekim problemima jednako efikasan kao C.
- Često se navodi da je funkcionalno programiranje lako/teško za učenje.

Prednosti funkcionalnog programiranja

Postoje razne prednosti funkcionalnog programiranja.

- Za funkcionalne programe lakše je konstruisati matematički dokaz ispravnosti.
- Stil programiranja nameće razbijanje koda u manje delove koji imaju jaku koheziju i izraženu modularnost.

Prednosti funkcionalnog programiranja

- Testiranje i debagovanje je jednostavnije —
    - Testiranje je jednostavnije jer je svaka funkcija potencijalni kandidat za unit testove. Funkcije ne zavise od stanja sistema što olakšava sintezu test primera i proveru da li je izlaz odgovarajući.
    - Debagovanje je jednostavnije (osim kada je nestriktna semantika u pitanju) jer su funkcije uglavnom male i jasno specijalizovane. Kada program ne radi, svaka funkcija je interfejs koji se može proveriti tako da se brzo izoluje koja funkcija je odgovorna za grešku.
- Mogu se jednostavno graditi biblioteke funkcija.

Razvoj funkcionalnog programiranja

- Ono što je svakako izvesno je da se velike pare ulažu u razvoj i podršku funkcionalnog stila programiranja (npr. Scala, Kotlin, funkcionalni koncepti u skript jezicima).
- Postoji velika cena prelaska na funkcionalno programiranje i za to treba vremena.
- Pomeranje prema programiranju koje je više u funkcionalnom stilu ide postepeno.

Pregled

Formalni model definisanja algoritma

- Lambda račun — λ-calculus je formalni model izračunljivosti funkcija
- Alonzo Church 1930
- Lambda račun se zasniva na apstrakciji i primeni funkcija korišćenjem vezivanja i supstitucije (zamene).
- Lambda račun funkcije tretira kao izraze koji se postepeno transformišu do rešenja, tj funkcija definiše algoritam.
- Lambda račun je formalni model definisanja algoritma.

Prvi funkcionalni jezik

- Iako to nije bila primarna ideja, tj lambda račun je razvijen kao jedan formalizam za izračunavanje bez ideje da to treba da se koristi za programiranje, danas se lambda račun smatra prvim funkcionalnim jezikom.
- Ekspresivnost lambda računa je ekvivalentna ekspresivnosti Tjuringovih mašina (1937)
- Lambda račun naglašava pravila za transformaciju izraza i ne zamara se arhitekturom mašine koja to može da ostvari
- Svi moderni funkcionalni jezici su zapravo samo sintaksno ulepšane varijante lambda računa.
- Ekspresivnost Haskela je ekvivalentna ekspresivnosti lambda računa.

Lambda račun sa i bez tipova

- Postoje više vrsta lambda račun, tj lambda račun bez tipova i lambda račun sa tipovima.
- Istorijski, prvi je nastao netipizirani lambda račun, tj domen funkcije nije ugrađen u lambda račun.
- Tipizirani lambda račun (1940) je vrsta lambda računa koja daje jedno ograničenje primene lambda računa, tj funkcije mogu da se primenjuju samo na odgovarajući tip podataka.
- Tipizirani lambda račun igra važnu ulogu u dizajnu sistema tipova programskih jezika.
- Osim u programskim jezicima, lambda račun je važan i u teoriji dokaza, filozofiji, lingvistici.

Bezimene funkcije

- Zadatak definisanja funkcije može se razdvojiti od zadatka imenovanja funkcije.
- Na primer:
  - Funkcija $sum(x, y) = x + y$ može da se definiše i bez njenog imenovanja kao funkcija koje promenljive x i y preslikava u njihov zbir, tj $(x, y) \rightarrow x + y$
  - Funkcija $id(x) = x$ može da se definiše kao $x \rightarrow x$
- Lambda račun daje osnove za definisanje bezimenih funkcija.
- Lambda izraz definiše parametre i preslikavanje funkcije, ne i ime funkcije, pa se takođe zove anonimna funkcija, ili bezimena funkcija.

Sintaksa — apstrakcija

- Lambda izraz, funkcijska apstrakcija, anonimna funkcija, bezimena funkcija.
- Sintaksa lambda izraza

$$\lambda promenljiva.telo$$

sa značenjem $promenljiva \rightarrow telo$

- Na primer:
  - $\lambda x.x + 1$ —

Sintaksa — apstrakcija

- Lambda izraz, funkcijska apstrakcija, anonimna funkcija, bezimena funkcija.
- Sintaksa lambda izraza

$$\lambda promenljiva.telo$$

sa značenjem *promenljiva* $\rightarrow$ *telo*

- Na primer:
  - $\lambda x.x + 1$ — intuitivno, funkcija inkrementiranja, $x \rightarrow x + 1$
  - $\lambda x.x$ —

Sintaksa — apstrakcija

- Lambda izraz, funkcijska apstrakcija, anonimna funkcija, bezimena funkcija.
- Sintaksa lambda izraza

$$\lambda promenljiva.telo$$

sa značenjem $promenljiva \rightarrow telo$

- Na primer:
  - $\lambda x.x + 1$ — intuitivno, funkcija inkrementiranja, $x \rightarrow x + 1$
  - $\lambda x.x$ — intuitivno, funkcija identiteta, $x \rightarrow x$
  - $\lambda x.x \cdot x + 3$ —

Sintaksa — apstrakcija

- Lambda izraz, funkcijska apstrakcija, anonimna funkcija, bezimena funkcija.
- Sintaksa lambda izraza

$$\lambda promenljiva.telo$$

sa značenjem $promenljiva \rightarrow telo$

- Na primer:
  - $\lambda x.x + 1$ — intuitivno, funkcija inkrementiranja, $x \rightarrow x + 1$
  - $\lambda x.x$ — intuitivno, funkcija identiteta, $x \rightarrow x$
  - $\lambda x.x \cdot x + 3$ — intuitivno, kvadriranje i uvećanje za 3, $x \rightarrow x \cdot x + 3$

