Funkcijsko programiranje

λ-račun

* prefiksna forma (npr. + x 3, \* x x)
* funkcije (λx. \*3x)
  + λ - keyword
  + x - parametar
  + . - razdvaja parametre od tijela

(λx. \*3x) 4 = \*3 4 = 12

λx. \*3x4 = \* 3 x 4 = 3 \* x \* 4

beta-redukcija

* jedino pravilo za računanje u lambda-računu
* zamjena formalnog parametra stvarnim parametrom
* (λx.\*3x)4 -> \* 3 4

normalna forma izraza – stanje u kojem daljnja redukcija izraza nije moguća

Church-Rosserov teorem - ukoliko se izraz M može reducirati u N i P onda postoji izraz Q u koji se mogu reducirati N i P

korolar - svaki λ-izraz ima najviše jednu normalnu formu

dokaz - Izraz M se reducira u dvije normalne forme N i P. Po Church-Rosserovom teoremu postoji Q u koji se N i P reduciraju, što nije moguće ako su N i P normalne forme dakle N=P=Q

λ-račun ne razlikuje jednostavne entitete (npr. brojeve) i složene entitete (npr. funkcije)

Q = (λx.\*xx)

P = λx.Q(Q(Qx))

Funkcijsko programiranje

* prednosti
  + varijablama se ne može promijeniti vrijednost
  + lakše se dokazuje formalna točnost programa
  + kraći kod
  + viši nivo razmišljanja
* nedostaci
  + teže napisati program
  + I/O model težak za razumijevanje
  + sporo

Haskell

* čisti funkcijski jezik
* jezik statičkog implicitnog tipa - ne treba deklarirat, ali se provjeravaju tipovi
* ne može se mijenjat vrijednost varijabli
* lijena evaluacija (ništa se ne radi dok nije nužno)
* sve je izraz
* funkcije ravnopravne varijablama
* polimorfizam

GHCI

* kompilacija i izvođenje
* izrazi se unose i odma izvode
* novi red = kraj unosa

Prelude - standard library, uvijek inkludan

typeclass - npr. Num sadrži Double, Float, Int, Integer - tip argumenta određuje koja se klasa koristi

implicitno određivanje tipova - izvor grešaka

Liste

* skupovi bilo kojih vrijednosti
* svi elementi su istog tipa
* tip se određuje automatski
* automatska alokacija memorije
* oslobađanje memorije -> garbage collector
* 1:[2,3] - dodaje 1 u listu ispred 2 i 3
* [1,2]++[3,4] - spaja dvije liste
* head - vraća prvi element
* tail - vraća sve elemente izuzev prvi

Funkcije

* evaluira i vraća cijelo tijelo funkcije
* Primjer

fac n = if n == 0 then 1 else n \* fac (n-1)

Međusobno rekurzivne funkcije

* pozivaju jedna drugu
* šalju si argumente
* Primjer:

take [] = []

take(x:xs) = x : skip xs

skip[] = []

skip(\_:ys) = y : take ys

take vraća prvi element liste(head) i poziva za ostatak(tail) skip

skip ne vraća ništa nego poziva take za ostatak liste (podvlaka \_ označava wild-card te se obično stavlja na mjesta za koja nas nije birga)

(dakle preskačemo svaki drugi element liste)

Lambda apstrakcije

* lambda funkcije
* Primjer: (\x -> x^2) - vraća kvadrat

Lijena evaluacija - izrazi se računaju tek kada se koriste

Curryjev postupak - argumenti se funkciji predaju zasebno

Primjer:

let fja x y = ...nešto

let fja2 = fja 3.14

fja2 2.71

dakle fja2 treba samo 1 argument jer joj je prvi zadan

Funkcije višeg reda - primaju druge funkcije kao parametar ili vraćaju druge funkcije

Eksplicitna definicija tipova - :l A.hs (za load A.hs biblioteke) i x::Int za definiranje

Parametarsko višeobličje (polimorfizam) - automatsko određivanje tipa

Uzorci (patterns) - [] prazna lista, (x:xs) rekurzivna definicija

neke naredbe

succ - nasljednik - succ 5 -> 6

truncate - odreži - truncate 3.14 -> 3

round - zaokruži - round 2.71 -> 3

sqrt - kvadratni korijen

not - negacija - not (3.14 > 2.71) -> False

gcd - najveći zajednički djelitelj - gcd 12 8 -> 4

map (+2) [1..5]

[3,4,5,6,7]

filter (>3) [1..5]

[4,5]

C# 3.0

* nastao 2007.
* uključen u .NET framework 3.5
* strongly typed
* spaja deklarativnu i imperativnu paradigmu
  + lambda izrazi
  + lijena evaluacija

Parametarski polimorfizam

* generics
* kod je apstraktniji
* prevodilac određuje tip

T Min<T>(T a, T b) where T : IComparable<T> {

if (a.CompareTo(b) < 0) return a; else return b;

}

T označava neki tip (ovim dijelo where T: IComparable<T> označava da je taj tip moguće uspoređivat pa se može koristit metoda CompareTo() )

