FORTRAN, LISP, COBOL, ALGOL , C, Pascal, Prolog, C++, Haskell, Python, Ruby, JAVA, PHP, Lua, JavaScript, C#, Scala, F#, Elixir, Perl

Paradigma – uzorak, obrazac, sablon; vrsta objekata sa zajednickim karakteristikama

Najopstija podela je na proceduralnu i deklarativnu paradigmu

Proceduralna paradigma - osnovni zadatak programera da opise nacin (proceduru) kojim se dolazi do resenja problema

Deklarativna paradigma - osnovni zadatak programera je da precizno opise problem, dok se mehanizam programskog jezika bavi pronalazenjem resenja problema

Osnovne programske paradigme  
-Imeprativna paradigma  
-Objektno-orijentisana paradigma   
-Funkcionalna paradigma   
-Logicka paradigma

Programski jezik je sredstvo koje koristi covek da izrazi proces pomocu kojeg racunar resava nekakav problem

Dodatne paradigme:  
-Komponentna paradigma   
-Paradigma upitnih jezika  
-Genericka paradigma  
-Vizuelna paradigma  
-Konkurentna paradigma  
-Reaktivna paradigma  
-Skript paradigma  
-Paradigma programiranja ogranicenja

**Imperativna (proceduralna) paradigma**

Osnovni pojam imperativnih jezika je naredba

Naredbe se grupi su u procedure i izvrsavaju se sekvencijalno ukoliko se eksplicitno u programu ne promeni redosled izvrsavanja naredbi

Upravljacke strukture su naredbe grananja, naredbe iteracije, i naredbe skoka (goto)

Oznake promenljivih su oznake memorijskih lokacija pa se u naredbama cesto mesaju oznake lokacija i vrednosti - to izaziva bocne efekte.

C, Pascal, Basic, Fortran, PL, ALGOL

**Objektno-orijentisana paradigma**

Simulacija (modeliranje) spoljasnjeg sveta pomocu objekata

Podaci i procedure (funkcije) se ucauravaju (enkapsuliraju) u objekte

Koristi se skrivanje podataka da bi se zastitila unutrasnja svojstva objekata

Objekti su grupisani po klasama

Klase su najcesce hijerarhijski organizovane i povezane mehanizmom nasledivanja

Simula 67, SmallTalk, C++, Eiffel, Java, C#

**Funkcionalna paradigma**

Izracunavanja su evaluacije matematickih funkcija

Zasnovana je na pojmu matematicke funkcije i ima formalnu strogo defnisanu matematicku osnovu u lambda racunu

Bez bocnih efekata – izlazna vrednost funkcije zavisi samo od ulaznih vrednosti argumenata funkcije

Lisp, Scheme, Haskell, ML, Scala, OCaml

**Logicka paradigma**

Deklarativna paradigma

Izvrsavanje programa zasniva se na sistematskom pretrazivanju skupa cinjenica uz koriscenje odredenih pravila zakljucivanja

Zasnovana na matematickoj logici, tj. na predikatskom racunu 1. Reda

Zasnovana na automatskom dokazivanju teorema (metod rezolucije)

Prolog, ASP, Datalog, CLP, ILOG, Solver, ParLog, LIFE

**Skript jezici**

Skript je spisak (lista) komandi koje mogu biti izvrsene u zadatom okruzenju bez interakcije sa korisnikom

Skript jezici mogu imati specican domen primene:  
-U prvobitnom obliku pojavljuju se kao komandni jezici operativnih sistema (npr Bash)  
-Velika primena u programiranju web aplikacija (JavaScript, TypeScript...)  
- Cesto se koriste i za povezivanje komponenti unutar neke aplikacije

Skript jezici mogu biti i jezici opste namene (npr Python)

Skript jezici imaju razne specicne osobine  
-Obicno se ne kompiliraju vec interpreteraju  
-Obicno dinamicki odreduju tipove

Unix Shell (sh), JavaScript, PHP, Perl, Python, XSLT, VBScript, Lua, Ruby

**Paradigma programiranja ogranicenja**

U okviru paradigme programiranja ogranicenja zadaju se relacije izmedu promenljivih u formi nekakvih ogranicenja

Ogranicenja mogu biti raznih vrsta (logicka, linearna...)

