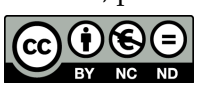
Programski jezici i paradigme

Milena Vujošević Janičić

26. april 2026.

Matematički fakultet, Univerzitet u Beogradu

Programski jezici i paradigme



Izdavač
Matematički fakultet, Univerzitet u Beogradu. Studentski trg 16, Beograd.
Za izdavača: *prof. dr XX*, dekan
Februar, 2026.

...

Predgovor

xxxxxxxxxxxxxxxxxxxxx

Beograd, 2026.

Milena Vujošević Janičić

Sadržaj

Uvod

Uvod u programske jezike i paradigme

1

Pregled

- ►
- ►
- ►
- ►

Jezik, u najopštijem smislu, predstavlja skup pravila koji omogućava komunikaciju između različitih subjekata. U svakodnevnom životu ljudi koriste prirodne jezike, u govornoj ili pisanoj formi. Razvoj računara i informacionih tehnologija uslovio je potrebu za nastankom posebne vrste jezika koja je namenjena komunikaciji između čoveka i mašine.

1.1 Definicija programskih jezika

Programski jezici predstavljaju formalizovane sisteme komunikacije. Njihova osnovna uloga jeste da omoguće zadavanje instrukcija računaru, ali se se takođe koriste i za komunikaciju između različitih računarskih sistema Za razliku od prirodnih jezika, koji su često višeznačni i kontekstualno zavisni, programski jezici teže preciznosti i jednoznačnosti.

Ne postoji jedinstvena, stroga matematička definicija programskog jezika, tako da se u literaturi mogu pronaći različita tumačenja ovog pojma. Programski jezik se može definisati kao formalno konstruisan jezik koji omogućava zadavanje instrukcija mašinama, posebno računarima. Takođe, može se posmatrati kao jezik za pisanje programa koje računar može da razume i izvrši, odnosno kao veštački jezik namenjen opisu računarskih procesa. U širem smislu, programski jezik predstavlja skup sintaktičkih i semantičkih pravila koji definišu kako se formiraju i izvršavaju programi.

Ključne karakteristike svakog programskog jezika su njegova sintaksa, koja određuje pravilnu strukturu izraza i naredbi, i semantika, koja određuje njihovo značenje. Zajedno, one čine osnovu za konstrukciju programa koji su istovremeno razumljivi ljudima i izvršivi na računarima.

Programski jezici se mogu klasifikovati na različite načine. Jedna od osnovnih podela jeste na mašinski zavisne i mašinski nezavisne jezike. Mašinski zavisni jezici, poput mašinskog koda ili asemblera, direktno su vezani za konkretnu arhitekturu računara i omogućavaju visok nivo kontrole nad hardverom, ali su teži za korišćenje i nisu prenosivi. dok mašinski nezavisni (viši) jezici omogućavaju viši nivo apstrakcije, veću čitljivost i lakšu prenosivost programa između različitih sistema. U ovom udžbeniku fokus će biti prvenstveno na mašinski nezavisnim programskim jezicima.

Broj programskih jezika koji su razvijeni tokom istorije računarstva izuzetno je veliki i meri se u hiljadama. Različiti izvori navode različite procene. Ipak, važno je naglasiti da svi programski jezici nemaju isti značaj niti su podjednako zastupljeni u praksi. Samo manji broj njih ima široku primenu u industriji i akademskoj zajednici.

Popularnost i značaj programskih jezika često se procenjuje pomoću različitih indeksa i analiza, kao što je TIOBE indeks, koji prati učestalost korišćenja jezika u različitim izvorima. Ipak, treba imati u vidu da popularnost ne mora nužno odražavati kvalitet ili pogodnost jezika za određeni problem, već pre svega njegovu rasprostranjenost i aktuelne trendove.

S obzirom na veliki broj postojećih programskih jezika, nemoguće je detaljno proučiti svaki od njih. Zbog toga se u praksi pristupa proučavanju opštih principa i koncepata koji su zajednički većem broju jezika. Upravo ti principi omogućavaju programerima da se lakše prilagođavaju novim jezicima i tehnologijama.

1.2 Paradigme i programski jezici

Pojam paradigma potiče iz grčkog jezika i označava uzor, obrazac ili model koji služi kao primer za ugled. U opštem smislu, paradigma predstavlja skup objekata ili pojava koje dele zajedničke karakteristike i koje se mogu posmatrati kao pripadnici iste klase.

U kontekstu programiranja, paradigma označava fundamentalni stil ili način razmišljanja o rešavanju problema pomoću računara. Programska paradigma definiše osnovne principe organizacije programa, način na koji se strukturiraju podaci i operacije nad njima, kao i način izražavanja programa.

Programske paradigme se često opisuju kao programski obrasci ili stilovi programiranja. One omogućavaju klasifikaciju programskih jezika na osnovu zajedničkih karakteristika koje dele. Na primer, neki jezici su zasnovani na imperativnom pristupu, gde se program sastoji od niza naredbi koje menjaju stanje sistema, dok drugi podržavaju deklarativni pristup, gde se opisuje šta treba izračunati, a ne kako to učiniti.

Razumevanje programskih paradigmi od suštinskog je značaja za efikasno programiranje, jer omogućava izbor odgovarajućeg pristupa u zavisnosti od prirode problema. Takođe, poznavanje različitih paradigmi olakšava učenje novih programskih jezika, budući da se mnogi jezici mogu posmatrati kao implementacije ili kombinacije postojećih paradigmi.

1.2.1 Povezanost paradigmi i jezika

Broj programskih paradigmi znatno je manji u odnosu na broj programskih jezika, što omogućava sistematičnije proučavanje osnovnih principa programiranja. Izučavanjem programskih paradigmi stiču se uvidi u globalna svojstva čitavih grupa jezika koji dele slične koncepte i pristupe rešavanju problema. Sama informacija da određeni programski jezik pripada konkretnoj paradigmi često je dovoljna da se stekne predstava o njegovim osnovnim karakteristikama.

Poznavanje određene programske paradigme značajno olakšava savladavanje novih programskih jezika koji toj paradigmi pripadaju. Umesto učenja svakog jezika od početka, programer se oslanja na već usvojene koncepte i obrasce razmišljanja, što ubrzava proces adaptacije. U tom smislu, vrednost znanja ne ogleda se samo u broju poznatih programskih jezika, već pre svega u razumevanju različitih paradigmi i njihovih reprezentativnih predstavnika.

Programske paradigme i programski jezici međusobno su usko povezani. Svakoj paradigmi odgovara veći broj programskih jezika koji implementiraju njene osnovne principe. Na primer, proceduralnoj paradigmi pripadaju jezici kao što su Pascal i C, dok objektno-orijentisanoj paradigmi pripadaju jezici poput Simule i Jave. Zbog toga je u praksi naročito važno izučavati najistaknutije predstavnike pojedinih paradigmi, jer oni najjasnije ilustruju njihove ključne ideje.

Savremeni programski jezici često nisu ograničeni samo na jednu paradigmu. Naprotiv, mnogi jezici podržavaju više paradigmi istovremeno. Na primer, jezik C++ omogućava korišćenje proceduralnog, objektno-orijentisanog i generičkog pristupa programiranju. Ovakva višestruka podrška pruža programerima fleksibilnost u izboru odgovarajućeg stila za rešavanje konkretnog problema.

Izbor programskog jezika predstavlja važan faktor u procesu razvoja softvera. Odgovarajući izbor može značajno pojednostaviti implementaciju rešenja, dok neadekvatan izbor može dovesti do nepotrebne složenosti. Zbog toga je razumevanje paradigmi ključno za donošenje informisanih odluka u vezi sa izborom jezika i pristupa programiranju.

1.3 Razvoj programskih jezika i paradigmi

Razvoj programskih jezika i paradigmi tesno je povezan sa razvojem računarske tehnike. Prvi elektronski raču-

nari pojavili su se sredinom XX veka. Jedan od ranih primera bio je računar ABC iz 1939. godine, namenjen rešavanju sistema linearnih jednačina. Nedugo zatim, 1946. godine, konstruisan je ENIAC, prvi elektronski računar opšte namene.

Krajem četrdesetih godina dolazi do značajne konceptualne promene u vidu fon Nojmanove arhitekture, koja postavlja osnovne principe organizacije računara kakvi se koriste i danas. Ova arhitektura, povezana sa računarom EDVAC iz 1951. godine, omogućila je skladištenje programa u memoriji računara, čime je otvoren put za razvoj savremenih programskih jezika.

1.3.1 Razvoj jezika

Razvoj programskih jezika započinje pedesetih godina XX veka. Jedan od prvih značajnih jezika bio je FORTRAN (FORmula TRANslating system, 1957), razvijen u kompaniji IBM pod vođstvom Džona Bakusa, namenjen prvenstveno naučnim i numeričkim proračunima. Ubrzo nakon toga nastaje LISP (List Processing, 1958), koji uvodi potpuno drugačiji pristup zasnovan na simboličkoj obradi podataka. COBOL (Common Business-Oriented language, 1959), čiji je razvoj povezan sa Grejs Hoper, bio je namenjen poslovnim aplikacijama.

Tokom šezdesetih godina razvijaju se jezici kao što su ALGOL, Simula i BASIC, koji postavljaju temelje za mnoge kasnije koncepte. Sedamdesete godine obeležene su pojavom jezika C, Pascal, Smalltalk i Prolog, koji predstavljaju ključne predstavnike različitih paradigmi. Osamdesetih godina nastaju jezici kao što su C++ i Erlang, dok devedesete donose veliki broj novih jezika, uključujući Haskell, Python, Java, JavaScript i druge. Razvoj se nastavlja i u XXI veku, uz pojavu jezika kao što su C#, Scala, F# i Elixir.

Veliki broj programskih jezika koji su nastali tokom vremena može se analizirati pomoću tzv. razvojnog stabla. Ovakav prikaz omogućava uvid u vremenski redosled nastanka jezika, kao i u njihove međusobne uticaje. Ipak, ne postoji jedinstveno razvojno stablo, jer

različiti autori naglašavaju različite jezike i veze među njima.

Razmatranje razvoja programskih jezika otvara i niz važnih pitanja: u kom periodu je nastao najveći broj jezika, koji jezici su imali najveći uticaj na dalji razvoj, kao i koji su razlozi za postojanje tako velikog broja različitih jezika. Odgovori na ova pitanja često su povezani sa tehnološkim napretkom, promenama u zahtevima korisnika i pojavom novih oblasti primene.

1.3.2 Razvoj programskih paradigmi

Razvoj programskih paradigmi tesno je povezan sa težnjom da se proces programiranja učini jednostavnijim i efikasnijim. Svaka nova paradigma obično je promovisana kroz jedan ili više programskih jezika koji su demonstrirali njene prednosti. Istovremeno, razvoj paradigmi bio je uslovljen i razvojem hardvera, jer su nove arhitekture omogućavale implementaciju složenijih koncepata.

Važno je naglasiti da ne postoji jedinstveno mišljenje o klasifikaciji programskih paradigmi. Različiti autori predlažu različite podele, u zavisnosti od kriterijuma koje smatraju relevantnim. Uprkos tome, razumevanje osnovnih paradigmi i njihovih karakteristika predstavlja ključ za uspešno savladavanje savremenih programskih jezika i tehnika programiranja.

1.3.3 Vrste programskih paradigmi

Najopštija i najčešće korišćena podela programskih paradigmi jeste na proceduralnu i deklarativnu paradigmu. Ova podela zasniva se na načinu na koji programer pristupa formulisanju rešenja problema.