Sintaksa — primena

- Lambda izraz se može primenjivati na druge izraze
- Sintaksa

$$(\lambda promenljiva.telo)izraz$$

— intuitivno, primena odgovara pozivu funkcije

- Na primer:
  - $(\lambda x.x + 1)5$
  - $(\lambda x.x \cdot x + 3)((\lambda x.x + 1)5)$

Sintaksa — lambda termovi

- Validni (ispravni) lambda izrazi se nazivaju lambda termovi.
- Lambda termovi se sastoje od promenljivih, simbola apstrakcije λ, tačke . i zagrada
- Induktivna definicija za građenje lambda termova:

  Promenljive Promenljiva je validni lambda term

  λ-apstrakcija Ako je t lambda term, a x promenljiva, onda je $\lambda x.t$ lambda term

  λ-primena Ako su t i s lambda termovi, onda je $(t\ s)$ lambda term

  Lambda termovi se mogu konstruisati samo konačnom primenom prethodnih pravila.

Sintaksa

- Prirodni brojevi se mogu definisati korišćenjem osnovne definicije lambda računa.
- Ukoliko lambda račun ne uključuje konstante u definiciji, onda se naziva čist.

Sintaksa

- Radi jednostavnosti, numerali se često podrazumevaju i koriste već u okviru same definicije lambda termova
  <con> :: = konstanta
  <id> ::= identifikator

  <exp> ::= <id> | <con> | λ<id>.<exp> | <exp><exp> | (<exp>)
- Ukoliko lambda račun uključuje konstante u definiciji, onda se naziva primenjen.

Sintaksa

- Slično, korišćenjem osnovnog lambda računa mogu se definisati i aritmetičke funkcije.
- Radi jednostavnosti, u okviru lambda termova koristimo aritmetičke funkcije imenovane na standardni način, kao što su +,-,* i slično.

Asocijativnost

- U okviru lambda izraza zagrade su važne.
- Na primer, termovi $\lambda x.((\lambda x.x + 1)x)$ i $(\lambda x.(\lambda x.x + 1))x$ su različiti termovi.

Asocijativnost

- Da bi se smanjila upotreba zagrada, postoje pravila asocijativnosti za primenu i apstrakciju.
- Primena funkcije je levo asocijativna. tj umesto $(e_1\, e_2)e_3$ možemo kraće da pišemo $e_1\, e_2\, e_3$
- Apstrakcija je desno asocijativna, tj umesto $\lambda x.(e_1\, e_2)$ pišemo skraćeno $\lambda x.e_1\, e_2$
- I sekvenca apstrakcija može da se skrati, npr $\lambda x.\lambda y.\lambda z.e$ se skraćeno zapisuje kao $\lambda xyz.e$

Asocijativnost

- Na primer, za izraz

$$\lambda xy.x(\lambda z.zy)yy\lambda z.xy(xz)$$

ekvivalentan izraz sa zagradama je

Asocijativnost

- Na primer, za izraz

$$\lambda xy.x(\lambda z.zy)yy\lambda z.xy(xz)$$

ekvivalentan izraz sa zagradama je

$$\lambda x.(\lambda y.(((((x(\lambda z.zy)))y)y)\lambda z.((xy)(xz))))$$

Haskell kao kalkulator

- Primena funkcije se piše bez zagrada, kao u lambda računu

```
Prelude> sqrt 2
1.4142135623730951
```

- Zagrade mogu da budu potrebne za grupisanje argumenata

```
Prelude> sqrt (abs (-2))
1.4142135623730951
```

- Primena funkcije je levo asocijativna, tako da bez zagrada, prethodni izraz bi se tumačio kao primena sqrt na funkciju abs (što interpreter prijavi kao grešku jer se ne poklapaju tipovi)

Slobodne i vezane promenljive

- U okviru lambda računa ne postoji koncept deklaracije promenljive.
- Promenljiva može biti vezana i slobodna (tj nije vezana).
- Na primer, u termu $\lambda x.x + y$ promenljiva x je vezana a promenljiva y je slobodna promenljiva.
- Intuitivno, slobodne promenljive u termu su one promenljive koje nisu vezane lambda apstrakcijom.

Slobodne i vezane promenljive

- Induktivna definicija:

  Promenljive Slobodna promenljiva terma x je samo x.

  Apstrakcija Skup slobodnih promenljivih terma $\lambda x.t$ je skup slobodnih promenljivih terma t bez promenljive x.

  Primena Skup slobodnih promenljivih terma $(t\ s)$ je unija skupova slobodnih promenljivih terma t i terma s.

- Na primer, term $\lambda x.x$ nema slobodnih promenljivih, dok term $\lambda x.x \cdot y$ ima slobodnu promenljivu y.

α ekvivalentnost

- α ekvivalentnost se definiše za lambda termove, sa ciljem da se uhvati intuicija da izbor imena vezane promenljive u lambda računu nije važan.
- Na primer, termovi $\lambda x.x$ i $\lambda y.y$ su α-ekvivalentni jer oba predstavljaju istu funkciju, tj identitet.
- Na primer, termovi $\lambda x.x$ i $\lambda x.y$ nisu α-ekvivalentni jer prvi predstavlja funkciju identiteta, a drugi konstantnu funkciju.
- S druge strane, termovi x i y nisu α-ekvivalentni jer nisu vezani u okviru lambda apstrakcije.

α ekvivalentnost

Zaokružiti slovo ispred alfa ekvivalentnih termova:

1. x i y
2. $\lambda k.5 + k/2$ i $\lambda h.5 + h/2$
3. $\lambda k.5 + k/2$ i $\lambda h.5 + h/3$
4. $\lambda a.a \cdot y - 1$ i $\lambda b.b \cdot z - 1$
5. $\lambda z.z \cdot y - 1$ i $\lambda x.x \cdot z - 1$
6. $\lambda ij.i - j \cdot 3$ i $\lambda mn.m - n \cdot 3$
7. $\lambda ij.i - j \cdot 3$ i $\lambda nm.m - n \cdot 3$

Izvođenje — redukcija

- Uveli smo sintaksu, sada treba da opišemo transformacije koje možemo da izvršimo.
- Za transformacije se koriste izvođenja (redukcije, konverzije).
- Postoje razne vrste redukcija.
- Redukcije se nazivaju slovima grčkog alfabeta.
- Redukcije daju uputstva kako transformisati izraze iz početnog stanja u neko finalno stanje.