Deklarativna paradigma: ova ogranicenja ne zadaju sekvencu koraka koji treba da se izvrse vec osobine resenja koje treba da se pronade

Postoje biblioteke za podrsku ovoj vrsti programiranja u okviru imperativnih/objektno orijentisanih/skript programskih jezika, npr. za jezike C, JAVA, C++, Python

**Komponentna paradigma**

Softver se sklapa od vecih gotovih komponenti

Komponenta je jedinica funkcionalnosti sa ½ugovoreniminterfejsom

Interfejs definise nacin na koji se komunicira sa komponentom, i on je u potpunosti odvojen od implementacije

Priblizavanje deklarativnom stilu programiranja

Cilj je da se uprosti proces programiranja i da se jednom kreirane komponente mnogo puta koriste

Komponente se medusobno povezuju da bi se kreirao kompleksan softver

Nacin povezivanja komponenti treba da bude jednostavan – prevlacenjem

Kod koji se komponentnim programiranjem generise moze da bude u razlicitim programskim jezicima, npr JAVA, C++, C#

**Upitni jezici**

Upitni jezici mogu biti vezani za baze podataka ili za pronalazenje informacija (engl. information retrieval)

Deklarativna paradigma

Upitni jezici baza podataka: Na osnovu struktuiranih cinjenica zadatih u okviru struktuiranih baza podataka daju konkretne odgovore koji zadovoljavaju nekakve trazene uslove;

SQL - RBP, SPARQL – podaci u RDF formatu, XQuery – pretrazivanje XML podataka

Upitni jezici za pronalazenje informacija su upitni jezici koji pronalaze dokumenta koji sadrze informacije relevantne za oblast istrazivanja

CQL (engl. Contextual Query Language) jezik za iskazivanje upita za pronalazenje informacija  
-Formalni jezik za predstavljanje upita za izvlacenje informacija u sistemima kao sto su web indeksi, bibliografski katalozi, informacije o muzejskim zbirkama  
-Dizajn jezika podrazumeva da su upiti takvi da se mogu lako citati i pisati, i da je i jezik intuitivan pri cemu zadrzava ekspresivnost kompleksnijih upitnih jezika

**Genericka paradigma**

Genericko programiranje je stil programiranja u kojem se algoritmi pisu sa apstrahovanim tipovima koji se obezbeduju kao parametri i time se izbegava dupliranje koda

Na primer, algoritmi trazenja minimuma, maksimuma, sortiranja, pretrage

**Vizuelna paradigma**

Vrsi modelovanje spoljasnjeg sveta (usko povezana sa objektno-orijentisanom paradigmom)

Koriste se graficki elementi (dijagrami) za opis akcija, svojstva i povezanosti sa raznim resursima

UML

Primeri strukturnih informacija:  
-dijagram klasa  
-dijagram objekata  
-dijagram komponenti

Primeri informacija o ponasanju:  
-dijagram komunikacije  
-dijagram aktivnosti  
-dijagram slucajeva upotrebe

**Konkurentna paradigma**

Konkurentnu paradigmu karakterise vise procesa koji se izvrsavaju u istom vremenskom periodu, a koji imaju isti cilj

Konkurentnost u uzem smislu - jedan processor, jedna memorija

Paralelno programiranje - vise procesora, jedna memorija

Distribuirano programiranje - vise procesora, vise memorija

Konkurentnost u uzem smislu karakterise preklapajuce izvrsavanje vise procesa koji koriste isti procesor i koji komuniciraju preko zajednicke memorije

Ukoliko postoji vise procesora sa pristupom jedinstvenoj memoriji, onda je u pitanju paralelno programiranje

Ukoliko postoji vise procesora od kojih svaki ima svoju memoriju, onda je u pitanju distribuirano programiranje

Procesi medusobno salju poruke da bi razmenili informacije

Distribuirano izracunavanje cine grupe umrezenih racunara koje imaju isti cilj za posao koji izvrsavaju.