U okviru proceduralne paradigme osnovni zadatak programera jeste da detaljno opiše postupak, odnosno proceduru, kojom se dolazi do rešenja problema. Ovaj pristup podrazumeva eksplicitno navođenje koraka koje

računar treba da izvrši. Fokus je, dakle, na pitanju kako se problem rešava.

Nasuprot tome, deklarativna paradigma podrazumeva drugačiji način razmišljanja. U ovom slučaju, programer opisuje šta predstavlja problem i koji su uslovi koje rešenje treba da zadovolji, dok izvršno okruženje jezika bavi pronalaženjem načina da se do tog rešenja dođe. Time se deo odgovornosti prebacuje sa programera na mehanizme implementirane u jeziku.

Pored ove osnovne podele, u literaturi se često izdvajaju četiri fundamentalne programske paradigme: imperativna, objektno-orijentisana, funkcionalna i logička paradigma. Imperativna paradigma zasniva se na modelu promenljivog stanja i izvršavanja naredbi. Objektno-orijentisana paradigma uvodi koncepte objekata, enkapsulacije, nasleđivanja i polimorfizma, omogućavajući modelovanje složenih sistema kroz međusobno povezane entitete. Funkcionalna paradigma posmatra program kao evaluaciju matematičkih funkcija i teži izbegavanju promenljivog stanja i sporednih efekata. Logička paradigma zasniva se na logici prvog reda, gde se program definiše skupom činjenica i pravila, a izvršavanje predstavlja proces zaključivanja.

Posebnu pažnju zaslužuje odnos između imperativne i proceduralne paradigme, koji u literaturi nije jednoznačno definisan. Prema jednom shvatanju, proceduralna paradigma obuhvata sve pristupe u kojima se opisuje algoritam rešavanja problema, pri čemu se imperativna paradigma može smatrati njenom podparadigmom. Prema drugom shvatanju, proceduralna paradigma predstavlja podskup imperativne paradigme, karakterisan organizacijom programa u podprograme (funkcije ili procedure). U ovom drugom slučaju, pojmovi imperativne i proceduralne paradigme često se koriste kao sinonimi.

Pored navedenih, postoje i brojne dodatne programske paradigme koje imaju značajnu primenu u savremenom razvoju softvera. Među njima se izdvajaju komponentna paradigma, paradigma upitnih jezika, generička paradigma, vizuelna paradigma, konkurentna paradigma,

reaktivna paradigma, skript paradigma, kao i paradigma programiranja ograničenja. Ove paradigme često nastaju kao odgovor na specifične zahteve u razvoju softvera i doprinose daljem obogaćivanju programerskih koncepata i tehnika.

Neke dodatne paradigme se mogu posmatrati i kao podparadigme ili kombinacije navedenih osnovnih pristupa. Granice između paradigmi nisu uvek strogo definisane, a savremeni programski jezici često podržavaju više različitih paradigmi istovremeno.

1.4 Domenski specifični jezici

Jezici opšte namene (engl. *general purpose languages* – GPL) predstavljaju programske jezike koji su namenjeni rešavanju širokog spektra problema u različitim domenima primene. Nasuprot njima, domenski specifični jezici (engl. *domain-specific languages* – DSL) predstavljaju jezike koji su dizajnirani sa ciljem da podrže rešavanje problema u jednom konkretnom domenu. Treba imati u vidu da domenski specifični jezici nisu obavezno i programski jezici, dok kada pričamo o jezicima opšte namene podrazumevamo da su u pitanju programski jezici. Domenski specifični jezici mogu se klasifikovati prema svojoj nameni. Važnu kategoriju čine jezici za obeležavanje teksta (npr. HTML, LaTeX) i jezici za stilizovanje (npr. CSS), koji nisu programski jezici. Pored njih, postoji i grupa jezika za modelovanje i formalnu specifikaciju (npr. VHDL, Verilog) kao i domenski specifični programski jezici koji omogućavaju implementaciju rešenja unutar konkretnog domena.

Iako se često posmatraju kao savremeni koncept, DSL jezici su prisutni još od samih početaka razvoja računarstva. Tipično se radi o relativno malim i specijalizovanim jezicima koji se fokusiraju na određene aspekte softverskih sistema. Njihova granica prema skript jezicima i bibliotekama često nije jasno definisana, budući da se DSL-ovi neretko koriste na način sličan bibliotekama unutar sistema implementiranih u jezicima opšte namene.

U praksi se retko dešava da se kompletna aplikacija razvija isključivo korišćenjem jednog DSL-a. Umesto toga, u okviru jednog sistema često se koristi više različitih DSL-ova, dok osnovu sistema čini jezik opšte namene.

Primer 1.4.1 Kao ilustrativan primer DSL-a može se navesti jezik DOT, koji se koristi za opisivanje grafova. Njegova jednostavna sintaksa omogućava korisnicima da na deklarativan način definišu čvorove i grane grafa, nakon čega se generiše odgovarajuća vizuelna reprezentacija.

1.4.1 Odnos sa jezicima opšte namene

Granica između jezika opšte namene i domenski specifičnih jezika nije uvek jasno definisana. Postoje jezici koji su inicijalno razvijeni za specifične potrebe, ali su vremenom našli širu primenu. Sa druge strane, postoje jezici koji su formalno opšte namene, ali se u praksi koriste gotovo isključivo u jednom domenu.

Na primer, programski jezik Perl je prvobitno razvijen za obradu teksta, slično alatima kao što su AWK ili shell skriptovi, ali je kasnije evoluirao u jezik opšte namene. Suprotno tome, PostScript je Tjuring-kompletan jezik, što znači da je teorijski sposoban da izrazi bilo koji izračunljiv problem, ali se u praksi koristi gotovo isključivo za opisivanje stranica za štampu.

Izbor odgovarajućeg jezika predstavlja važan aspekt razvoja softvera. U nekim situacijama dovoljno je koristiti postojeći jezik, bilo opšte namene ili domenski specifičan. Međutim, u određenim slučajevima opravdano je razviti novi DSL.

Kreiranje novog DSL-a ima smisla ukoliko ne postoji adekvatno rešenje, a specifičnost domena zahteva izražajniji i prirodniji način modelovanja problema. Takođe, važno je da se problemi tog tipa javljaju dovoljno često kako bi ulaganje u razvoj DSL-a bilo opravdano.

DSL može biti fokusiran na različite aspekte domena, kao što su način predstavljanja problema, način njegovog

rešavanja ili određene specifične operacije karakteristične za dati domen.

Jedna od ključnih prednosti DSL-ova jeste njihova jednostavnost upotrebe u poređenju sa opštim bibliotekama implementiranim u jezicima opšte namene. Dobro dizajniran DSL omogućava veću produktivnost programera, jer omogućava izražavanje složenih koncepata na koncizan i intuitivan način.

Pored toga, DSL-ovi unapređuju komunikaciju između programera i domenskih eksperata, jer koriste terminologiju i koncepte koji su bliski konkretnom domenu. Na taj način smanjuje se jaz između implementacije i specifikacije problema.

Takođe, DSL-ovi omogućavaju fokusiranje na suštinske aspekte problema, dok se implementacioni i tehnički detalji apstrahuju i skrivaju od korisnika jezika.

1.4.2 Osobine i vrste

Domenski specifični jezici mogu biti realizovani na različite načine:

- kao interpretirani jezici,
- kao kompajlirani jezici,
- kao jezici koji generišu kôd u jeziku opšte namene (npr. Java ili C),
- kao jezici koji generišu kôd u drugom DSL-u,
- kao jezici koji proizvode vizuelne reprezentacije ili druge oblike izlaza.

Primer DSL-a za specifičan domen predstavlja GCLC, jezik namenjen opisivanju geometrijskih konstrukcija. Ovaj jezik omogućava transformaciju tekstualnog opisa u različite izlazne formate, uključujući vektorske grafike i tekstualne reprezentacije kao što je XML.

Domenski specifični jezici mogu se podeliti na interne i eksterne.

Eksterni DSL-ovi imaju sopstvenu sintaksu i mogu se koristiti nezavisno od drugih jezika. Njihov razvoj ima

dugu tradiciju, posebno u Unix zajednici, a najčešće su tekstualni, iako mogu biti i grafički.

Interni DSL-ovi, poznati i kao ugrađeni (engl. *embedded*) DSL-ovi, implementirani su unutar nekog matičnog jezika. Oni koriste sintaksu i mehanizme tog jezika kako bi obezbedili specifičnu funkcionalnost. Ovakav pristup prisutan je još od ranih verzija Lisp jezika, ali i u savremenim jezicima kao što su Ruby, Java i C#.

Primer 1.4.2 Postoji veliki broj DSL jezika koji se koriste u različitim oblastima:

- HTML, CSS i LaTeX — jezici za opisivanje i stilizovanje teksta,
- XML — jezik za enkodiranje i razmenu podataka,
- ANTLR, Lex i Yacc — jezici za definisanje leksera i parsera,
- VHDL i Verilog — jezici za modelovanje hardvera,
- DOT — jezik za opisivanje grafova,
- GCLC — jezik za geometrijske konstrukcije,
- P4 — jezik za programiranje mrežnih uređaja.

Granica između DSL-ova i skript jezika nije uvek jasna. Na primer, jezici kao što su Matlab, Mathematica i Maple mogu se posmatrati i kao DSL-ovi i kao skript jezici, u zavisnosti od konteksta upotrebe.

Primer 1.4.3 SQL predstavlja interesantan primer domenski specifičnog jezika. Njegova namena je rad sa relacionim bazama podataka, što ga svrstava u DSL. Međutim, SQL poseduje značajan broj ključnih reči i funkcionalnosti, zbog čega je znatno kompleksniji od tipičnih DSL-ova.

Takođe, za razliku od većine DSL-ova koji su relativno laki za usvajanje, savladavanje SQL-a zahteva značajan vremenski i intelektualni napor. SQL se često koristi u kombinaciji sa jezicima opšte namene, kao što su Java, C ili C++.

Domenski specifični jezici predstavljaju važan koncept u savremenom razvoju softvera. Oni omogućavaju efikasnije rešavanje problema u specifičnim domenima kroz upotrebu jezika koji su prilagođeni konkretnim potrebama.

DSL-ovi se mogu značajno razlikovati po svojoj složenosti, ali im je zajednički cilj unapređenje produktivnosti, jasnoće i komunikacije u procesu razvoja softverskih sistema. Iako olakšavaju rad u određenim domenima, njihova upotreba često zahteva dodatno ekspertsko znanje iz oblasti primene.

1.4.3 Jezici za obeležavanje teksta i podataka

Pored programskih jezika, u savremenim informacionim sistemima široko se koriste i jezici za obeležavanje i strukturiranje teksta i podataka, kao što su HTML, CSS, XML, SGML i JSON. Iako imaju izuzetno važnu ulogu u razvoju softvera, ovi jezici se **ne mogu smatrati programskim jezicima** u klasičnom smislu.

Naime, po definiciji, programskim jezikom se opisuje neko izračunavanje, odnosno definišu se programi koje računar može da izvrši. Programski jezici omogućavaju definisanje algoritama, kontrolu toka izvršavanja, obradu podataka i interakciju sa okruženjem. Nasuprot tome, jezici za obeležavanje služe za opisivanje strukture i organizacije teksta ili podataka, a ne za izvršavanje programa.

Iz tog razloga, jezici za obeležavanje ne mogu se uklopiti u postojeće programske paradigme, niti sami po sebi predstavljaju posebnu programsku paradigmu. Njihova osnovna uloga je da omoguće jasno i formalno definisanje strukture informacija, koje zatim mogu biti prikazane, obrađene ili razmenjene između različitih sistema.