δ redukcija

- Najprostiji tip lambda izraza su konstante — one se ne mogu dalje transformisati.
- δ redukcija se označava sa $\rightarrow_\delta$ i odnosi se na trasformaciju funkcija koje kao argumente sadrže konstante.
- Na primer, $3 + 5 \rightarrow_\delta 8$
- Ukoliko je jasno o kojoj redukciji je reč, onda se piše samo $\rightarrow$

α redukcija ili preimenovanje

- α redukcija ili α preimenovanje, dozvoljava da se promene imena vezanim promenljivama.
- Na primer, α redukcija izraza $\lambda x.x$ može da bude u $\lambda y.y$
- Termovi koji se razlikuju samo po α konverziji su α ekvivalentni.
- Na primer $\lambda x.yx = \lambda z.yz = \lambda a.ya\ldots$ (y nije vezana promenljiva i za nju ne možemo da vršimo preimenovanje)
- Alfa preimenovanje je nekada neophodno da bi se izvršila beta redukcija

α redukcija

- α redukcija nije u potpunosti trivijalna, treba voditi računa.
- Na primer, $\lambda x.\lambda x.x$ može da se svede na $\lambda y.\lambda x.x$ ali ne i na $\lambda y.\lambda x.y$
- Ukoliko funkciju $\lambda x.\lambda x.x$ primenimo na npr broj 3, dobijamo preslikavanje kojim se broj tri preslikava u funkciju identiteta
- Ukoliko funkciju $\lambda y.\lambda x.x$ primenimo na npr broj 3, ponovo dobijamo preslikavanje kojim se broj tri preslikava u funkciju identiteta
- Ukoliko funkciju $\lambda y.\lambda x.y$ primenimo na npr broj 3, dobijamo preslikavanje kojim se broj tri preslikava u funkciju konstantnog preslikavanja u broj 3

α redukcija

- Takođe, alfa redukcijom ne sme da se promeni ime promenljive tako da bude uhvaćeno drugom apstrakcijom, na primer $\lambda x.\lambda y.x$ smemo da zamenimo sa $\lambda z.\lambda y.z$ ali ne smemo da zamenimo sa $\lambda y.\lambda y.y$

β-redukcija — primena funkcije

- Kada funkciju primenimo na neki izraz, želeli bi da možemo da izračunamo vrednost funkcije.
- U okviru lambda računa, to se sprovodi β-redukcijom:

$$(\lambda promenljiva.telo)izraz \rightarrow_{\beta} [izraz/promenljiva]telo$$

β-redukcija u telu lambda izraza formalni argument zamenjuje aktuelnim argumentom i vraća telo funkcije (dakle svako pojavljivanje promenljive u telu se zamenjuje sa datim izrazom)

β-redukcija

- Na primer:
  - $(\lambda x.x + 1)5 \rightarrow_\beta [5/x](x + 1) = 5 + 1 \rightarrow_\delta 6$
  - $(\lambda x.x \cdot x + 3)((\lambda x.x + 1)5) \rightarrow_\beta [6/x](x \cdot x + 3) = 6 \cdot 6 + 3 \rightarrow_\delta 39$
- Višestruka primena β-redukcije (oznaka $\twoheadrightarrow_\beta$)
- Na primer:
  - $(\lambda x.x \cdot x + 3)((\lambda x.x + 1)5) \twoheadrightarrow_\beta 39$
- Primenjujemo β-redukciju sve dok možemo — to odgovara izračunavanju vrednosti funkcije.

Supstitucija

- Prethodni primeri su bili jednostavni jer je primena obuhvatala konstante i jednostavne lambda izraze.
- Da bi se izmene vršile na ispravan način, potrebno je precizno definisati pojam zamene — supstitucije.
- Ukoliko postoji problem kolizije imena, potrebno je uraditi alfa preimenovanje kako bi se izbegla kolizija.
- Na primer, $(\lambda x.x(\lambda x.x))(\lambda x.x\ x)$
- Supstitucija se definiše rekurzivno po strukturi terma.

Supstitucija

- Supstitucija $[I/P]T$ je proces zamene svih slobodnih pojavljivanja promenljive P u telu lambda izraza T izrazom I na sledeći način (x i y su promenljive, a M i N lambda izrazi):

Promenljive	$[N/x]x = N$
	$[N/x]y = y$ pri čemu je $x \neq y$
Apstrakcija	$[N/x](\lambda x.M) = \lambda x.M$
	$[N/x](\lambda y.M) = \lambda y.([N/x]M)$ ukoliko je $x \neq y$ i y ne pripada skupu slobodnih promenljivih za N
Primena	$[N/x](M_1 M_2) = ([N/x])(M_1)([N/x])(M_2)$