Moze se shvatiti kao vrsta paralelnog izracunavanja ali sa drugacijom medusobnom komunikacijom koja namece nove izazove

Pisanje konkurentnih (konkurentnih u uzem smislu, paralelnih, distribuiranih) programa je znacajno teze od pisanja sekvencijalnih programa

Konkurentno programiranje namece nove probleme, po pitanju sinhronizacije procesa i pristupa zajednickim podacima

Ada, Modula, ML, Java, C, C++, Haskell

**Reaktivna paradigma**

Reaguje na dogadaje - vodjena dogadjajima (engl. event-driven)  
Reaguje na rast upotrebe - skalabilna (engl. scalable)  
Reaguje na padove - mogucnost brzog oporavka od pada (engl. resilient)  
-Pad moze biti softverski izuzetak, hardverski pad, pad veze  
-Da bi se to ostvarilo, mora da bude sastavni deo arhitekture sistema (medusobno slabo zavisne komponente)  
Reaguje na zahteve korisnika - brza reakcija na korisnicke zahteve (engl. responsive), cak i kada je opterecenje sistema veliko i postoje padovi  
-Grade se nad event-driven, scalable i resilient arhitekturama ali mora da se vodi racuna o algoritmima, dizajnu sistema i puno drugih detalja

Reaktivne apstrakcije:  
-Apstrakcija dogadaja = futures  
-Apstrakcija tokova dogadaja = observables  
-Arhitektura slanja poruka = njeni cvorovi su actors  
-Pracenje padova = koncept supervisors  
Skaliranje koriscenjem distribuiranih sistema = distributed actors

Reaktivno programiranje nije vezano za programski jezik, vec je to stil programiranja koji se moze ostvariti u najrazlicitijim programskim jezicima

**Jezici za obelezavanje teksta/podataka i programske paradigme**

HTML, CSS, XML, SGML, JSON

**NISU** programski jezici jer se ne uklapaju u deniciju programskih jezika

Naime, po definiciji, programskim jezikom se zadaje nakakvo izracunavanje odnosno definisu se programi koje racunar moze da izvrsi a ovim jezicima definise se struktura teksta/podataka, ne pisu se programi i nema nikakvog izvrsavanja

HTML definise strukturu veb stranice i odreduje naslove, paragrafe i slicno, dakle koristi se za odredjivanje strukturnih osobina veb stranice, a ne njene funkcionalnosti

HTML ne sadrzi kontrolu toka, evaluaciju izraza, funkcije, promenljive, pojam dogadaja, ne moze da modifikuje podatke niti da ima ulaz/izlaz

Paralelno sa razvojem jezika za obelezavanje (posebno XML), razvijeni su specijalizovani programski jezici za razne obrade koje se odnose na jezike za obelezavanje: XSLT, XQuery, XLS - ovi jezici se mogu pridruziti raznim paradigmama (najcesce domenski specificni jezici i skript jezici)

**PROGRAMIRANJE OGRANICENJA**

Zadatak je naci dopustivo resenje, odnosno odrediti vrednosti promenljivih koje zadovoljavaju sva postavljena ogranicenja, ili dokazati da takvo resenje ne postoji

Deklarativna programska paradigma

x < y u imperativnoj paradigmi se evaluira u tacno ili netacno, dok u paradigmi ogranicenja zadaje relaciju izmedu objekata x i y koja mora da vazi

Programiranje ogranicenja je savremen pristup resavanju teskih kombinatornih problema, ima primene pre svega u operacionim istrazivanjima, tj. u resavanju kombinatornih i optimizacionih problema

Programiranje ogranicenja je nastalo u okviru logickog programiranja

Programiranje ogranicenja nad konacnim domenom sastoji se od tri dela  
-Generisanje promenljivih i njihovih domena   
-Generisanje ogranicenja nad promenljivama   
-Obelezavanje (labeling) - instanciranje promenljivih

Kriptaritmetike su matematicke igre u kojima se resavaju jednacine kod kojih su cifre brojeva zamenjene odredenim slovima