Delegat - to je poput pokazivača na klasu, točnije na jednu metodu iz nje, najbolje objašnjeno prek primjera

delegate void MojDelegat();

public class MojaKlasa {

// dodatni podaci, svojstva, metode

public void MojaMetoda() {

Console.WriteLine ("Ovo je moja metoda");

}

}

public class DrugaKlasa {

public void Glavna() {

MojaKlasa klasa = new MojaKlasa();

Ispis(klasa.MojaMetoda);

}

void Ispis(MojDelegat delegat) {

delegat();

}

}

Anonimni delegat (C# 2.0)

public class DrugaKlasa {

public void Glavna() {

MojaKlasa klasa = new MojaKlasa();

Ispis(delegate {

Console.WriteLine("test");

});

}

void Ispis(MojDelegat delegat) {

delegat();

}

}

Automatsko određivanje tipa

* keyword var
* razlika od korištenja object: var dodjeljuje tip, ali ne dolazi do boxinga (dakle kada se ponovo koristi nije potrebno naglasiti unboxati)
* var a = 5; int b = a;
* object a = 5; int b = (int) a;
* smije se koristiti samo za lokalne deklaracije !!!

Lambda izrazi - oblik: (a,b) => a+b

* vrste:
  + predikat (određuje pripadnost grupi): (a) => a > 3.14
  + projekcija (vraća tip drugačiji od ulaznog): (a) => a.Length
* Predefinirani delegati za lambda izraze oblika:

Public delegate T Func<A0, A1, ..., T>( A0 arg0, A1 arg1, ...);

Metode proširenja (extension methods)

* pozivaju se nad varijablama - npr. int x = 4; x.MyExtension();
* moraju biti public i static
* moraju biti u static klasi
* ispred oznake tipa prvog parametra(tipa koji se proširuje) moraju imat ključnu riječ this
* unutar njih je moguće koristiti samo metode javnog sučelja tipa kojeg proširujemo

Izrazi za inicijalizaciju objekata

* alternativa konstruktorima
* inicijalizacija svojstva ili varijabli

Primjer:

public class MojaKlasa {

int prva;

int druga;

public int Druga {

get { return druga; }

set { druga = value; }

}

public MojaKlasa(int prva) {

this.prva = prva;

}

MojaKlasa klasa = new MojaKlasa(4) { Druga = 5 };

Anonimni tipovi

* klase koje ne definiramo eksplicitno
* nastaju automatski u trenu prevođenja
* ne može ih se eksplicitno koristiti
* možemo normalno koristiti njihove objekte
* dva su anonimna tipa jednaka ako im je jednak poredak, naziv i tip članskih varijabli
* prevodilac ne stvara više istih anonimnih tipova neko iskorištava postojeće

LINQ

* Language INtegrated Query
* pisanje upita nad listama

var ljudovi = new[] {

new {ime = "jovica", prezime = "jovic", age = 35},

new {ime = "katica", prezime = "jovic", age = 33},

new {ime = "jovica jr.", prezime = "jovic", age = 13},

new {ime = "danko", prezime = "bananko", age = 44}

};

var rez = from ljud in ljudovi

where ljud.prezime == "jovic"

orderby ljud.age

select new {ljud.ime, ljud.prezime, ljud.age};

ili

var rez = ljudovi

.Where(ljud => ljud.prezime == "jovic")

.OrderBy(ljud => ljud.age)

.Select(ljud => new {ljud.ime, ljud.prezime, ljud.age});

još neke naredbe:

group ljud by ljud.prezime

from ljud in ljudovi

join covjek in covjeci

on ljud.prezime equals covjek.prezime

Paralelizam

* poslovi se izvode paralelno dok ne dođe do kritičnog odsječka pa opet nakon njega
* kreiranje nove instance procesa: UNIX - fork(), Windows - CreateProcess()
* kreiranje novih dretvi: using System.Threading

Thread dretva = new Thread(FunkcijaDretve);

dretva.Start(i);

static void FunkcijaDretve(object objekt) { }

PFX library

* System.Threading
* PLINQ
  + nadogradnja LINQ
  + radi u paralelnom načinu
* TPL - task parallel library
  + library za podršku paralelnom izvođenju programa
  + Statička klasa Parallel
    - Parallel.For
    - Parallel.ForEach
    - Parallel.Invoke - poziva delegate u paralelnom okruženju
  + Task i Future razredi
    - Task - najmanja cjelina posla
    - svaki posao u svojoj dretvi
    - task manager - sadrži sve aktivne poslove te brine za njihovo stvaranje i sinkronizaciju
    - Future - izveden iz Task, vraća vrijednost
* povećanje performansi na višejezgrenim i višeprocesorskim sustavima
* programer ne mora paziti na podjelu, sinkronizaciju i broj dertvi
* nedostaci
  + ukoliko se prečesto zaključava objekte korištenje paralelnog načina može biti i mnogo sporije nego obično
  + kod kompleksnih poslova(koriste dijeljeni dio memorije ili su poslovi djelomično nezavisni) mora se ručno sinkronizirat
  + teže rukovanje iznimkama(sve se nalaze u ExceptionCollection)
  + problem očuvanja poretka
  + za male skupove podataka se ne isplati jer je vrijeme pripreme za aktivnost veće od vremena obrade podataka