- β-redukcija se definiše preko naredne supstitucije: $(\lambda promenljiva.telo)izraz \rightarrow_\beta [izraz/promenljiva]telo$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \to (\lambda x.x+3)(4+5) \to (\lambda x.x+3)9 \to$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \rightarrow (\lambda x.x+3)(4+5) \rightarrow (\lambda x.x+3)9 \rightarrow (9+3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \to (\lambda x.x + 3)(4 + 5) \to (\lambda x.x + 3)9 \to (9 + 3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \to (\lambda x.x + 3)(4 + 5) \to (\lambda x.x + 3)9 \to (9 + 3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \to (\lambda x'.x'\ (\lambda x.x))y \to$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \to (\lambda x.x + 3)(4 + 5) \to (\lambda x.x + 3)9 \to (9 + 3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \to (\lambda x'.x'\ (\lambda x.x))y \to y(\lambda x.x)$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow (\lambda x''.x''\ x'')(\lambda x'.x') \rightarrow$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow (\lambda x''.x''\ x'')(\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'.x') \rightarrow$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow (\lambda x''.x''\ x'')(\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'''.x''') \rightarrow$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \rightarrow (\lambda x.x+3)(4+5) \rightarrow (\lambda x.x+3)9 \rightarrow (9+3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow (\lambda x''.x''\ x'')(\lambda x'.x') \rightarrow$ $(\lambda x'.x')\ (\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'''.x''') \rightarrow (\lambda x'''.x''') \rightarrow$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \to (\lambda x.x + 3)(4 + 5) \to (\lambda x.x + 3)9 \to (9 + 3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \to (\lambda x'.x'\ (\lambda x.x))y \to y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \to (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \to (\lambda x''.x''\ x'')(\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'''.x''') \to (\lambda x'''.x''') \to (\lambda x.x)$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \rightarrow (\lambda x.x + 3)(4 + 5) \rightarrow (\lambda x.x + 3)9 \rightarrow (9 + 3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow (\lambda x''.x''\ x'')(\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'''.x''') \rightarrow (\lambda x'''.x''') \rightarrow (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \rightarrow$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \to (\lambda x.x+3)(4+5) \to (\lambda x.x+3)9 \to (9+3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \to (\lambda x'.x'\ (\lambda x.x))y \to y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \to (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \to (\lambda x''.x''\ x'')(\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'''.x''') \to (\lambda x'''.x''') \to (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \to (\lambda x.(\lambda y.y\ x))(\lambda z.u) \to$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \to (\lambda x.x + 3)(4 + 5) \to (\lambda x.x + 3)9 \to (9 + 3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \to (\lambda x'.x'\ (\lambda x.x))y \to y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \to (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \to (\lambda x''.x''\ x'')(\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'''.x''') \to (\lambda x'''.x''') \to (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \to (\lambda x.(\lambda y.y\ x))(\lambda z.u) \to \lambda y.y(\lambda z.u)$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \to (\lambda x.x+3)(4+5) \to (\lambda x.x+3)9 \to (9+3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \to (\lambda x'.x'\ (\lambda x.x))y \to y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \to (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \to (\lambda x''.x''\ x'')(\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'''.x''') \to (\lambda x'''.x''') \to (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \to (\lambda x.(\lambda y.y\ x))(\lambda z.u) \to \lambda y.y(\lambda z.u)$
- $(\lambda x.x\ x)(\lambda z.u) \to$

Beta redukcija

- $(\lambda x.x + 3)((\lambda x.x + 5)4) \to (\lambda x.x + 3)(4 + 5) \to (\lambda x.x + 3)9 \to (9 + 3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \to (\lambda x'.x'\ (\lambda x.x))y \to y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \to (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \to (\lambda x''.x''\ x'')(\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'''.x''') \to (\lambda x'''.x''') \to (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \to (\lambda x.(\lambda y.y\ x))(\lambda z.u) \to \lambda y.y(\lambda z.u)$
- $(\lambda x.x\ x)(\lambda z.u) \to (\lambda z.u)(\lambda z.u) \to$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \rightarrow (\lambda x.x+3)(4+5) \rightarrow (\lambda x.x+3)9 \rightarrow (9+3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow (\lambda x''.x''\ x'')(\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'''.x''') \rightarrow (\lambda x'''.x''') \rightarrow (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \rightarrow (\lambda x.(\lambda y.y\ x))(\lambda z.u) \rightarrow \lambda y.y(\lambda z.u)$
- $(\lambda x.x\ x)(\lambda z.u) \rightarrow (\lambda z.u)(\lambda z.u) \rightarrow (\lambda z.u)(\lambda z'.u) \rightarrow$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \rightarrow (\lambda x.x+3)(4+5) \rightarrow (\lambda x.x+3)9 \rightarrow (9+3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow (\lambda x''.x''\ x'')(\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'''.x''') \rightarrow (\lambda x'''.x''') \rightarrow (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \rightarrow (\lambda x.(\lambda y.y\ x))(\lambda z.u) \rightarrow \lambda y.y(\lambda z.u)$
- $(\lambda x.x\ x)(\lambda z.u) \rightarrow (\lambda z.u)(\lambda z.u) \rightarrow (\lambda z.u)(\lambda z'.u) \rightarrow u$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \to (\lambda x.x+3)(4+5) \to (\lambda x.x+3)9 \to (9+3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \to (\lambda x'.x'\ (\lambda x.x))y \to y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \to (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \to (\lambda x''.x''\ x'')(\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'''.x''') \to (\lambda x'''.x''') \to (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \to (\lambda x.(\lambda y.y\ x))(\lambda z.u) \to \lambda y.y(\lambda z.u)$
- $(\lambda x.x\ x)(\lambda z.u) \to (\lambda z.u)(\lambda z.u) \to (\lambda z.u)(\lambda z'.u) \to u$
- $(\lambda x.x\ x)(\lambda x.x\ x) \to$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \rightarrow (\lambda x.x+3)(4+5) \rightarrow (\lambda x.x+3)9 \rightarrow (9+3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow (\lambda x''.x''\ x'')(\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'''.x''') \rightarrow (\lambda x'''.x''') \rightarrow (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \rightarrow (\lambda x.(\lambda y.y\ x))(\lambda z.u) \rightarrow \lambda y.y(\lambda z.u)$
- $(\lambda x.x\ x)(\lambda z.u) \rightarrow (\lambda z.u)(\lambda z.u) \rightarrow (\lambda z.u)(\lambda z'.u) \rightarrow u$
- $(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow (\lambda x'.x'\ x')(\lambda x.x\ x) \rightarrow$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \rightarrow (\lambda x.x+3)(4+5) \rightarrow (\lambda x.x+3)9 \rightarrow (9+3) \rightarrow 12$
- $(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y'.y')(\lambda y.y) \rightarrow \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \rightarrow (\lambda x'.x'\ (\lambda x.x))y \rightarrow y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \rightarrow (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \rightarrow (\lambda x''.x''\ x'')(\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'.x') \rightarrow (\lambda x'.x')\ (\lambda x'''.x''') \rightarrow (\lambda x'''.x''') \rightarrow (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \rightarrow (\lambda x.(\lambda y.y\ x))(\lambda z.u) \rightarrow \lambda y.y(\lambda z.u)$
- $(\lambda x.x\ x)(\lambda z.u) \rightarrow (\lambda z.u)(\lambda z.u) \rightarrow (\lambda z.u)(\lambda z'.u) \rightarrow u$
- $(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow (\lambda x'.x'\ x')(\lambda x.x\ x) \rightarrow (\lambda x.x\ x)(\lambda x.x\ x) \rightarrow$