B-Prolog implementira ISO standard Prologa ali i razna prosirenja

Programiranje ogranicenja u B-Prologu  
Generisanje promenljivih i njihovih domena  
-Definisanje domena: Vars in D, ili Vars :: D  
-D se definise kao Pocetak..Korak..Kraj pa tako 1..5..20 definise domen 1,6,11,16  
-Korak nije obavezan, podrazumeva se da je jedan, 1..10 svi brojevi od 1 do 10  
-Postoje predikati za opsta ogranicenja, a mogu se zadavati numericka ogranicenja  
-Opsta, npr alldifferent, ili alldistinct  
-Numericka pocinju sa #, npr #<=...   
-Labeling se odnosi na instanciranje promenljivih i na njihovo prikazivanje

**FUNKCIONALNO PROGRAMIRANJE**

Izvrsavanje programa moze se svesti i na evaluaciju izraza a u zavisnosti od izraza, imamo logicku paradigmu (izrazi su relacije) i funkcionalnu (izrazi su funkcije)

Pomenute paradigme se temelje na razlicitim teorijskim modelima  
-Formalizam za imperativne jezike je Tjuringova i URM masina  
-Formalizam za logicke jezike je Logika prvog reda  
-Formalizam za funkcionalne jezike je Lambda racun

Funkcionalni jezici su mnogo blizi svom teorijskom modelu nego imperativni jezici pa je poznavanje lambda racuna vazno i ono omogucava bolje funkcionalno programiranje

Svi programi koji se mogu napisati imperativnim stilom, mogu se napisati i funkcionalnim stilom

Lambda racun naglasava pravila za transformaciju izraza i ne zamara se arhitekturom masine koja to moze da ostvari

Osnovni cilj funkcionalne paradigme je da oponasa matematicke funkcije - rezultat toga je pristup programiranju koji je u osnovi drugaciji od imperativnog programiranja

Funkcionalno programiranje je stil koji se zasniva na izracunavanju izraza kombinovanjem funkcija  
-Definisanje funkcije (pridruzivanje imenu funkcije vrednosti izraza pri cemu izraz moze sadrzati pozive drugih funkcija)  
-Primena funkcije (poziv funkcije sa zadatim argumentima)  
-Kompozicija funkcija (navodjenje niza poziva funkcija) - kreiranje programa  
Program u funkcionalnom programiranju je niz definicija i poziva funkcija a njegovo izvrsavanje je evaluacija funkcija

max3(x,y,z) = max(max(x,y), z) max3() je kompozicija max() funkcija

Funkcija je ravnopravna sa ostalim tipovima podataka, moze biti povratna vrednost ili parametar druge funkcije

Najistaknutiji predstavnik funkcionalne paradigme bio je programski jezik Lisp (LISt Processing - LISP)

Funkcionalni jezici su laksi za razumevanje, najvise zbog toga sto je vrednost izraza nezavisna od konteksta u kojem se izraz nalazi

Osnovna karakteristika cistih funkcionalnih programskih jezika je **transparentnost referenci** sto kao posledicu ima nepostojanje propratnih (bocnih) efekta

Jezici: Scheme, ML, Miranda, Erlang, SML, Haskell, OCaml, F#, Scala, Elixir

**HASKELL**

Cist funkcionalni jezik, lenja evaluacija (izbegavaju se nepotrebna izracunavanja)

Haskel ima mocni sistem tipova - tipovi se ne moraju uvek navoditi, tj postoji automatsko zakljucivanje tipova  
Staticki tipiziran jezik - tipovi se odreduju u fazi kompilacije  
Strogo tipiziran jezik - svi se tipovi moraju poklapati, nema implicitnih konverzija  
Podrska za paralelno i distirbuirano programiranje

Podrzava parametarski polimorfizam (viseoblicje) i preopterecivanje sto omogucava sazeto i genericko programiranje  
Podrzava kompaktan i ekspresivan nacin definisanja listi kao osnovnih struktura funkcionalnog programiranja