Jedan od najpoznatijih jezika za obeležavanje je HTML, koji se koristi za definisanje strukture veb stranica. HTML omogućava označavanje naslova, pasusa, tabela,

lista i drugih elemenata dokumenta, dok sam način njihovog prikaza određuje veb pregledač, često uz dodatne informacije definisane pomoću CSS-a.

Drugim rečima, HTML se koristi za opisivanje strukture sadržaja, a ne njegove funkcionalnosti. Za razliku od programskih jezika, HTML ne sadrži osnovne elemente programiranja, kao što su promenljive, funkcije, kontrola toka, evaluacija izraza ili ulazno-izlazne operacije. Zbog toga se HTML ne može koristiti za realizaciju algoritama niti za direktnu obradu podataka.

Slično važi i za druge jezike za obeležavanje i strukturiranje podataka, kao što su XML i JSON, koji se najčešće koriste za razmenu podataka između različitih sistema i aplikacija. Njihova osnovna namena je definisanje strukture podataka na jasan i standardizovan način.

Paralelno sa razvojem jezika za obeležavanje, posebno XML-a, razvijeni su i specijalizovani programski jezici namenjeni obradi i transformaciji strukturiranih podataka. Među takve jezike spadaju XSLT, XQuery i slični jezici koji omogućavaju pretragu, transformaciju i obradu podataka zapisanih u XML formatu.

Ovi jezici se, za razliku od samih jezika za obeležavanje, mogu svrstati u određene programske paradigme. Najčešće pripadaju domenski specifičnim jezicima ili skript jezicima, jer su usmereni na rešavanje konkretnih problema u oblasti obrade strukturiranih podataka.

Na taj način, iako jezici za obeležavanje ne predstavljaju programske jezike, oni imaju veoma važnu ulogu u savremenom razvoju softvera i predstavljaju osnovu za brojne alate i tehnologije koje se koriste u praksi.

Rezime

-
-
-
-

Ispitna pitanja

1.
2.
3.
4.
5.

Programski jezici i paradigme

Imperativno programiranje 2

Pregled

- ►
- ►
- ►
- ►

2.1 ...

2.2 ...

Rezime

- ►
- ►
- ►
- ►

Pitanja

1.
2.
3.
4.
5.

Objektno-orijentisano programiranje 3

Pregled

- ►
- ►
- ►
- ►

3.1 ...

3.2 ...

Rezime

- ►
- ►
- ►
- ►

Pitanja

1.
2.
3.
4.
5.

Funkcionalno programiranje 4

Pregled

-
-
-
-

Funkcionalna paradigma, ili funkcijska paradigma, je programska paradigma koja se zasniva na pojmu matematičke funkcije i u kojoj se izvršavanje programa svodi na evaluaciju izraza. Pojam funkcije u funkcionalnim jezicima se ne poklapa sa pojmom funkcije u tipičnim imperativnim jezicima. U funkcionalnom programiranju funkcija nije samo mehanizam organizacije koda, već osnovni nosilac značenja i računanja.

Imperativnu paradigmu karakteriše postojanje naredbi: izvršavanje programa svodi se na izvršavanje naredbi i na postepeno menjanje stanja memorije. Nasuprot tome, program se može posmatrati i kao izraz čija se vrednost izračunava. U zavisnosti od prirode izraza paradigma može biti logička ili funkcionalna. U logičkoj paradigmi izrazi su relacije, a rezultat je vrednost `true` ili `false`. U funkcionalnoj paradigmi izrazi su funkcije, a rezultat može biti proizvoljna vrednost odgovarajućeg tipa.

Različite paradigme zasnivaju se na različitim teorijskim modelima. Formalizam za imperativne jezike čine Tjuringova mašina i URM mašina, formalizam za logičke jezike je logika prvog reda, dok je formalizam za funkcionalne jezike lambda račun*.

Ekspresivnost funkcionalnih jezika ekvivalentna je ekspresivnosti lambda računa. Ekspresivnost lambda računa ekvivalentna je ekspresivnosti Tjuringovih mašina, a ekspresivnost Tjuringovih mašina ekvivalentna je ekspresivnosti imperativnih jezika. Zato važi važan

* Iako je objektno orijentisano programiranje veoma važna paradigma, za nju ne postoji formalizam koji je karakteriše.

zaključak: svi programi koji se mogu napisati imperativnim stilom mogu se napisati i funkcionalnim stilom i obratno.

Funkcionalni jezici su mnogo bliži svom teorijskom modelu nego imperativni jezici, pa poznavanje lambda računa doprinosi boljem razumevanju funkcionalnog programiranja. Lambda račun naglašava pravila za transformaciju izraza i ne vezuje se za konkretnu arhitekturu mašine koja računanje realizuje. U tome je jedno od njegovih najvećih teorijskih pojednostavljenja i jedna od najvećih prednosti funkcionalnog stila.

4.1 Funkcionalni stil programiranja

Funkcionalno programiranje je stil koji se zasniva na izračunavanju izraza kombinovanjem funkcija. U tom okviru osnovne aktivnosti jesu definisanje funkcije, primena funkcije i kompozicija funkcija. Program u funkcionalnom programiranju može se posmatrati kao niz definicija i primena funkcija, dok samo izvršavanje programa predstavlja evaluaciju funkcija.

Primer 4.1.1 Pretpostavimo da je definisana funkcija maksimuma:

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

Tada se funkcija sa tri argumenta može definisati kao primena dve funcije maksimuma sa dva argumenta:

$$\max_3(x, y, z) = \max(\max(x, y), z).$$

Na sličan način mogu se graditi i složenije funkcije, na primer za maksimum šest brojeva:

$$\max(\max_3(a, b, c), \max_3(d, e, f))$$

ili

$$\max_{3}(\max(a,b),\max(c,d),\max(e,f)).$$

Isti problem može se rešiti i postepenim ugnježđavanjem binarne funkcije maksimuma:

$$\max(\max(\max(a,b),\max(c,d)),\max(e,f)).$$

Osnovna apstrakcija funkcionalnog programiranja jeste funkcija. Funkcija je ravnopravna sa ostalim tipovima podataka i može biti i povratna vrednost i parametar druge funkcije.

Primer 4.1.2 Funkcija `duplo` prima drugu funkciju kao argument.

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

Da bi se uspešno programiralo u funkcionalnom stilu, neophodno je da jezik obezbedi osnovne funkcije ugrađene u sam sistem, mehanizme za formiranje novih i složenijih funkcija, strukture za predstavljanje podataka i biblioteku funkcija koje se mogu koristiti u daljem radu. Ukoliko je jezik dobro definisan, broj zaista osnovnih funkcija i struktura podataka ostaje relativno mali, dok se velika izražajna moć dobija kombinovanjem malog broja opštih mehanizama.

4.1.1 Istorijski razvoj

Funkcionalna paradigma nastaje krajem pedesetih godina dvadesetog veka, a njen prvi istaknuti predstavnik bio je Lisp iz 1959. godine. Naziv Lisp potiče od izraza *LISt Processing*.

Razvoj funkcionalne paradigme može se pratiti od kombinatorne logike iz 1924. godine, preko lambda računa iz 1930. godine i jezika Lisp iz 1959. godine, zatim kroz rad na SECD mašini i jeziku ISWIM 1964. godine, Backusov jezik FP iz 1977. godine, pojavu jezika Scheme i ML 1978. godine, Mirande 1985. godine, Erlanga 1986. godine i

konačno SML-a i Haskella 1990. godine. Kasniji razvoj uključuje i jezike kao što su OCaml iz 1996. godine, F# iz 2002. godine, Scala iz 2003. godine i Elixir iz 2012. godine.

Uz same funkcionalne jezike, važno je i to da su od druge decenije dvadeset prvog veka funkcionalni koncepti snažno podržani i u jezicima kao što su C++, Java i skript jezici, posebno Python.

Funkcionalna paradigma dobija snažan podsticaj Backusovim predavanjem povodom Tjuringove nagrade za 1977. godinu. John Backus je tu nagradu dobio za svoj rad na razvoju imperativnog jezika Fortran. Svaki dobitnik Tjuringove nagrade drži predavanje prilikom formalne dodele, a to predavanje se potom objavljuje u časopisu Communications of the ACM. Backus je u svom izlaganju tvrdio da su čisti funkcionalni programski jezici bolji od imperativnih jer su programi napisani u njima čitljiviji, pouzdaniji i verovatnije ispravni. Suština njegove argumentacije bila je da su funkcionalni jezici lakši za razumevanje, i tokom razvoja i nakon razvoja, zato što je vrednost izraza nezavisna od konteksta u kojem se izraz nalazi.

Slika 4.1: Backusovo predavanje kao važna istorijska tačka razvoja funkcionalne paradigme.

Iako jezik FP nije zaživeo kao praktični jezik, Backusovo izlaganje je snažno motivisalo nove debate i nova istraživanja.

Danas su funkcionalni jezici obično jezici opšte namene. Ne postoje strogo ograničeni specifični domeni za funkcionalno programiranje. Funkcionalni jezici su posebno pogodni za paralelno i distribuirano programiranje. Zbog toga savremeni jezici sve više usvajaju funkcionalne koncepte.

Slika 4.2: Lisp i njegova standardizovana varijanta ANSI Common Lisp.

4.1.2 Jezik Haskell

Haskell je savremeni funkcionalni jezik koji je dobio ime po Haskellu Brooksu Curryju (1900–1982), logičaru i matematičaru čiji su rezultati važni za teorijske osnove funkcionalnog programiranja.

Slika 4.3: Haskell Brooks Curry.

Razvoj Haskella započinje 1987. godine, kada međunarodni odbor kreće u dizajn novog zajedničkog funkcionalnog jezika. Već 1990. objavljena je specifikacija Haskell 1.0, u periodu od 1990. do 1997. usledile su četiri izmene standarda, 1998. definisan je Haskell 98, a 2010. objavljen je Haskell 2010.

Standardizaciju sprovodi međunarodni odbor *Haskell Committee*. Za sistematično proučavanje jezika posebno su korisni

- zvanična dokumentacija na adresi `https://www.haskell.org/documentation`,
- praktični izvori kao što je Real World Haskell, `http://book.realworldhaskell.org/`,
- primeri industrijskih primena `https://wiki.haskell.org/Haskell_in_industry` i
- pregled zajednice na `https://haskellcosm.com/`.

Haskell je čist funkcionalni jezik sa lenjom evaluacijom, pa se nepotrebna izračunavanja izbegavaju kad god je to moguće. Odlikuju ga moćan sistem tipova, automatsko zaključivanje tipova, statička i jaka tipizacija, parametarski polimorfizam i preopterećivanje. Pored toga, Haskell podržava kompaktan i ekspresivan način definisanja listi, funkcije višeg reda i monadičko programiranje, koje omogućava kontrolisane propratne efekte bez narušavanja transparentnosti referenci. Važan deo ekosistema čine razrađena biblioteka standardnih funkcija i dodatni moduli dostupni kroz Standard Library i Hackage, uz podršku za paralelno i distribuirano programiranje.

Alati i radno okruženje

GHC (Glasgow Haskell Compiler) je i kompajler za programski jezik Haskell. On proizvodi optimizovan kod koji se može koristiti i u industrijskim primenama. Kod se obično piše u datotekama sa ekstenzijom `.hs`.

GHCi je interaktivno okruženje u kome se mogu evaluirati izrazi, proveravati tipovi i učitavati moduli. Biblioteka `Prelude` sadrži osnovne funkcije koje su automatski dostupne.