Beta redukcija

- $(\lambda x.x+3)((\lambda x.x+5)4) \to (\lambda x.x+3)(4+5) \to (\lambda x.x+3)9 \to (9+3) \to 12$
- $(\lambda x.x)(\lambda y.y) \to (\lambda y.y)$
- $(\lambda x.x\ x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to (\lambda y'.y')(\lambda y.y) \to \lambda y.y$
- $(\lambda x.x\ (\lambda x.x))y \to (\lambda x'.x'\ (\lambda x.x))y \to y(\lambda x.x)$
- $(\lambda x.x\ (\lambda x.x))(\lambda x.x\ x) \to (\lambda x.x\ (\lambda x'.x'))(\lambda x''.x''\ x'') \to (\lambda x''.x''\ x'')(\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'.x') \to (\lambda x'.x')\ (\lambda x'''.x''') \to (\lambda x'''.x''') \to (\lambda x.x)$
- $(\lambda x.\lambda y.y\ x)(\lambda z.u) \to (\lambda x.(\lambda y.y\ x))(\lambda z.u) \to \lambda y.y(\lambda z.u)$
- $(\lambda x.x\ x)(\lambda z.u) \to (\lambda z.u)(\lambda z.u) \to (\lambda z.u)(\lambda z'.u) \to u$
- $(\lambda x.x\ x)(\lambda x.x\ x) \to (\lambda x'.x'\ x')(\lambda x.x\ x) \to (\lambda x.x\ x)(\lambda x.x\ x) \to \ldots$

Eta redukcija

- η redukcija se može shvatiti kao funkcijsko proširenje. Ideja je da se uhvati intuicija po kojoj su dve funkcije jednake ukoliko imaju identično spoljašnje ponašanje, odnosno ako se za sve vrednosti evaluiraju u iste rezultate.
- Izrazi $\lambda x.\ f\ x$ i f označavaju istu funkciju ukoliko se x ne javlja kao slobodna promenljiva u f. To je zato što ako se funkcija $\lambda x.\ f\ x$ primeni na neki izraz e, onda je to isto što i $f\ e$, i to važi za svaki izraz e.
- To zapisujemo ovako

$$\lambda x.\ f\ x \rightarrow_{\eta} f$$

sa značenjem da se izraz $\lambda x.\ f\ x$ redukuje u izraz f

Funkcije višeg reda — funkcija kao argument

- Funkcije višeg reda su funkcije koje kao argument ili kao povratnu vrednost imaju funkciju.
- Funkcija koja očekuje funkciju tu će funkciju primeniti negde u okviru svog tela.
- Na primer:
  - $\lambda x.(x\ 2) + 1$ — lambda izraz kod kojeg x primenimo na dvojku (dakle x je nekakva funkcija), a onda na to dodamo broj 1.
    $(\lambda x.(x\ 2) + 1)(\lambda x.x + 1) \rightarrow_\beta (\lambda x.x + 1)2 + 1 \twoheadrightarrow_\beta 4$
    $(\lambda x.(x\ 2) + 1)(\lambda x.x) \twoheadrightarrow_\beta 3$

Funkcije višeg reda — funkcija kao povratna vrednost

- Funkcija koja vraća funkciju će u svom telu sadržati drugi lambda izraz
- Na primer:
  - $\lambda x.(\lambda y.2 \cdot y + x)$
    $(\lambda x.(\lambda y.2 \cdot y + x))5 \rightarrow_{\beta} \lambda y.2 \cdot y + 5$
  - Dakle, kada se početni lambda izraz primeni na 5, onda 5 ulazi u izraz na mesto x, i dobijamo novu funkciju kao rezultat.
- Kako je ovakva upotreba česta, uvedena je sintaksna skraćenica koju smo ranije pominjali, tj $\lambda x.(\lambda y.2 \cdot y + x)$ pišemo kao $\lambda xy.2 \cdot y + x$
- Dakle umesto zagrada, nižemo argumente.

Funkcije sa više argumenata

- Lambda izrazi ograničeni su samo na jedan argument.
- Kako definisati funkcije sa više argumenata, npr $f(x, y) = x + y$?

Funkcije sa više argumenata

- Lambda izrazi ograničeni su samo na jedan argument.
- Kako definisati funkcije sa više argumenata, npr $f(x, y) = x + y$?
- Bilo koja funkcija sa više argumenata može se definisati pomoću funkcije sa samo jednim argumentom (rezultat iz 1924. godine).
- Postupak se naziva Curryjev postupak (po američkom matematičaru Haskell Brooks Curry).
- Ideja: funkcija koja treba da uzme dva argumenta će prvo da uzme jedan argument, od njega će napraviti funkciju koja će onda da uzme drugi argument.