Funkcije viseg reda omogucavaju visok nivo apstrakcije i koriscenja funkcijskih oblikovnih obrazaca (uocavanje obrazaca izracunavanja koje se cesto sprovode i njihovo izdvajanje u funkcije viseg reda)  
Haskell ima podrsku za monadicko programiranje koje omogucava da se propratni efekti izvedu bez narusavanja transparentnosti reference  
Razradjena biblioteka standardnih funkcija (Standard Library) i dodatnih modula (Hackage)

**SVOJSTVA FUNKCIONALNIH JEZIKA**

Funkcionalni jezici imaju osnovne tipove podataka (celobrojna vrednosti, realne vrednosti, logicke vrednosti, stringovi)  
**Osnovna struktura podataka koja se javlja u svim funkcionalnim jezicima je lista**  
Funkcionalni jezici podrzavaju i torke koje mogu da sadrze elemente razlicitih tipova  
Cesto: torke zauzimaju kontinualni prostor u memoriji (slicno kao nizovi) dok su liste impelementirane preko povezanih listi  
Izbor odgovarajuce strukture zavisi od problema (da li je broj elemenata fiksiran)

Osnovni tipovi u Haskelu: Bool, Char, String, Int, Integer, Float

Torka u Haskelu:  
-Polinom drugog stepena  
p2 :: (Float, Float, Float)  
p2 = (1, 2, 3)

Lista u Haskelu:  
-Polinom proizvoljnog stepena  
pn :: [Float]  
pn = [1, 2, 3]  
Funkcije za rad sa listama: head, length, take, sum, !!  
Prelude> [0, 2..] !! 50 ce dati 100 – lista jeste beskonacna al ce haskel da izracuna do pedesetog elem. zbog lenjog izracunavanja

First class citizen = u okviru programskog jezika za neki gradivni element se kaze da je gradjanin prvog reda ako u okviru jezika ne postoje restrikcije po pitanju njegovog kreiranja i koriscenja

Gradjani prvog reda mogu da se cuvaju u promenljivama, da se prosleduju funkcijama, da se kreiraju u okviru funkcija i da se vrate kao povratna vrednost funkcija

U dinamicki tipiziranim jezicima (tj gde se tipovi odreduju u fazi izvrsavanja programa), gradani prvog reda imaju tip koji se proverava u fazi izvrsavanja

U funkcionalnom programiranju funkcije su gradani prvog reda

Funkcije viseg reda (funkcijske forme) imaju jednu ili vise funkcija kao parametre ili imaju funkciju kao rezultat, ili oba  
Primer: kompozicija funkcija - ima dve funkcije kao parametre i rezultat koji je takode funkcija  
Na primer, f(x) = x + 5 i g(x) = 2 · x, kompozicija funkcija f i g je h = f ◦ g = f(g(x)) = (2 · x) + 5

Primer: α funkcija (apply to all ili map)

Prelude> map (+1) [1,5,3,1,6]  
[2,6,4,2,7]

Primer: φ funkcija (filter)

Prelude> filter (>3) [1,5,3,2,1,6,4,3,2,1]  
[5,6,4]

Primer: ρ funkcija (reduce)

Prelude> foldr (-) 0 [1,2,3]  
2  
- 1-(2-(3-0))???

Nepostojanje stanja  
Funkcionalni jezici nemaju implicitno stanje, cime je razumevanje efekta rada funkcije znacajno olaksano

Izvodenje 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 moguce pa se ponavljanje ostvaruje kroz rekurziju

**Transparentnost referenci**: vrednost izraza je svuda jedinstveno odredena: ako se na dva mesta referencira na isti izraz, onda je vrednost ta dva izraza ista

Svojstvo transparentnosti referenci govori da redosled naredbi nije bitan

U funkcionalnom programiranju neobavezan je eksplicitni redosled navodenja funkcija jer ih mozemo kombinovati na razne nacine, bitno je da se definise izraz koji predstavlja resenje problema

To naravno nije slucaj kod imperativnih jezika gde je bitan redosled izvodenja operacija, x se nakon inicijalizacije moze promeniti, pa stoga imperativni jezici nemaju transparentnost referenci. (Naredba dodele ima propratni efekat, a ona je u osnovi imperativnog programiranja)