Primer 4.1.3 Prikazane komande ilustruju osnovni rad u okruženju GHCi: dobijanje pomoći, ispis teksta, rad sa aritmetičkim izrazima, razliku između realnog i celobrojnog deljenja, kao i ponovno korišćenje poslednjeg dobijenog rezultata preko promenljive `it`. Komande `:help`, `:?` i `:h` služe za pomoć, dok `:quit` i `:q` izlaze 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
```

```
Prelude> putStrLn "Zdravo svima!"
Zdravo svima!
Prelude> 2+3-1
4
Prelude> 9/2
4.5
Prelude> div 9 2
4
Prelude> it^5
1024
```

Komanda `:help` prikazuje veliki broj dodatnih mogućnosti interpretera, na primer rad sa više linija, dodavanje modula, pregled definicija modula, promenu tekućeg direktorijuma, pravljenje oznaka za uređivač i definisanje novih komandi.

Za kompajliranje jednostavnog programa može se koristiti komanda:

```
ghc hello.hs
```

Ova komanda prevodi datoteku `hello.hs` u izvršivi program, pod pretpostavkom da je u njoj definisana funkcija `main`. Time se vidi osnovna podela rada u jeziku Haskell: definicije se mogu ili prevesti u samostalan izvršivi program, ili se mogu učitati u interaktivno okruženje radi ispitivanja i testiranja.

Ako želimo izvršivu verziju programa, potrebno je, dakle, definisati funkciju `main`. Ista datoteka zatim može da se učita i kao modul u interpreter:

```
ghc 1.hs
./1
Prelude> :load 1.hs
*Main> :module
```

Učitavanje modula u GHCi omogućava interaktivno proveravanje i dopunsko testiranje već napisanih definicija, bez potrebe da se ceo program svaki put pokreće iz početka. Komanda `:module` pritom služi za pregled ili promenu konteksta aktivnih modula.

4.2 Strukture podataka i tipovi

U funkcionalnom programiranju podaci i funkcije čine jedinstven konceptualni okvir: program se gradi od izraza, a izrazi uvek imaju neku vrednost i neki tip. Zato razmatranje struktura podataka i tipova nije samo tehničko pitanje implementacije, već jedan od osnovnih načina za razumevanje kako funkcionalni jezici modeluju probleme. Izbor odgovarajuće strukture podataka određuje način obrade informacija, dok sistem tipova obezbeđuje da se nad tim podacima primenjuju samo smislene operacije. Osnovne strukture podataka koje se javljaju u funkcionalnim jezicima su torke i liste.

4.2.1 Osnovni tipovi, torke i liste

Kao i drugi programski jezici, funkcionalni jezici raspolažu osnovnim tipovima podataka, kao što su celobrojne i realne vrednosti, logičke vrednosti i stringovi. U Haskellu, to su

```
Bool, Char, String, Int, Integer, Float
```

Primer 4.2.1 U sledećem primeru vidi se da GHCi automatski zaključuje tip izraza: promenljivu kojoj je pridružen broj `3` prepoznaje kao `Integer`, znak `'a'` kao `Char`, a string `"pera"` kao listu karaktera, odnosno `[Char]`.

```
1 Prelude> let x = 3
2 Prelude> :type x
3 x :: Integer
4 Prelude> let x = 'a'
5 Prelude> :type x
6 x :: Char
7 Prelude> let x = "pera"
8 Prelude> :type x
9 x :: [Char]
```

Ipak, za funkcionalno programiranje posebno su važne one strukture koje prirodno podržavaju rekurzivnu

obradu podataka. Zbog toga lista zauzima centralno mesto u gotovo svim funkcionalnim jezicima. Pored listi, značajnu ulogu imaju i torke, koje omogućavaju objedinjavanje vrednosti različitih tipova u jedinstvenu celinu.

Često torke zauzimaju kontinualni prostor u memoriji, slično nizovima, dok su liste tipično implementirane preko povezanih listi. Izbor odgovarajuće strukture zato zavisi od prirode problema, a naročito od toga da li je broj elemenata unapred poznat i fiksiran ili treba da ostane promenljiv.

Torke su kolekcije fiksiranog broja vrednosti, ali potencijalno različitih tipova. Za parove postoje ugrađene selektorske funkcije `fst` i `snd`. Torke mogu biti i parametri i povratne vrednosti funkcija.

Primer 4.2.2 U ovom primeru definisana je torka sa dva elementa različitih tipova. Komentari pokazuju kako se pomoću funkcija `fst` i `snd` pristupa prvom, odnosno drugom elementu para.

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

Liste su kolekcije proizvoljnog broja vrednosti istog tipa. One mogu biti i parametri i povratne vrednosti funkcija i predstavljaju najvažniju radnu strukturu u Haskellu. Zapis listi u Haskellu je kompaktan i izražajan.

Primer 4.2.3 Sledeći zapis objedinjuje više uobičajenih načina zadavanja listi: eksplicitno navođenje elemenata, zadavanje intervala, koraka, beskonačne liste i opadajuće vrednosti.

```
1 Prelude> [1,2,3]
2 [1,2,3]
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> ['A'..'F']
"ABCDEF"
Prelude> ['A'..'z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_'
abcdefghijklmnopqrstuvwxyz"
Prelude> [5..1]
[]
Prelude> [5,4..1]
[5,4,3,2,1]
Prelude> [1,6..]   -- beskonačna lista
```

Primer 4.2.4 Sledeće komande ilustruju tipične operacije nad listama: izdvajanje glave liste, određivanje dužine, uzimanje početnog segmenta, računanje zbira i pristup elementu po indeksu. Poslednji primer dodatno pokazuje lenjo izračunavanje: lista jeste beskonačna, ali se izračunava samo do traženog elementa.

```
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]
783
Prelude> [0..10] !! 5
5
Prelude> [0,2..] !! 50
100
```

Primer 4.2.5 Naredni primer upoređuje dve prirodne reprezentacije polinoma. Polinom drugog stepena može se predstaviti torkom:

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

Polinom proizvoljnog stepena prirodno se predstavlja listom:

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

Prvi zapis je pogodan kada je broj koeficijenata unapred poznat i fiksiran. Drugi zapis je pogodniji kada broj koeficijenata može da varira, jer liste nemaju unapred zadatu dužinu.

4.2.2 Sistem tipova

Sistem tipova određuje kako se u jeziku opisuju vrednosti i koje su operacije nad njima dozvoljene. Zaključivanje tipova može biti statičko i dinamičko. Statičko zaključivanje tipova vrši se u fazi kompilacije; ono je manje fleksibilno, ali omogućava ranije otkrivanje grešaka i efikasnije izvršavanje. Dinamičko zaključivanje tipova vrši se u fazi izvršavanja; ono je fleksibilnije, ali deo provere ostavlja za trenutak kada program već radi. Funkcionalni jezici mogu koristiti oba pristupa. Haskell je statički tipiziran, dok je primer dinamičkog zaključivanja prisutan u jeziku Elixir.

U funkcionalnom programiranju u Haskellu tipovi imaju naročito važnu ulogu zato što već u fazi prevođenja omogućavaju proveru da li su izrazi smisleno sastavljeni i da li se funkcije primenjuju na odgovarajuće argumente. Time sistem tipova postaje ne samo tehnički mehanizam kontrole, već i sredstvo za precizno opisivanje namene programa.

Za Haskell je važno i to da je strogo tipiziran jezik. To znači da se tipovi moraju poklapati i da nema prećutnih,

implicitnih konverzija koje bi mogle prikriti grešku u programu. Zbog toga izraz u kome se pokušava kombinovanje nekompatibilnih tipova biva odbačen pre izvršavanja, umesto da se problem otkrije tek naknadno.

Još jedna važna osobina Haskella je to da se tipovi ne moraju uvek navoditi, jer postoji automatsko zaključivanje tipova. Prevodilac iz same definicije funkcije i iz načina njene upotrebe izvodi najopštiji tip koji ta definicija može da ima. Ta osobina čini zapis kraćim, ali ne ukida mogućnost da se tip napiše eksplicitno kada su potrebni veća preciznost, bolja dokumentovanost ili jasnije definisan interfejs funkcije.

Navođenje tipova jeste eksplicitno zapisivanje potpisa pomoću operatora `::`. Tim zapisom se navodi kog je tipa neka vrednost, promenljiva ili funkcija. Na primer, zapis `x :: Integer` znači da je `x` celobrojna vrednost tipa `Integer`, dok zapis `f :: Int -> Int` znači da funkcija `f` prima argument tipa `Int` i vraća rezultat istog tipa. Eksplicitno navođenje tipova nije obavezno u svim situacijama, ali je u akademskom i produkcionom kodu veoma korisno jer čini program preglednijim i olakšava proveru ispravnosti.

4.2.3 Tipovi funkcija i polimorfizam

Tip funkcije opisuje preslikavanje iz jednog tipa u drugi. Na primer, zapisi `Bool -> Bool`, `Int -> Int`, `[Char] -> Int` i `(Int, Int) -> Int` pokazuju da funkcija može preslikavati logičku vrednost u logičku vrednost, ceo broj u ceo broj, listu karaktera u broj ili par brojeva u jedan broj. Strelica `->` odvaja tip argumenta od tipa rezultata, a kod funkcija sa više argumenata zapis se čita zdesna nalevo, što će kasnije biti povezano sa Karijevim postupkom.

U praksi se tipovi mogu proveravati i interaktivno. U okruženju GHCi za to služi komanda `:type` ili njen skraćeni oblik `:t`. Na taj način se može videti kako je prevodilac razumeo dati izraz, čak i onda kada tip nije eksplicitno naveden.

Primer 4.2.6 Polimorfizam funkcije `uvecaj` vidi se u tome što se ista definicija može primeniti na različite numeričke tipove, sve dok oni podržavaju operaciju sabiranja i literal `1`.

```
Prelude> let uvecaj x = x+1
Prelude> uvecaj 5
6
Prelude> uvecaj 55.5
56.5
Prelude> uvecaj
    12345678901234567890123456789012345678901
12345678901234567890123456789012345678902
```

Tipske promenljive se koriste za definisanje tipova kod polimorfnih funkcija.

Primer 4.2.7 Tipska promenljiva `a` pokazuje da funkcije `length` i `reverse` rade nad listama proizvoljnog tipa:

```
Prelude> :type length
length :: [a] -> Int
Prelude> :type reverse
reverse :: [a] -> [a]
```

Funkcija `length` vraća dužinu liste, dok `reverse` vraća istu listu obrnutim redosledom.

```
Prelude> length [1,2,3]
3
Prelude> length "abcde"
5
```

Preopterećivanje se u Haskellu ostvaruje preko tipskih razreda ili klasa. Tipovi se tako mogu grupisati po operacijama koje podržavaju. Polimorfizam sa tipskim promenljivama koristimo kada ista struktura podataka može sadržati različite tipove elemenata, na primer listu brojeva ili listu karaktera. Preopterećivanje koristimo kada su nad različitim tipovima definisane iste operacije,

na primer sabiranje celih i realnih brojeva.

Definicija 4.2.1 *Tipski razred (klasa) definiše koje funkcije neki tip mora da implementira da bi pripadao tom razredu.*

Među osnovnim tipskim razredima nalaze se `Eq` za tipove sa jednakošću, `Ord` za tipove sa uređenjem, `Num` za numeričke tipove, `Integral` za celobrojne tipove, `Fractional` za razlomačke tipove, kao i `Show` (tip koji se može konvertovati u string), `Read` (tip koji se može konstruisati na osnovu stringa) i `Bounded` (tipovi koji imaju gornje i donje ograničenje). Razred `Ord` nasleđuje `Eq`, a `Num` nasleđuje `Ord`. Razred `Integral` nasleđuje `Num` i uvodi operacije `div` i `mod`, dok `Fractional` isto nasleđuje `Num` i uvodi operacije `/` i `recip`.

Primer 4.2.8 Potpisi pokazuju kako se u Haskellu tip i ograničenja nad tipovima navode eksplicitno. Na primer, `sum` zahteva numerički tip, `elem` tip sa jednakošću, a `max` tip sa uređenjem.