Funkcije sa više argumenata

- Funkcija oblika $f(x_1, x_2, ...x_n) = telo$ u lambda računu se definiše kao $\lambda x_1.(\lambda x_2.(...(\lambda x_n.telo)))$ odnosno skraćeno kao $\lambda x_1 x_2..x_n.telo$
- Na primer, $f(x, y) = x + y$ se definiše kao $\lambda xy.x + y$

$$((\lambda xy.x + y)2)6 \rightarrow_\beta \lambda y.(2 + y)6 \rightarrow_\beta 2 + 6 \rightarrow_\delta 8$$

Karijev postupak

- Ako funkcija uzima više argumenata, koristi se Curryjev postupak

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

- Primena funkcije max na 3 daje kao rezultat funkciju koja se primenjuje na broj 10, čiji je rezultat broj 10.
- Zbog leve asocijativnosti ne moraju da se pišu zagrade i dovoljno je max 3 10
- Kod sqrt moraju zagrade

Karijeve funkcije

- Karijeve funkcije argumente uzimaju jedan po jedan što ih čini veoma fleksibilnim
- Karijeve funkcije se mogu delimično evaluirati, tako da se definišu nove funkcije kojima su neki argumenti početne funkcije fiksirani

Karijeve funkcije

Delimičnom evaluacijom fiksiraju se levi argumenti funkcije

```
pomnozi :: Int -> Int -> Int -> Int
pomnozi i j k = i*j*k
```

```
*Main> let p3 = pomnozi 3
*Main> :t p3
p3 :: Int -> Int -> Int
*Main> let p34 = p3 4
*Main> :t p34
p34 :: Int -> Int
*Main> p34 2
24
*Main> p34 5
60
```

Normalni oblik

- Višestrukom beta redukcijom izračunavamo vrednost izraza i zaustavljamo se tek onda kada dalja beta redukcija nije moguća.
- Tako dobijen lambda izraz naziva se **normalni oblik** i on intuitivno odgovara vrednosti polaznog izraza.
- Vrednost polaznog izraza ne mora biti broj, to može biti i neki drugi lambda izraz na koji se više ne može primeniti beta redukcija.

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \rightarrow$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k + 1)((\lambda m.m - 1)2) \to (\lambda k.k + 1)(2 - 1) \to$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k + 1)((\lambda m.m - 1)2) \rightarrow (\lambda k.k + 1)(2 - 1) \rightarrow (\lambda k.k + 1)1 \rightarrow$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \to (\lambda k.k+1)(2-1) \to (\lambda k.k+1)1 \to (1+1) \to 2$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \rightarrow (\lambda k.k+1)(2-1) \rightarrow (\lambda k.k+1)1 \rightarrow (1+1) \rightarrow 2$
2. $(\lambda k.k\ (k\ 4))(\lambda y.y-2) \rightarrow$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \rightarrow (\lambda k.k+1)(2-1) \rightarrow (\lambda k.k+1)1 \rightarrow (1+1) \rightarrow 2$
2. $(\lambda k.k\ (k\ 4))(\lambda y.y-2) \rightarrow ((\lambda y.y-2)\ ((\lambda y.y-2)\ 4)) \rightarrow$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \to (\lambda k.k+1)(2-1) \to (\lambda k.k+1)1 \to (1+1) \to 2$
2. $(\lambda k.k\ (k\ 4))(\lambda y.y-2) \to ((\lambda y.y-2)\ ((\lambda y.y-2)\ 4)) \to ((\lambda y.y-2)\ (4-2))) \to$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \rightarrow (\lambda k.k+1)(2-1) \rightarrow (\lambda k.k+1)1 \rightarrow (1+1) \rightarrow 2$
2. $(\lambda k.k\ (k\ 4))(\lambda y.y-2) \rightarrow ((\lambda y.y-2)\ ((\lambda y.y-2)\ 4)) \rightarrow ((\lambda y.y-2)\ (4-2))) \rightarrow (\lambda y.y-2)\ 2 \rightarrow 2-2 \rightarrow 0$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \to (\lambda k.k+1)(2-1) \to (\lambda k.k+1)1 \to (1+1) \to 2$
2. $(\lambda k.k\ (k\ 4))(\lambda y.y-2) \to ((\lambda y.y-2)\ ((\lambda y.y-2)\ 4)) \to ((\lambda y.y-2)\ (4-2))) \to (\lambda y.y-2)\ 2 \to 2-2 \to 0$
3. $((\lambda kmn.k-m+n)10)5 \to ((\lambda k.\lambda mn.k-m+n)10)5$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \to (\lambda k.k+1)(2-1) \to (\lambda k.k+1)1 \to (1+1) \to 2$
2. $(\lambda k.k\ (k\ 4))(\lambda y.y-2) \to ((\lambda y.y-2)\ ((\lambda y.y-2)\ 4)) \to ((\lambda y.y-2)\ (4-2))) \to (\lambda y.y-2)\ 2 \to 2-2 \to 0$
3. $((\lambda kmn.k-m+n)10)5 \to ((\lambda k.\lambda mn.k-m+n)10)5 \to ((\lambda k.(\lambda mn.k-m+n))10)5 \to$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \to (\lambda k.k+1)(2-1) \to (\lambda k.k+1)1 \to (1+1) \to 2$
2. $(\lambda k.k\ (k\ 4))(\lambda y.y-2) \to ((\lambda y.y-2)\ ((\lambda y.y-2)\ 4)) \to ((\lambda y.y-2)\ (4-2))) \to (\lambda y.y-2)\ 2 \to 2-2 \to 0$
3. $((\lambda kmn.k-m+n)10)5 \to ((\lambda k.\lambda mn.k-m+n)10)5 \to ((\lambda k.(\lambda mn.k-m+n))10)5 \to (\lambda mn.10-m+n)5 \to$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \rightarrow (\lambda k.k+1)(2-1) \rightarrow (\lambda k.k+1)1 \rightarrow (1+1) \rightarrow 2$
2. $(\lambda k.k\ (k\ 4))(\lambda y.y-2) \rightarrow ((\lambda y.y-2)\ ((\lambda y.y-2)\ 4)) \rightarrow ((\lambda y.y-2)\ (4-2))) \rightarrow (\lambda y.y-2)\ 2 \rightarrow 2-2 \rightarrow 0$
3. $((\lambda kmn.k-m+n)10)5 \rightarrow ((\lambda k.\lambda mn.k-m+n)10)5 \rightarrow ((\lambda k.(\lambda mn.k-m+n))10)5 \rightarrow (\lambda mn.10-m+n)5 \rightarrow (\lambda m.(\lambda n.10-m+n))5 \rightarrow$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k+1)((\lambda m.m-1)2) \to (\lambda k.k+1)(2-1) \to (\lambda k.k+1)1 \to (1+1) \to 2$
2. $(\lambda k.k\ (k\ 4))(\lambda y.y-2) \to ((\lambda y.y-2)\ ((\lambda y.y-2)\ 4)) \to ((\lambda y.y-2)\ (4-2))) \to (\lambda y.y-2)\ 2 \to 2-2 \to 0$
3. $((\lambda kmn.k-m+n)10)5 \to ((\lambda k.\lambda mn.k-m+n)10)5 \to ((\lambda k.(\lambda mn.k-m+n))10)5 \to (\lambda mn.10-m+n)5 \to (\lambda m.(\lambda n.10-m+n))5 \to (\lambda n.10-5+n) \to (\lambda n.5+n)$