Propratni efekat je svaka ona promena implicitnog stanja koja narusava transparentnost referenci programa

Programi sa transparentnim referencama su formalno koncizni, prikladni za formalnu verifikaciju, manje podlozni greskama i lakse ih je transformisati, optimizovati i paralelizovati

Paralelizacija je moguca zbog transparentnosti referenci jer se mogu delovi izraza sracunati nezavisno, a onda se zdruziti naknadno

Medjutim, transparentnost referenci ima cenu

Ukoliko zelimo da imamo u potpunosti transparentnost referenci, ne smemo dopustiti nikakve propratne efekte  
U praksi je to veoma tesko jer postoje neki algoritmi koji se sustinski temelje na promeni stanja (npr random) dok neke funkcije postoje samo zbog svoji propratnih efekata (npr scanf, tj. funkcije za ulaz/izlaz)  
Zbog toga, vecina funkcionalnih programskih jezika dopusta kontrolisane propratne efekte (tj. imperativnost je prisutna u vecoj ili manjoj meri i na razlicite nacine)  
Vecina funkcionalnih jezika ukljucuju naredne imperativne osobine: promenljive (mutable variables) i konstrukte koji se ponasaju kao naredbe dodele

Staticko zakljucivanje tipova je manje fleksiblno ali efikasnije, dinamicko zakljucivanje tipova je fleksibilnije ali manje efikasno

Funkcionalni jezici: Haskell – staticko, Elixir – dinamicko

Funkcionalni programski jezici su najcesce 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 cesto automatski sam zakljuci tipove

Kod koji se pise najcesce sam po sebi polimorfan, a zakljucuju se najopstiji moguci tipovi na osnovu tipskih razreda

Lisp (lots of irritating silly parentheses)

Noviji funkcionalni jezici koriste sintaksu koja je slicna sintaksi imperativnih jezika:  
-*Pattern matching* - poklapanje obrazaca, pokusavamo da poklopimo vrednost prema obrascu, i, po potrebi, da vezemo promenljivu sa uspesnim poklapanjem  
-*Comprehensions* - skracenice

Izraz je **striktan** ako nema vrednost kad bar jedan od njegovih operanada nema vrednost  
Izraz je **nestriktan** kad moze da ima vrednost - cak i ako neki od njegovih operanada nema vrednost

Primer: a&b ima vrednost ⊥ ako a ima vrednost ⊥ - tj nije vazna vrednost izraza b

Kod striktne semantike, prvo se izracunaju vrednosti svih operanada, pa se onda izracunava vrednost izraza  
Kod nestriktne semantike izracunavanje operanada se odlaze sve dok te vrednosti ne budu neophodne. Ta strategija je poznata kao zadrzano ili lenjo izracunavanje (lazy evaluation)  
Nestriktna semantika omogucava kreiranje beskonacnih struktura i izraza

Vecina funkcionalnih jezika ima striktnu semantiku, npr Lisp, OCaml I Scala  
Miranda i Haskel imaju nestriktnu semantiku

Proces kompilacije vrlo slozen jer je asembler fakticki imperativan jezik, izazov napraviti kompajler koji podrzava transparentnost referenci, beskonacne strukture podataka, nestriktnu semantiku itd.

Postoje razne vrste sakupljaca otpadaka, Haskel koristi generacijski sakupljac otpadaka (objekti u memoriji se dele po starosti a ocekivanje je da ce mladi objekti biti brze iskorisceni i uklonjeni)

Prednosti I mane funkcionalnog programiranja:

Stanje i propratni efekti - za velike programe tesko je pratiti stanje i razumeti propratne efekte, lakse je kada imamo uvek isto ponasanje funkcija, ali svet koji nas okruzuje je pun promena i razlicitih stanja i nije prirodno da koncepti jezika budu u suportnosti sa domenom koji se modeluje.