```
1 sum  :: Num a => [a] -> a
2 elem :: Eq a => a -> [a] -> Bool
3 max  :: Ord a => a -> a -> a
4 head :: [a] -> a
5 fst  :: (a, b) -> a
6 inc  :: Num a => a -> a
7 map  :: (a -> b) -> [a] -> [b]
```

Funkcija `sum` prima listu elemenata tipa `a` i vraća vrednost tipa `a`, pri čemu `a` mora biti numerički tip.

Funkcija `elem` prima vrednost tipa `a` i listu elemenata tipa `a`, i vraća logičku vrednost (Bool), pri čemu `a` mora podržavati poređenje na jednakost (Eq).

Funkcija `max` prima dve vrednosti tipa `a` i vraća veću od njih, pri čemu a mora biti tip koji podržava poređenje (Ord).

Funkcija `head` prima listu i vraća element istog tipa kao što su elementi liste, pri čemu ne postoje ograničenja

nad tipom elemenata koji čine listu.

Funkcija `fst` prima par vrednosti tipova `a` i `b` i vraća vrednost tipa `a`, tj. prvi element para.

Funkcija `inc` prima vrednost tipa `a` i vraća vrednost istog tipa, pri čemu `a` mora biti numerički tip.

Funkcija `map` prima funkciju koja preslikava `a` u `b` i listu elemenata tipa `a`, i vraća listu elemenata tipa `b` dobijenu primenom te funkcije na svaki element liste.

Primer 4.2.9 Tipski razredi ograničavaju gde se neka funkcija može upotrebiti. Funkcija `sum` radi samo nad numeričkim tipovima, pa se uspešno primenjuje na listu brojeva, ali ne i na listu karaktera. Funkcija `elem` radi nad tipovima sa jednakošću, a `max` nad tipovima sa uređenjem.

```
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
    ']

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"
```

Nad jednom tipskom promenljivom moguće je postaviti više ograničenja, a moguće je imati i više tipskih

promenljivih sa različitim ograničenjima.

Primer 4.2.10 Sledeći potpisi pokazuju da jedna funkcija može zahtevati više tipskih osobina istovremeno. Na taj način se polimorfizam precizira dodatnim uslovima koje tipovi moraju da zadovolje.

```
f1 :: (Num a, Show b) => (a -> b) -> (a -> b)
f2 :: (Ord a, Show a) => [a] -> a -> [String]
```

Funkcija `f1` prima funkciju koja preslikava vrednosti tipa `a` u vrednosti tipa `b` i vraća funkciju istog tipa, pri čemu `a` mora biti numerički tip, a `b` mora biti tip koji se može prikazati kao tekst (Show).

Funkcija `f2` prima listu elemenata tipa `a` i jednu vrednost tipa `a`, i vraća listu stringova ([String]), pri čemu `a` mora biti tip koji podržava poređenje (Ord) i konverziju u tekst (Show).

Primer 4.2.11 (Polimorfnost konstanti.) Sledeći primer pokazuje da numerička konstanta nije vezana za jedan konkretan tip dok se taj tip ne zahteva u kontekstu. Tek dodatnom anotacijom ili odgovarajućom upotrebom određuje se da li je reč o tipu `Int`, `Integer`, `Float` ili `Double`.

```
ghci> :t 20
20 :: Num t => t
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

4.3 Osnovna svojstva funkcionalnih jezika

Osnovna svojstva funkcionalnih jezika neposredno proizlaze iz njihovog matematičkog i teorijskog utemeljenja. Za razliku od imperativnih jezika, u kojima je u središtu pažnje promena stanja programa kroz niz naredbi, funkcionalni jezici polaze od ideje da se računanje može izraziti kao evaluacija funkcija i izraza. Zbog toga se ključne osobine ovih jezika ne mogu razumeti samo na nivou sintakse, već pre svega kroz način na koji oni predstavljaju podatke, funkcije i samo izvršavanje programa.

U ovom kontekstu posebno su važni sledeći pojmovi: funkcije kao građani prvog reda (*first class citizen*), oslanjanje na matematički model funkcije, odsustvo implicitnog stanja i svojstvo transparentnosti referenci. Upravo ta svojstva određuju način na koji se funkcionalni programi pišu, čitaju, analiziraju i transformišu. Ona ujedno objašnjavaju i zbog čega su funkcionalni jezici pogodni za modularno projektovanje, formalnu analizu i paralelno izračunavanje, ali i zašto se u njima pojedini problemi rešavaju drugačije nego u imperativnim jezicima.

4.3.1 Funkcije i transparentnost referenci

Definicija 4.3.1 *Funkcija je preslikavanje elemenata jednog skupa (domena) u elemente drugog skupa (kodomena).*

Definicija funkcije uključuje zadavanje domena, kodomena i preslikavanja. Preslikavanje se može zadati izrazom ili tabelom.

Primer 4.3.1 Za funkciju

$$\text{kub}(x) \equiv x \cdot x \cdot x$$

parametar x tokom evaluacije predstavlja vezanu vred-

nost i ne menja se, za razliku od promenljive u imperativnom jeziku. Znak ≡ znači „definiše se kao“, pa zapis kaže da je za svaki realan broj x vrednost funkcije kub(x) jednaka proizvodu $x \cdot x \cdot x$. Na primer, kub(2.0) = $2.0 \cdot 2.0 \cdot 2.0 = 8$.

Primer 4.3.2 Jedna od osnovnih karakteristika matematičkih funkcija je da se izračunavanje preslikavanja kontroliše rekurzijom i kondicionalnim izrazima, a ne sekvencom i iteracijom.

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

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

Definicija 4.3.2 *Za neki gradivni element se kaže da je građanin prvog reda (first class citizen) ako u okviru jezika ne postoje restrikcije po pitanju njegovog kreiranja i korišćenja.*

U funkcionalnom programiranju funkcije su građani prvog reda. To znači da se sa njima postupa kao sa svakim drugim podatkom: mogu se prosleđivati, vraćati, kombinovati i čuvati za kasniju upotrebu.

Primer 4.3.3 Korisno je uporediti ovaj pojam sa jezikom C. Brojevi, znakovi i pokazivači mogu se posmatrati kao građani prvog reda jer se mogu čuvati u promenljivama, prosleđivati funkcijama i vraćati kao rezultat. Funkcije u jeziku C nisu građani prvog reda u punom smislu, već se njima rukuje preko pokazivača na funkcije. Nizovi u C-u takođe nisu građani prvog reda.

Primena funkcije je izračunavanje vrednosti funkcije za konkretne argumente. Matematički posmatrano, ona

odgovara uparivanju funkcije sa elementima njenog domena, dok se rezultat dobija evaluacijom izraza koji definiše dato preslikavanje. U funkcionalnom programiranju upravo je taj čin primene centralni mehanizam računanja.

Funkcije višeg reda imaju jednu ili više funkcija kao parametre, ili imaju funkciju kao rezultat, ili oba.

Primer 4.3.4 Ako su $f(x) = x + 5$ i $g(x) = 2 \cdot x$, onda je njihova kompozicija

$$h = f \circ g, \qquad h(x) = f(g(x)) = (2 \cdot x) + 5.$$

Kompozicija funkcija je primer funkcije višeg reda koja i kao argumente i kao povratnu vrednost ima funkciju.

Primer 4.3.5 Funkcija `map` je funkcija višeg reda. Svi ovi primeri pokazuju da funkcija `map` primenjuje zadatu funkciju na svaki element strukture. Funkcija može biti aritmetička, string-operacija, kompozicija funkcija, pa čak i funkcija koja se primenjuje na elemente ugnježdenih listi ili torki.

```
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]
Prelude> map ((>10) . (^2)) [1,2,3,4,5,6]
-- map (>10) (map (^2) [1,2,3,4,5,6])
[False,False,False,True,True,True]
```

U apstraktnijem zapisu α funkcija, odnosno apply to

all, prima funkciju kao parametar i pri primeni na listu daje novu listu dobijenu primenom te funkcije na svaki element. Ako je $f(x) = 2 \cdot x$, onda je $\alpha(f, (1,2,3)) = (2,4,6)$.

Primer 4.3.6 Funkcija `filter` je funkcija višeg reda. U narednim primerima `filter` zadržava samo one elemente koji zadovoljavaju zadati uslov. Tako se iz liste mogu izdvojiti samo elementi veći od tri, samo trojke ili samo parni brojevi.

```
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]
Prelude> length
          (filter (==True)
            (map ((>3) . (\x -> sqrt((fst x)^2 +
    (snd x)^2)))
              [(1,2), (2,2), (1,0), (3,5), (-2,
    -1), (-1, 5), (4, 2)]))
3
```

Apstraktna φ funkcija, odnosno `filter`, prima predikat i iz liste zadržava samo one elemente za koje taj predikat vraća vrednost `True`. Ako je $f(x) = odd(x)$, tada je $\varphi(f, (1,2,3)) = (1,3)$.

Primer 4.3.7 Funkcija `fold` je funkcija višeg reda. Primer poredi levo i desno savijanje liste. Za sabiranje je rezultat isti, dok se kod oduzimanja vidi da smer obrade elemenata utiče na rezultat, što jasno razdvaja ulogu funkcija `foldl` i `foldr`.

```
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))
Prelude> map ((>1) . foldl (+) 0) [[1,2,3],
    [4,5,6]]
[True, True]
```

Apstraktna ρ funkcija, odnosno reduce, jednu listu svodi na jednu vrednost. Ako je $f(x, y) = x + y$, tada je $\rho(f, (1, 2, 3)) = 6$.

Matematičke funkcije nemaju propratne efekte i ne zavise od globalnih ili nelokalnih promenljivih. Zato isti argument uvek daje isti rezultat. Upravo ta osobina predstavlja jedan od osnovnih uzora funkcionalnog programiranja. U imperativnim jezicima takvo ponašanje ne mora da važi, jer rezultat poziva može zavisiti od prethodnog toka izvršavanja i od stanja memorije.

Primer 4.3.8 U narednom primeru funkcija zavisi od tekućeg stanja promenljive `x`, pa uzastopni pozivi ne daju isti rezultat:

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

Za razumevanje ovog koda neophodno je poznavati tekuće stanje programa.

Stanje je osnovna karakteristika imperativnih jezika. Stanje programa čine sve vrednosti u memoriji kojima program tokom izvršavanja ima pristup.

Primer 4.3.9 Promena stanja najčešće se vrši naredbom dodele, eksplicitno ili skriveno.

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

Definicija 4.3.3 *Propratni efekat je svaka promena implicitnog stanja.*

Primer 4.3.10 Zbog promene globalne promenljive `y`, različiti konteksti poziva funkcije `foo` mogu dati neočekivane rezultate.

```
1 int y;
2
3 int foo(int x){
4     y = 0;
5     return 2*x;
6 }
```

Primer 4.3.11 Suština problema propratnih efekata vidi se u različitim kontekstima primene iste funkcije. Iako `foo(2)` uvek vraća broj 4, sporedna promena promenljive `y` menja ponašanje šireg izraza:

```
1 y = foo(2);               // y = 4
2 if (y == foo(2)) { ... } // netačno, jer foo
      postavi y na 0
3
4 y = 5;
5 x = foo(2);
6 z = x + y;                // z = 4
7
8 y = 5;
9 z = foo(2) + y;           // z = 4
```

Funkcionalni jezici nemaju implicitno stanje. Izvođenje programa svodi se na evaluaciju izraza, i to bez oslanjanja na promenljivo stanje memorije. Pošto ne postoje promenljive u imperativnom smislu niti naredba dodele kao osnovni mehanizam promene, iterativne konstrukcije nisu prirodne u istom smislu kao u imperativnim jezicima. Zbog toga se ponavljanje najčešće izražava rekurzijom.