Primeri

Izvesti normalni oblik primenom odgovarajućih redukcija na termove (prikazati postupak):

1. $(\lambda k.k \cdot k + 1)((\lambda m.m + 1)2)$
2. $(\lambda k.k\ 4)(\lambda y.y - 2)$
3. $((\lambda kmn.k \cdot m + n)2)3$

Normalni oblik

- Bilo bi dobro da je normalni oblik jedinstven i da ga mi možemo uvek pronaći.
- Ali:
  - Nemaju svi izrazi svoj normalni oblik. Na primer, $(\lambda x.x\ x)(\lambda x.x\ x)$
  - Za neke izraze mogu da postoje različite mogućnosti primene beta redukcije. Na primer,
    $(\lambda x.5 \cdot x)((\lambda x.x + 1)2) \rightarrow_\beta (\lambda x.5 \cdot x)(2 + 1)$
    $(\lambda x.5 \cdot x)((\lambda x.x + 1)2) \rightarrow_\beta 5 \cdot ((\lambda x.x + 1)2)$
    Postavlja se pitanje da li je važno kojim putem se krene?

Da li je važno kojim putem se krene?

- $(\lambda y.y\ a)\underline{((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow$

Da li je važno kojim putem se krene?

- $(\lambda y.y\ a)\underline{((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow (\lambda y.y\ a)(\lambda z.(\lambda u.u)\ z)$

Da li je važno kojim putem se krene?

- $(\lambda y.y\ a)\underline{((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow (\lambda y.y\ a)(\lambda z.(\lambda u.u)\ z)$
- $\underline{(\lambda y.y\ a)((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow$

Da li je važno kojim putem se krene?

- $(\lambda y.y\ a)\underline{((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow (\lambda y.y\ a)(\lambda z.(\lambda u.u)\ z)$
- $\underline{(\lambda y.y\ a)((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow ((\lambda x.x)(\lambda z.(\lambda u.u)\ z))\ a$

Da li je važno kojim putem se krene?

- $(\lambda y.y\ a)\underline{((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow (\lambda y.y\ a)(\lambda z.(\lambda u.u)\ z)$
- $\underline{(\lambda y.y\ a)((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow ((\lambda x.x)(\lambda z.(\lambda u.u)\ z))\ a$
- $(\lambda y.y\ a)((\lambda x.x)(\lambda z.\underline{(\lambda u.u)\ z})) \rightarrow$

Da li je važno kojim putem se krene?

- $(\lambda y.y\ a)\underline{((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow (\lambda y.y\ a)(\lambda z.(\lambda u.u)\ z)$
- $\underline{(\lambda y.y\ a)((\lambda x.x)(\lambda z.(\lambda u.u)\ z))} \rightarrow ((\lambda x.x)(\lambda z.(\lambda u.u)\ z))\ a$
- $(\lambda y.y\ a)((\lambda x.x)(\lambda z.\underline{(\lambda u.u)\ z})) \rightarrow (\lambda y.y\ a)((\lambda x.x)(\lambda z.z))$
- Da li ćemo u ova tri slučaja daljom primenom beta redukcija stići do istog izraza?

Svojstvo konfluentnosti

- Church-Rosser teorema: ako se lambda izraz može svesti na dva različita lambda izraza M i N, onda postoji treći izraz Z do kojeg se može doći i iz M i iz N.
- Posledica teoreme je da svaki lambda izraz ima najviše jedan normalni oblik (dakle, ako postoji, on je jedinstven)
- To znači da nije bitno kojim putem se dolazi do normalnog oblika, ukoliko do normalnog oblika dođemo, znamo da smo došli do jedinstvenog normalnog oblika.
- Kako da dođemo do normalnog oblika?

Poredak izvođenja redukcija

- Aplikativni poredak
- Na primer, $(\lambda x.5 \cdot x)(2+1) \rightarrow_\beta (\lambda x.5 \cdot x)3 \rightarrow_\beta 5 \cdot 3 \rightarrow 15$
- Ovo odogovara pozivu po vrednosti (call-by-value) — izračunavamo vrednost argumenta i tek kada ga izračunamo šaljemo ga u funkciju i funkcija dočeka u svom telu izračunati argument

Poredak izvođenja redukcija

- Normalni poredak: beta redukcijom uvek redukovati najlevlji izraz.
- Na primer, $(\lambda x.5 \cdot x)(2+1) \rightarrow_{\beta} 5 \cdot (2+1) \rightarrow 5 \cdot 3 \rightarrow 15$
- Ovo odgovara evaluaciji po imenu (call-by-name) ili evaluaciji po potrebi (call-by-need)

Poredak izvođenja redukcija

- Normalni poredak: beta redukcijom uvek redukovati najlevlji izraz.
- Na primer, $(\lambda x.5 \cdot x)(2+1) \rightarrow_{\beta} 5 \cdot (2+1) \rightarrow 5 \cdot 3 \rightarrow 15$
- Ovo odgovara evaluaciji po imenu (call-by-name) ili evaluaciji po potrebi (call-by-need)

Teorema standardizacije

Ako je Z normalni oblik izraza E, onda postoji niz redukcija u normalnom poretku koji vodi od E do Z.