Paralelno programiranje - jednostavno i bezbedno konkurentno programiranje je posledica transparentnih referenci, medutim, rad sa podacima koji se ne menjaju dovodi do moguceg rada sa podacima koji su umeduvremenu izmenjeni, dok rad sa podacima koji se menjaju jeste komplikovaniji i zahteva kodiranje kompleksne logike

Stil programiranja - programi su cesto kraci i laksi za citanje, treba znati procitati funkcionalni kod

Produktivnost programera - produktivnost je veca, ali vecina programera ne gradi nove sisteme vec rade na odrzavanju starih koji su pisani u drugim (imperativnim) jezicima

Prednosti:  
-Testiranje je jednostavnije jer je svaka funkcija potencijalni kandidat za unit testove, funkcije ne zavise od stanja sistema sto olaksava sintezu test primera i proveru da li je izlaz odgovarajuci  
-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 moze proveriti tako da se brzo izoluje koja funkcija je odgovorna za gresku

**LAMBDA RACUN**

Lambda racun se zasniva na apstrakciji i primeni funkcija koriscenjem vezivanja i supstitucije (zamene)

Lambda racun funkcije tretira kao izraze koji se postepeno transformisu do resenja, tj funkcija definise algoritam

Lambda racun je formalni model definisanja algoritma

Sintaksa lambda izraza: λpromenljiva.telo sa znacenjem promenljiva → telo

λx.x + 1 funkcija inkrementiranja x → x + 1  
λx.x funkcija identiteta, x → x  
λx.x · x + 3 kvadriranje i uvecanje za 3, x → x · x + 3

Lambda izraz se moze primenjivati na druge izraze  
Sintaksa: (λpromenljiva.telo)izraz - primena odgovara pozivu funkcije

(λx.x + 1)5  
(λx.x · x + 3)((λx.x + 1)5)

Promenljiva je validni **lambda term**  
λ-apstrakcija - ako je t lambda term, a x promenljiva, onda je λx.t lambda term  
λ-primena - ako su t i s lambda termovi, onda je (t s) lambda term

Lambda termovi se mogu konstruisati samo konacnom primenom prethodnih pravila

Ukoliko lambda racun ne ukljucuje konstante u definiciji, onda se naziva **cist**

Zagrade su vazne - termovi *λx.((λx.x+1)x)* i *(λx.(λx.x+1))x* su razliciti termovi

Primena funkcije je levo asocijativna. tj umesto *(e1e2)e3* mozemo krace da pisemo *e1e2e3*  
Apstrakcija je desno asocijativna, tj umesto *λx.(e1e2)* pisemo skraceno *λx.e1e2*  
Sekvenca apstrakcija moze da se skrati, npr *λx.λy.λz.e* se skraceno zapisuje kao *λxyz.e*

U termu *λx.x + y* promenljiva x je **vezana** a promenljiva y je **slobodna** promenljiva  
Slobodne promenljive u termu su one promenljive koje nisu vezane lambda apstrakcijom

Slobodna promenljiva terma *x* je samo x  
Skup slobodnih promenljivih terma *λx.t* je skup slobodnih promenljivih terma t bez promenljive x  
Skup slobodnih promenljivih terma *(t s)* je unija skupova slobodnih promenljivih terma t i terma s

Termovi *λx.x* i *λy.y* su α-ekvivalentni jer oba predstavljaju istu funkciju, tj identitet

**REDUKCIJE**

**δ redukcija**  
-Najprostiji tip lambda izraza su konstante jer se one ne mogu dalje transformisati  
-δ redukcija se oznacava sa →δ i odnosi se na trasformaciju funkcija koje kao argumente sadrze konstante  
-Na primer, 3 + 5 →δ 8  
-Ukoliko je jasno o kojoj redukciji je rec, onda se pise samo →

**α redukcija ili preimenovanje**   
-Dozvoljava da se promene imena vezanim promenljivama  
-Na primer, α redukcija izraza *λx.x* moze da bude u *λy.y*  
-Termovi koji se razlikuju samo po α konverziji su α ekvivalentni  
-Na primer *λx.yx = λz.yz = λa.ya*... (y nije vezana promenljiva i za nju ne mozemo da vrsimo preimenovanje)  
-Alfa preimenovanje je nekada neophodno da bi se izvrsila beta redukcija