Primer 4.3.12 Funkcija `fact` definiše faktorijel rekurzivno, bez promenljivih koje menjaju stanje i bez naredbe dodele. Time se ponavljanje izražava kroz samu definiciju funkcije. Primer pokazuje da je rekurzija u funkcionalnim jezicima prirodna i direktno prati matematičku definiciju funkcije.

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

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

U modernom funkcionalnom programiranju rekurzija je često skrivena kroz upotrebu osnovnih funkcija višeg reda.

Definicija 4.3.4 *Transparentnost referenci znači da je vrednost izraza svuda jedinstveno određena: ako se na dva mesta referencira isti izraz, vrednost je ista i izraz se može zameniti tom vrednošću bez promene ponašanja programa.*

Primer 4.3.13 Ako je promenljiva `x` definisana kao `fact 5`, onda se u izrazu `x + x` svako pojavljivanje `x` može zameniti izrazom `fact 5`. Polazna definicija izgleda ovako:

```
x = fact 5
```

U funkcionalnom programiranju izrazi

$$f(2) + g(3) \qquad \text{i} \qquad g(3) + f(2)$$

imaju istu vrednost. U imperativnim jezicima to ne mora biti tačno, jer poziv funkcije može promeniti stanje programa.

Primer 4.3.14 Ovaj primer pokazuje zašto je u imperativnim jezicima redosled poziva bitan. Funkcija `f` menja globalnu promenljivu `y`, pa rezultat izraza koji kombinuje `f(2)` i `g(3)` zavisi od toga koja funkcija se poziva prva.

```
1  int y;
2
3  int f(int x) {
4      y = 0;
5      return 2*x;
6  }
7
8  int g(int x) {
9      return x + 2*y;
10 }
11 int main() {
12     y = 5;
13     int a = f(2) + g(3);
14
15     y = 5;
16     int b = g(3) + f(2);
17
18     printf("a = %d\nb = %d\n", a, b);
19 }
20 a = 7
21 b = 17
```

Iako se pre poziva funkcija postavi `y = 5`, izrazi `f(2) + g(3)` i `g(3) + f(2)` u ovom primeru daju različite rezultate: 7 i 17. Redosled poziva funkcija se mora pažljivo pratiti.

Transparentnost referenci donosi niz važnih pogodnosti. Programi su formalno koncizni, prikladni za formalnu verifikaciju, manje podložni greškama i lakši za transformaciju, optimizaciju i paralelizaciju. Paralelizacija je moguća zato što se delovi izraza mogu računati nezavisno, a zatim naknadno objediniti bez bojazni da će međusobno uticati jedni na druge. Ipak, transparent-

nost referenci ima i svoju cenu, jer otežava neposredno zapisivanje algoritama koji su suštinski zasnovani na promeni stanja.

Ako želimo potpunu transparentnost referenci, ne smemo dopustiti nikakve propratne efekte. U praksi je to teško, jer neki algoritmi suštinski koriste stanje, a neke funkcije postoje upravo zbog propratnih efekata, na primer ulazno-izlazne funkcije.

Zbog toga većina funkcionalnih jezika dopušta kontrolisane propratne efekte. U zavisnosti od prisutnosti imperativnih osobina, razlikuju se čisti funkcionalni jezici i funkcionalni jezici koji nisu čisti. Čisti funkcionalni jezici koriste dodatne mehanizme kako bi omogućili izračunavanje sa stanjem, a istovremeno zadržali transparentnost referenci. Mali je broj takvih jezika, na primer Haskell, Clean i Miranda.

Većina funkcionalnih jezika uključuje i neke imperativne osobine, pre svega promenljive koje mogu menjati vrednost (mutable variables) i konstrukte koji se ponašaju kao naredbe dodele. Upravo zato treba razlikovati jezike koji samo podržavaju funkcionalni stil od jezika koji su zaista čisti funkcionalni jezici.

Korisni praktični izvori o modelovanju slučajnosti i propratnih efekata u čistim funkcionalnim jezicima dostupni su i na narednim adresama funkcija random i propratni efekti.

Funkcionalni jezici su jezici koji podržavaju i ohrabruju funkcionalni stil programiranja. Primeri su SML, Clean, Haskell i Miranda. Moderni višeparadigmatski jezici kao što su Common Lisp, OCaml, Scala, Python, Ruby, F# i Clojure takođe u različitoj meri podržavaju funkcionalni stil. Neki imperativni jezici, kao što su C# i Java, eksplicitno podržavaju funkcijske koncepte, dok su u drugima, kao što su C i C++, oni ostvarivi samo posredno. Ako je čisto funkcionalni podskup takvog jezika vrlo slab, takav jezik se obično ne naziva funkcionalnim jezikom.

4.3.2 Sintaksa, semantika i implementacija

Da bi se funkcionalni jezici razumeli u potpunosti, nije dovoljno posmatrati samo njihove osnovne ideje i karakteristična svojstva. Potrebno je razmotriti i na koji način su programi u tim jezicima zapisani, kako se određuje značenje takvih zapisa i kako se oni konačno prevode u oblik koji računar može da izvrši. Zato se proučavanje funkcionalnih jezika prirodno oslanja na tri međusobno povezana nivoa: sintaksu, semantiku i implementaciju.

Sintaksa opisuje kako se ispravni programi grade od osnovnih jezičkih elemenata, semantika objašnjava kako se ti programi izvršavaju i kakvo značenje imaju njihovi izrazi, dok implementacija pokazuje kako se takvi apstraktni modeli realizuju u konkretnim prevodiocima i izvršnim sistemima. U funkcionalnom programiranju ova pitanja imaju poseban značaj, jer su pojmovi kao što su rekurzija, poklapanje obrazaca, lenja evaluacija i transparentnost referenci tesno povezani i sa oblikom programa i sa načinom njegovog izvršavanja.

Sintaksa

Prvi funkcionalni jezik, Lisp, koristi sintaksu koja se znatno razlikuje od sintakse tipičnih imperativnih jezika. Naziv Lisp tradicionalno se objašnjava kao *List Processing*, a često se navodi i duhovita interpretacija *Lots of Irritating Silly Parentheses*. Upravo ta izrazito zagrađena sintaksa pokazuje da funkcionalni jezici istorijski nisu nastajali kao varijacija imperativnih jezika, već iz sopstvenih teorijskih osnova i sa sopstvenim principima izražavanja.

Primer 4.3.15 Sledeći kod predstavlja standardnu rekurzivnu definiciju faktorijela u Lispu. Ako je argument nula, vraća se vrednost 1, a u suprotnom se broj množi faktorijelom prethodnika.

```
1 (define (factorial n)
```

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

Izrazi u Lisp-u su prefiksni.

Primer 4.3.16 U istom jeziku mogu se napisati i rekurzivna i iterativna verzija istog algoritma. Druga definicija uvodi pomoćnu funkciju `iter`, koja nosi trenutno akumulirani proizvod i brojač, pa na taj način oponaša imperativni stil bez eksplicitne petlje.

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

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

Primer 4.3.17 U sledećoj varijanti (jezik Scheme) upotrebljen je konstruktor `cond`, koji omogućava grananje kroz više uslova. Funkcija pritom eksplicitno obrađuje i negativan argument, za koji vraća vrednost `#f`.

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

Primer 4.3.18 Lisp, kao višeparadigmatski jezik, dozvoljava i konstrukcije koje odgovaraju petljama:

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

Ovaj zapis prolazi kroz vrednosti od 0 do 16 i za svaku ispisuje odgovarajući faktorijel. Time se jasno vidi da Lisp, pored funkcionalnog načina razmišljanja, podržava i stil pisanja blizak imperativnim petljama.

Noviji funkcionalni jezici koriste sintaksu bližu imperativnim jezicima, ali pritom zadržavaju karakteristične mehanizme kao što su poklapanje obrazaca (*pattern matching*) i skraćenice (*comprehensions*). Time se postiže važan kompromis: spoljašnji zapis postaje pristupačniji, a suštinska funkcionalna ideja ostaje očuvana.

Poklapanje obrazaca u Haskellu može se pisati preko `case` izraza:

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

Sa uparenim obrascem vrši se vezivanje, a kada vezivanje nije potrebno koristi se wildcard _. Oblik `case` bira prvi obrazac koji odgovara vrednosti izraza `exp`, dok donja grana sa _ predstavlja podrazumevani slučaj kada prethodni obrasci nisu odgovarali.

Primer 4.3.19 Funkcija `gcd` računa najveći zajednički delilac Euklidovim algoritmom. Ako je drugi argument nula, rezultat je prvi argument; u suprotnom se poziv nastavlja rekurzivno sa parom `(y, x mod y)`.

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

Primer 4.3.20 Ista funkcija može se zapisati i kraće, tako što se sami parametri koriste kao obrasci. Takav zapis je u Haskellu često pregledniji od eksplicitnog `case` izraza.

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

Primer 4.3.21 Ova definicija koristi `case` da razlikuje praznu listu od neprazne liste. Kod neprazne liste glava `h` nije važna za rezultat, već se dužina dobija kao 1 plus dužina repa `t`.

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

Primer 4.3.22 Ista funkcija može se zapisati i obrascima direktno u zaglavlju funkcije. Takav stil je posebno prirodan kada struktura podataka određuje tok računanja.

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

Osnovni paterni za liste su:

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

Ovi paterni služe za razlaganje liste prema njenoj strukturi. Oni omogućavaju da se različiti slučajevi obrade direktno prema broju i rasporedu elemenata.

Primer 4.3.23 Funkcija sabira prvi i treći element liste. Za praznu listu vraća 0, za listu sa jednim elementom taj element, za listu sa dva elementa prvi element, a za listu sa najmanje tri elementa zbir prvog i trećeg.

```
1 prviPlusTreci :: [Integer] -> Integer
2 prviPlusTreci l = case l of
3     [] -> 0
4     [x1] -> x1
5     [x1, _] -> x1
6     x1:_:x3:_ -> x1+x3
```

Primer 4.3.24 Ista ideja može se izraziti obrascima direktno u samoj definiciji funkcije. Prednost takvog zapisa jeste u tome što se slučajevi jasno vide već na nivou potpisa svake grane.

```
1 prviPlusTreci :: [Integer] -> Integer
2 prviPlusTreci [] = 0
3 prviPlusTreci [x1] = x1
4 prviPlusTreci [x1, _] = x1
5 prviPlusTreci (x1 : _ : x3 : _) = x1 + x3
```

Skraćenice su sintaksni dodatak radi produktivnijeg programiranja i često su vezane za definisanje listi na način blizak matematičkim definicijama:

```
1 [2*x | x <- [0..], x^2 > 10]
2 [izraz | generator, filter]
```

Ovaj opšti oblik opisuje kako se gradi nova lista: generator proizvodi kandidate, a filter zadržava samo one koji zadovoljavaju uslov.