Lenja evaluacija

- Normalnim poretkom redukcija ostvaruje se lenja evaluacija — izrazi se evaluiraju samo ukoliko su potrebni.
- Normalnim poretkom, tj lenjom evaluacijom se izbegavaju nepotrebna izračunavanja.
- Na primer:
  AP: $(\lambda x.1)(12345 \cdot 54321) \rightarrow_\beta (\lambda x.1)670592745 \rightarrow_\beta 1$
  NP: $(\lambda x.1)(12345 \cdot 54321) \rightarrow_\beta 1$
- Dakle, evaluiramo izraz tek onda kada nam njegova vrednost treba

Lenja evaluacija

- Normalni poredak, tj lenja evaluacija, nam garantuje završetak izračunavanja uvek kada je to moguće
- Na primer
  AP: $(\lambda x.1)((\lambda x.x\ x)(\lambda x.x\ x)) \rightarrow_{\beta} \ldots$ (ne završava)
  NP: $(\lambda x.1)((\lambda x.x\ x)(\lambda x.x\ x)) \rightarrow_{\beta} 1$
- Normalni poredak, tj lenja evaluacija, odgovara nestriktnoj semantici
- Normalni poredak, tj lenja evaluacija, omogućava korišćenje beskonačnih struktura

```
sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Efikasnost izračunavanja kod lenje evaluacije

- Postoje tehnike koje primenjuju kompajleri, a koje obezbeđuju da se izračunavanja ne ponavljaju, ovo je važno sa stanovišta efikasnosti izračunavanja.
- Na primer, za $(\lambda x.x + x)(12345 \cdot 54321) \rightarrow_\beta (12345 \cdot 54321) + (12345 \cdot 54321)$ ne bi bilo dobro dva puta nezavisno računati proizvod $(12345 \cdot 54321)$ već je potrebno to samo jednom uraditi, i za to postoje tehnike redukcije grafova.

Pregled

1. Uvod u funkcionalno programiranje
2. Programski jezik Haskell
3. Svojstva funkcionalnih jezika
4. Teorijske osnove — lambda račun
5. Literatura i pitanja
   - Literatura
   - Pitanja

Literatura

- https://www.haskell.org/
- https://www.haskell.org/documentation
- Real world Haskell
  http://book.realworldhaskell.org/
- http://learnxinyminutes.com/docs/haskell/
- Lambda račun http://poincare.matf.bg.ac.rs/~nenad/fp/lambda%20racun%20i%20kombinatori.ps

Pitanja

- Na koji način je John Backus uticao na razvoj funkcionalnih jezika?
- Koji su najpoznatiji funkcionalni programski jezici?
- Koji je domen upotrebe funkcionalnih programskih jezika?
- Koje su osnovne karakteristike funkcionalnih programskih jezika?

Pitanja

- Šta je svojstvo transparentnosti referenci i na koji način ovo svojstvo utiče na redosled naredbi u funkciji?
- Koje su osobine programa u kojima se poštuje pravilo transparentnosti referenci?
- Da li je moguće u potpunosti zadržati svojstvo transparentnost referenci?
- Koji je odnos transparentnosti referenci sa bočnim efektima?
- Da li je moguće obezbediti promenu stanja programa i istovremeno zadržati svojstvo transparentnosti referenci?

Pitanja

- Šta su funkcionalni jezici? Šta su čisto funkcionalni jezici?
- Navesti primere čisto funkcionalnih jezika?
- Koje su osnovne aktivnosti u okviru funkcionalnog programiranja?
- Kako izgleda program napisan u funkcionalnom programskom jeziku?
- Šta je potrebno da obezbedi funkcionalni programski jezik za uspešno proramiranje?

Pitanja

- Šta je striktna/nestriktna semantika?
- Kakvu semantiku ima jezik Haskell?
- Kakvu semantiku ima jezik Lisp?
- Koje su prednosti funkcionalnog programiranja?
- Koje su mane funkcionalnog programiranja?

Pitanja

- Šta uključuje definisanje funkcije?
- Šta su funkcije višeg reda? Navesti primere.
- Da li matematičke funkcije imaju propratne efekte?

Pitanja

- Koji je formalni okvir funkcionalnog programiranja?
- Koji se jezik smatra prvim funkcionalnim jezikom?
- Koja je ekspresivnost lambda računa?
- Koji su sve sinonimi za lambda izraz?
- Navesti definiciju lambda terma.

Pitanja

- Da li čist lambda račun uključuje konstante u definiciji?
- Navesti primer jednog lambda izraza, objasniti njegovo značenje i primeniti dati izraz na neku konkretnu vrednost.
- Koja je asocijativnost primene a koja apstrakcije?
- Navesti ekvivalentan izraz sa zagradama za izraz ...
- Koje su slobodne a koje vezane promenljive u izrazu ...

Pitanja

- Navesti definiciju slobodne promenljive? Koje promenljive su vezane?
- Koja je uloga pojma alfa ekvivalentnosti?
- Šta su redukcije?
- Šta je delta redukcija? Navesti primer.
- Šta je alfa redukcija? Navesti primer.

Pitanja

- Kada se koristi alfa redukcija?
- Šta je beta redukcija? Navesti primer.
- Definisati supstituciju.
- Navesti primer lambda izraza koji definiše funkciju višeg reda koja prima funkciju kao argument.
- Navesti primer lambda izraza koji definiše funkciju višeg reda koja ima funkciju kao povratnu vrednost.

Pitanja

- Čemu služi Karijev postupak?
- Kako se definišu funkcije sa više argumenata?
- Šta je normalni oblik funkcije?
- Da li svi izrazi imaju svoj normalni oblik?
- Navesti svojstvo konfluentnosti.

Pitanja

- Da li izraz može imati više normalnih obilika?
- Koja je razlika između aplikativnog i normalnog poretka?
- Šta govori teorema standardizacije?
- Šta se dobija lenjom evaluacijom?
- Koje su osnovne karakteristike Haskela?
- Šta izračunava naredni Haskel program ...