α redukcija nije u potpunosti trivijalna, treba voditi racuna - *λx.λx.x* moze da se svede na *λy.λx.x* ali ne i na *λy.λx.y*Ukoliko funkciju *λx.λx.x* primenimo na npr broj 3, dobijamo preslikavanje kojim se broj tri preslikava u funkciju identiteta  
Ukoliko funkciju *λy.λx.x* primenimo na npr broj 3, ponovo dobijamo preslikavanje kojim se broj tri preslikava u funkciju identiteta  
Ukoliko funkciju *λy.λx.y* primenimo na npr broj 3, dobijamo preslikavanje kojim se broj tri preslikava u funkciju konstantnog preslikavanja u broj 3

Takodje, alfa redukcijom ne sme da se promeni ime promenljive tako da bude uhvaceno drugom apstrakcijom, na primer *λx.λy.x* smemo da zamenimo sa *λz.λy.z* ali ne smemo da zamenimo sa *λy.λy.y*

β-redukcija:

*(λx.x + 1)5 →β [5/x](x + 1) = 5 + 1 →δ 6*

*(λx.x · x + 3)((λx.x + 1)5) →β [6/x](x · x + 3) = 6 · 6 + 3 →δ 39*

Visestruka primena β-redukcije (oznaka *→→*β)  
*(λx.x · x + 3)((λx.x + 1)5) →→β 39*

f(x, y) = x + y se definise kao *λxy.x + y*

*((λxy.x + y)2)6 →β λy.(2 + y)6 →β 2 + 6 →δ 8*

**Karijev postupak**

Ako funkcija uzima vise 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, ciji je rezultat broj 10  
Zbog leve asocijativnosti ne moraju da se pisu zagrade i dovoljno je max 3 10  
Kod sqrt moraju zagrade

Visestrukom beta redukcijom izracunavamo vrednost izraza i zaustavljamo se tek onda kada dalja beta redukcija nije moguca  
Tako dobijen lambda izraz naziva se **normalni oblik** i on intuitivno odgovara vrednosti polaznog izraza

Church-Rosser teorema: ako se lambda izraz moze svesti na dva razlicita lambda izraza M i N, onda postoji treci izraz Z do kojeg se moze doci i iz M i iz N

Posledica teoreme je da svaki lambda izraz ima najvise jedan normalni oblik (dakle, ako postoji, on je jedinstven)

Poredak izvodenja redukcija

-Aplikativni poredak  
*(λx.5 · x)(2 + 1) →β (λx.5 · x)3 →β 5 · 3 → 15*Ovo odogovara pozivu po vrednosti (call-by-value) - izracunavamo vrednost argumenta i tek kada ga izracunamo saljemo ga u funkciju i funkcijadoceka u svom telu izracunati argument

-Poredak izvodenja redukcija  
Normalni poredak: beta redukcijom uvek redukovati najlevlji izraz  
*(λx.5 · x)(2 + 1) →β 5 · (2 + 1) → 5 · 3 → 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 izracunavanja  
  
  
AP: *(λx.1)(12345 · 54321) →β (λx.1)670592745 →β 1*NP: *(λx.1)(12345 · 54321) →β 1*

Normalni poredak, tj lenja evaluacija, nam garantuje zavrsetak izracunavanja uvek kada je to moguce  
AP: *(λx.1)((λx.x x)(λx.x x)) →β ...* (ne zavrsava)NP: *(λx.1)((λx.xx)(λx.x x)) →β 1*

Normalni poredak, tj lenja evaluacija, odgovara nestriktnoj semantici  
Normalni poredak, tj lenja evaluacija, omogucava koriscenje beskonacnih struktura

Efikasnost izracunavanja kod lenje evaluacije

-Postoje tehnike koje primenjuju kompajleri, a koje obezbeduju da se izracunavanja ne ponavljaju, ovo je vazno sa stanovista efikasnosti izracunavanja  
*(λx.x+x)(12345·54321) →β (12345·54321)+(12345·54321)* - ne bi bilo dobro dva puta nezavisno racunati proizvod (12345·54321) vec je potrebno to samo jednom uraditi, i za to postoje tehnike redukcije grafova