Primer 4.3.25 Primeri pokazuju da se skraćenice mogu koristiti za generisanje kvadrata, parova, filtriranih kombinacija i aritmetički izvedenih listi. Funkcija `deljiviSaTri` dodatno pokazuje da se isti mehanizam lako pretače u novu definisanu funkciju.

```
[x^2 | x <- [1..5]]
-- [1,4,9,16,25]
[(x,y) | x <- [1,2,3], y <- [4,5]]
-- [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
[(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)]
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. Može se izlagati formalno ili neformalno, ali je u oba slučaja njena uloga višestruka: omogućava programeru da razume kako se program izvršava pre njegovog pokretanja, pokazuje šta implementacija mora da obezbedi prilikom kreiranja kompajlera, doprinosi boljem razumevanju karakteristika jezika i omogućava dokazivanje svojstava programskog jezika i samih programa.

Jedno interesantno semantičko svojstvo je striktnost semantike.

Definicija 4.3.5 *Izraz je striktan ako nema vrednost kada bar jedan od njegovih operanada nema izračunatu vrednost. Izraz je nestriktan kada može imati vrednost čak i ako neki od njegovih operanada nema izračunatu vrednost.*

Kod striktne semantike prvo se izračunavaju vrednosti svih operanada, pa tek onda vrednost izraza. Kod nestriktne semantike izračunavanje operanada se odlaže

dok te vrednosti ne postanu neophodne. Ta strategija poznata je kao zadržano ili lenjo izračunavanje. Nestriktna semantika omogućava kreiranje beskonačnih struktura i izraza.

Primer 4.3.26 Od konkretnog izraza može zavisiti da li će i na osnovu koliko poznatih argumenata biti izračunata njegova vrednost. Na primer, izraz $a\&b$ može imati vrednost $\perp$ već kada a ima vrednost $\perp$, pa vrednost izraza b u tom slučaju nije presudna i ne mora biti sračunata. Međutim, od programskog jezika zavisi da li će to svojstvo implementirati i koristiti.

Primer 4.3.27 Analizirajte izraze $a||b$, $a+b$, $a-b$, $a\cdot b$ i a/b.

Prenos parametara koji podržava striktnu semantiku jeste prenos po vrednosti (*call-by-value*). Kod nestriktne semantike odlaganje izračunavanja odgovara prenosu po potrebi (*call-by-need*). Semantika je striktna ako je svaki izraz tog jezika striktan, a nije striktna ukoliko se dozvole nestriktni izrazi. Većina funkcionalnih jezika ima striktnu semantiku, kao što su Lisp, OCaml i Scala, dok Miranda i Haskell imaju nestriktnu semantiku.

Primer 4.3.28 Suma svih neparnih kvadrata brojeva manjih od 10000: izraz konstruiše beskonačnu listu prirodnih brojeva, kvadrira ih, zadržava samo neparne kvadrate manje od 10000, a zatim ih sabira. Rezultat je 166650.

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

Implementacija jezika

Funkcionalnim programiranjem definišu se izrazi čije se vrednosti evaluiraju. Time se oponašaju matematičke funkcije na visokom nivou apstrakcije, što je veoma

udaljeno od konkretnog hardvera računara na kome se program izvršava. Ipak, bez obzira na nivo apstrakcije od koga se polazi, na kraju se mora doći do asemblera i izvršivog koda. Izvršivi program uvek odgovara hardveru računara, a asembler je po prirodi imperativan. Upravo zbog toga proces kompilacije funkcionalnih jezika predstavlja složen implementacioni zadatak.

Poseban izazov je napraviti kompajler koji podržava transparentnost referenci, beskonačne strukture podataka, nestriktnu semantiku i sve ostale pomenute koncepte. Zbog izazova efikasne implementacije, funkcionalni jezici su dugo bili najčešće interpretirani jezici. SECD mašina predstavlja istorijski važan korak, iako se danas više ne koristi, dok je G-mašina osnova za implementaciju Haskell kompajlera. Za detaljniji opis G-mašine može se pogledati i `https://amelia.how/posts/the-gmachine-in-detail.html`.

Iako u funkcionalnim jezicima nema promenljivih u imperativnom smislu, podaci se u fazi izvršavanja interno ipak moraju čuvati i memorija se alocira na hipu. Zato se životni vek podataka i dalje mora rešavati na nivou implementacije. Dealokacijom memorije upravlja sakupljač otpadaka. Haskell koristi generacijski sakupljač otpadaka, zasnovan na ideji da se objekti u memoriji dele po starosti i da mladi objekti često brzo prestaju da budu korisni, pa se zato brže uklanjaju.

4.3.3 Prednosti i mane funkcionalnog programiranja

Za funkcionalne programe lakše je konstruisati matematički dokaz ispravnosti, a sam stil programiranja nameće razbijanje koda u manje delove sa jakom kohezijom i izraženom modularnošću. Testiranje je jednostavnije jer je svaka funkcija prirodan kandidat za testiranje jedinica koda (*unit test*), a funkcije ne zavise od stanja sistema, pa je lakše sintetisati test primere i proveriti očekivani izlaz. Debagovanje je takođe često jednostavnije, osim kada je u pitanju nestriktna semantika, jer su funkcije uglavnom male i jasno specijalizovane, pa se greška

brže lokalizuje. Na isti način mogu se pregledno graditi i biblioteke funkcija.

Efikasnost se dugo navodila kao mana funkcionalnog programiranja, ali danas više nije suštinski problem. Često se navodi da je Haskell na nekim problemima efikasan koliko i C. Pored toga, u literaturi se često raspravlja i o tome da li je funkcionalno programiranje lako ili teško za učenje. Stil programiranja jeste drugačiji: programi su često kratki i lakši za čitanje, ali samo ako je čitalac naučio da čita funkcionalni kod. Produktivnost programera može biti veća jer je kod kraći, ali većina programera u praksi ne gradi sisteme od nule, već održava postojeće sisteme pisane u drugim, najčešće imperativnim jezicima.

Savremeni razvoj pokazuje da se značajna sredstva ulažu u razvoj i podršku funkcionalnog stila programiranja, na primer u jezicima kao što su Scala i Kotlin, kao i u funkcionalnim konceptima unutar skript jezika. Ipak, prelazak na funkcionalno programiranje ima veliku cenu i zahteva vreme, pa se pomeranje ka programiranju koje je više u funkcionalnom stilu odvija postepeno.

4.4 Teorijske osnove — lambda račun

Definicija 4.4.1 *Lambda račun je formalni model izračunljivosti funkcija.*

Lambda račun je razvio Alonzo Church 1930. godine. Zasniva se na apstrakciji i primeni funkcija, uz korišćenje vezivanja i supstitucije. U tom modelu funkcije se tretiraju kao izrazi koji se postepeno transformišu do rešenja, pa se lambda račun može posmatrati kao formalni model definisanja algoritma.

Lambda račun daje osnove za definisanje bezimenih funkcija. Drugim rečima, funkcija se može opisati i bez posebnog imena, samo navođenjem parametara i pravila preslikavanja. Takav način zapisivanja kasnije postaje osnova za anonimne funkcije u savremenim jezicima.

Primer 4.4.1 Funkcija $sum(x, y) = x + y$ može se opisati kao preslikavanje $(x, y) \mapsto x + y$, a funkcija identiteta kao $x \mapsto x$. Lambda izraz formalizuje upravo tu ideju.

Iako lambda račun nije prvobitno razvijen sa idejom da neposredno služi programiranju, danas se često smatra prvim funkcionalnim jezikom u teorijskom smislu. Njegova ekspresivnost ekvivalentna je ekspresivnosti Tjuringovih mašina. Za razliku od modela koji polaze od mašine i njenih stanja, lambda račun naglašava pravila transformacije izraza i ne bavi se arhitekturom sistema koji ta pravila realizuje. Upravo zbog toga svi moderni funkcionalni jezici mogu da se posmatraju kao sintaksno ulepšane varijante lambda računa. U tom smislu važi i formulacija da je ekspresivnost Haskella ekvivalentna ekspresivnosti lambda računa.

Postoje netipizirani i tipizirani lambda račun. Istorijski je prvo nastao netipizirani lambda račun. Tipizirani lambda račun uvodi ograničenje da se funkcije mogu primenjivati samo na odgovarajuće tipove podataka i zato igra važnu ulogu u dizajnu sistema tipova programskih jezika.

4.4.1 Sintaksa lambda izraza

Sintaksa lambda izraza je:

$$\lambda\text{promenljiva.telo}$$

sa značenjem „promenljiva se preslikava u telo". Lambda izraz se naziva i funkcijska apstrakcija, anonimna funkcija ili bezimena funkcija.

Primer 4.4.2

$$\lambda x.x + 1 \qquad x \to x + 1$$

$$\lambda x.x \qquad x \to x$$

$$\lambda x.x \cdot x + 3x \longrightarrow x \cdot x + 3$$

Intuitivno, prvi izraz predstavlja funkciju inkrementiranja, drugi funkciju identiteta, a treći funkciju koja kvadrira argument i zatim uvećava rezultat za 3.

Lambda izraz se može primenjivati na druge izraze:

$$(\lambda\text{promenljiva.telo})\ \text{izraz}$$

što intuitivno odgovara pozivu funkcije. Na primer:

$$(\lambda x.x + 1)5, \qquad (\lambda x.x \cdot x + 3)((\lambda x.x + 1)5).$$

Validni lambda izrazi nazivaju se lambda termovi. Oni se sastoje od promenljivih, simbola apstrakcije λ, tačke i zagrada.

Definicija 4.4.2 (Induktivna definicija lambda terma) *Lambda termovi mogu se konstruisati samo konačnom primenom narednih pravila:*

- *promenljiva je lambda term;*
- *ako je t lambda term, onda je i $\lambda x.t$ lambda term;*
- *ako su t i s lambda termovi, onda je i $(t\ s)$ lambda term.*

Svaka ispravna konstrukcija u lambda računu se svodi na jedan od ova tri osnovna oblika.

Čist i primenjen lambda račun. Posebno se razlikuju čist lambda račun, koji ne uključuje konstante u definiciji, i primenjen lambda račun, koji konstante uključuje. U čistom obliku sve mora biti izraženo isključivo pomoću apstrakcije i primene, dok se u primenjenom obliku dopuštaju i dodatne konstante radi praktičnijeg rada. Zbog jednostavnosti izlaganja često se i u lambda termovima podrazumevaju numerali i aritmetičke funkcije.

Primer 4.4.3 (Churchovi brojevi.) Prirodni brojevi se mogu definisati korišćeenjem osnovne definicije

lambda računa

$$0 = \lambda f.\lambda z.z,$$
$$1 = \lambda f.\lambda z.fz,$$
$$2 = \lambda f.\lambda z.f(fz),$$
$$3 = \lambda f.\lambda z.f(f(fz)).$$

Intuitivno, 0 znači da se funkcija f ne primenjuje nijednom, 1 da se primenjuje jednom, 2 dva puta, a 3 tri puta.

U primenjenom lambda računu često se koristi i naredna jednostavna gramatika:

$\langle con \rangle$ ::= konstanta, $\langle id \rangle$ ::= identifikator,

$\langle exp \rangle ::= \langle id \rangle \mid \langle con \rangle \mid \lambda \langle id \rangle . \langle exp \rangle \mid \langle exp \rangle \langle exp \rangle \mid (\langle exp \rangle)$.

Slično, korišćenjem osnovnog lambda računa mogu se definisati i aritmetičke funkcije, ali se radi jednostavnosti u praksi u okviru lambda termova često koriste standardno imenovane operacije kao što su $+$, $-$, $*$ i slične.

Asocijativnost. Zagrade su u lambda računu suštinske, jer određuju strukturu izraza i time neposredno utiču na njegovo značenje. Da bi se njihov broj smanjio, usvajaju se sledeća pravila:

- primena funkcije je levo asocijativna,
- apstrakcija je desno asocijativna,
- niz apstrakcija može se skratiti, pa se na primer $\lambda x.\lambda y.\lambda z.e$ zapisuje kao $\lambda xyz.e$.

Primer 4.4.4 Posebno je važno uočiti da su termovi

$$\lambda x.((\lambda x.x + 1)x) \qquad \text{i} \qquad (\lambda x.(\lambda x.x + 1))x$$

različiti termovi, upravo zbog zagrada.

Primer 4.4.5 Za izraz

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

ekvivalentan izraz sa eksplicitnim zagradama glasi

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

Ovaj primer pokazuje kako pravila asocijativnosti određuju tačno mesto svake primene i svake apstrakcije.

Primer 4.4.6 Primena funkcije u Haskellu piše se bez zagrada, kao u lambda računu. Zbog toga Haskell na vrlo neposredan način odražava osnovnu sintaksnu intuiciju lambda računa.

U narednom primeri, prvi izraz primenjuje funkciju `sqrt` direktno na broj 2. Drugi pokazuje da su zagrade potrebne kada argument sam predstavlja složeniji izraz:

```
1 Prelude> sqrt 2
2 1.4142135623730951
3 Prelude> sqrt (abs (-2))
4 1.4142135623730951
```

Bez zagrada bi se drugi izraz tumačio kao primena funkcije `sqrt` na funkciju `abs`, što interpreter prijavljuje kao grešku jer se tipovi ne poklapaju.

4.4.2 Slobodne i vezane promenljive

U okviru lambda računa ne postoji koncept deklaracije promenljive u smislu poznatom iz imperativnih jezika. Umesto toga, od presudnog je značaja razlikovanje vezanih i slobodnih promenljivih. Intuitivno, slobodna promenljiva terma je promenljiva koja nije vezana odgovarajućom lambda apstrakcijom.

Definicija 4.4.3 (Induktivna definicija slobodne promenljive) *Skup slobodnih promenljivih*

- *za term x je $\{x\}$;*
- *za term $\lambda x.t$ dobija se iz skupa slobodnih promenljivih terma t uklanjanjem x;*
- *za term $(t\ s)$ je unija skupova slobodnih promenljivih izraza t i s.*

Ova definicija omogućava da se strogo razdvoji ono što je lokalno vezano unutar apstrakcije od onoga što ostaje „otvoreno“ u izrazu.

Primer 4.4.7 Term $\lambda x.x$ nema slobodnih promenljivih.

Primer 4.4.8 Term $\lambda x.x \cdot y$ ima slobodnu promenljivu y.

4.4.3 Redukcije

Za transformacije izraza koriste se izvođenja, odnosno redukcije ili konverzije. Različite vrste redukcija tradicionalno se označavaju slovima grčkog alfabeta i daju pravila po kojima se izraz prevodi iz početnog u neko jednostavnije ili konačno stanje. Upravo ta pravila čine dinamičku stranu lambda računa.

α ekvivalentnost i α redukcija

α ekvivalentnost izražava činjenicu da izbor imena vezane promenljive nije važan.

Primer 4.4.9 $\lambda x.x$ i $\lambda y.y$ su α-ekvivalentni. Intuitivno, oba izraza predstavljaju isto preslikavanje, tj. funkciju identita.

Međutim, $\lambda x.x$ i $\lambda x.y$ nisu α-ekvivalentni, jer ne

predstavljaju istu funkciju: prva je funkcija identiteta dok je druga funkcija koja promenljivu x preslikava u vrednost y (konstantna funkcija).

Primer 4.4.10 Termovi x i y nisu α-ekvivalentni, jer promenljive u njima nisu vezane lambda apstrakcijom.

Primer 4.4.11 Primeri α-ekvivalentnih i neekvivalentnih termova:

$$\lambda k.5 + k/2 \equiv_\alpha \lambda h.5 + h/2,$$

$$\lambda ij.i - j \cdot 3 \equiv_\alpha \lambda mn.m - n \cdot 3.$$

$$\lambda k.5 + k/2 \not\equiv_\alpha \lambda h.5 + h/3,$$

$$\lambda a.a \cdot y - 1 \not\equiv_\alpha \lambda b.b \cdot z - 1.$$

$$\lambda z.z \cdot y - 1 \not\equiv_\alpha \lambda x.x \cdot z - 1$$

$$\lambda ij.i - j \cdot 3 \not\equiv_\alpha \lambda nm.m - n \cdot 3$$

α redukcija ili α preimenovanje dozvoljava promenu imena vezanim promenljivama, ali samo tako da se ne naruši značenje izraza, odnosno da je dobijeni izraz α-ekvivalentan sa početnim.

Primer 4.4.12 Moguće su naredne α redukcije:

$$\lambda x.yx \to_\alpha \lambda z.yz \to_\alpha \lambda a.ya \to_\alpha \ldots$$

pri čemu promenljiva y nije vezana i zato se njeno ime ne sme menjati.

Primer 4.4.13 Izraz $\lambda x.\lambda x.x$ može se svesti na $\lambda y.\lambda x.x$, ali ne i na $\lambda y.\lambda x.y$.

Primer 4.4.14 α redukcijom ne sme se izazvati hvatanje promenljive: $\lambda x.\lambda y.x$ smemo zameniti sa $\lambda z.\lambda y.z$,

ali ne sa $\lambda y.\lambda y.y$.

Alfa preimenovanje je nekada neophodno da bi se ispravno izvršila β-redukcija, jer sprečava neželjeno hvatanje promenljivih.

δ redukcija

δ redukcija odnosi se na konstante i ugrađene operacije. Najprostiji tip lambda izraza čine konstante, koje se same po sebi više ne mogu dalje transformisati, ali nad kojima mogu delovati već poznate operacije. Na primer:

$$3 + 5 \rightarrow_\delta 8.$$

Kada je iz konteksta jasno o kojoj se redukciji radi, oznaka redukcije se može i izostaviti.

β-redukcija

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

Definicija 4.4.4 *β-redukcija formalizuje primenu funkcije:*

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

Supstitucija $[I/P]T$ je proces zamene svih slobodnih pojavljivanja promenljive P u telu izraza T izrazom I.

Primer 4.4.15 Primeri β-redukcije su:

$$\begin{aligned}(\lambda x.x + 1)5 &\rightarrow [5/x](x + 1) = 5 + 1\\ &\rightarrow_\delta 6\end{aligned}$$

$$(\lambda x.x \cdot x + 3)((\lambda x.x + 1)5) \rightarrow_\beta [6/x](x \cdot x + 3) = 6 \cdot 6 + 3$$
$$\rightarrow_\delta 39$$

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

$$(\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$$

U jednostavnim primerima postupak supstitucije izgleda gotovo mehanički, ali u opštem slučaju, na primer kod izraza $(\lambda x.x(\lambda x.x))(\lambda x.x\ x)$, potrebno je sasvim precizno definisati zamenu i po potrebi prethodno izvršiti α preimenovanje da bi se izbegla kolizija imena.

Definicija 4.4.5 *Neka su x i y promenljive, a M i N lambda izrazi. Tada se supstitucija definiše na sledeći način:*

- *Supstitucija za promenljive*
  - $[N/x]x = N$,
  - $[N/x]y = y$ *ako je* $x \neq y$
- *Supstitucija za apstrakciju*
  - $[N/x](\lambda x.M) = \lambda x.M$
  - $[N/x](\lambda y.M) = \lambda y.([N/x]M)$ *ako je* $x \neq y$ *i* y *ne pripada skupu slobodnih promenljivih izraza* N, *i*
- *Supstitucija za primenu*
  $[N/x](M_1M_2) = ([N/x]M_1)([N/x]M_2)$.

Zamena se vrši samo nad slobodnim pojavljivanjima promenljive i da se, kada je to neophodno, najpre primeni α preimenovanje kako bi se izbegla kolizija imena. Bez tog uslova formalna pravila bi mogla promeniti smisao izraza koji se redukuje.

Višestruka β-redukcija znači da redukciju primenjujemo sve dok je moguće. To odgovara izračunavanju vrednosti funkcije.

η redukcija

Definicija 4.4.6 *η redukcija glasi*

$$\lambda x.f\ x \rightarrow_\eta f,$$

pod uslovom da se x ne javlja kao slobodna promenljiva u izrazu f.

Ova redukcija izražava ideju funkcijskog proširenja: funkcija $\lambda x.f\ x$ i funkcija f imaju isto spoljašnje ponašanje (kada se x ne pojavljuje slobodno u izrazu f). Naime, oba izraza primenjena na y daju $f\ y$

4.4.4 Funkcije višeg reda i funkcije sa više argumenata

Lambda račun prirodno podržava funkcije višeg reda. Funkcija može biti i argument i povratnu vrednost. To nije dodatna mogućnost, već neposredna posledica činjenice da su funkcije i same izrazi nad kojima se može dalje računati.

Primer 4.4.16 Naredni lamda izraz

$$\lambda x.(x\ 2) + 1$$

prima funkciju kao argument. Na primer, ako se primeni na izraz $\lambda x.x + 1$ postaje $\lambda x.((\lambda x.x + 1)2) + 1$:

$$(\lambda x.(x\ 2) + 1)(\lambda x.x + 1) \rightarrow (\lambda x.x + 1)2 + 1 \rightarrow 4$$

Slično, možemo je primeniti na funkciju identiteta:

$$(\lambda x.(x\ 2) + 1)(\lambda x.x) \rightarrow ((\lambda x.x)\ 2) + 1 \rightarrow 3$$

U prvom slučaju prosleđena funkcija uvećava broj 2 za 1, pa se zatim na rezultat dodaje još 1. U drugom slučaju koristi se identitet, pa je krajnji rezultat 2 + 1.

Primer 4.4.17 Funkcija koja vraća funkciju u svom telu sadrži drugi lambda izraz. Na primer, naredni lambda izraz

$$\lambda x.(\lambda y.2 \cdot y + x)$$

vraća funkciju kao rezultat. Pri primeni izraza na broj 5 dobija se nova funkcija $\lambda y.2 \cdot y + 5$.

Zbog desne asocijativnosti isti izraz može se kraće zapisati kao $\lambda xy.2 \cdot y + x$.

Lambda izrazi su ograničeni na jedan argument, ali bilo koja funkcija sa više argumenata može se definisati pomoću funkcije sa po jednim argumentom. Taj rezultat poznat je još od 1924. godine. Ovaj postupak naziva se Karijev postupak. Njegova osnovna ideja je da funkcija koja treba da uzme dva argumenta najpre uzme jedan argument, a zatim od njega izgradi novu funkciju koja očekuje drugi argument. U opštem slučaju, funkcija oblika $f(x_1, x_2, \ldots, x_n) =$ telo u lambda računu zapisuje se kao $\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.\text{telo})))$, odnosno skraćeno kao $\lambda x_1 x_2 \ldots x_n.\text{telo}$.

Primer 4.4.18

$$f(x, y) = x + y \qquad \rightsquigarrow \qquad \lambda xy.x + y$$

$$((\lambda xy.x + y)2)6 \rightarrow (\lambda y.2 + y)6 \rightarrow 8$$

Primer 4.4.19 Karijev postupak u praksi se jasno vidi na sledećem primeru. Izraz `(max 3)` predstavlja novu funkciju koja očekuje još jedan argument, pa su zapisi `(max 3) 10` i `max 3 10` ekvivalentni.

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

Primer 4.4.20 Delimična evaluacija Karijeve funkcije posebno je pregledna u sledećem primeru. Najpre se fiksira prvi argument, zatim i drugi, pa se dobija nova funkcija koja čeka još samo poslednji argument.

```
1  pomnozi :: Int -> Int -> Int -> Int
2  pomnozi i j k = i*j*k
3
4  let p3  = pomnozi 3
5  let p34 = p3 4
6  *Main> :t p3
7  p3 :: Int -> Int -> Int
8  *Main> :t p34
9  p34 :: Int -> Int
10 *Main> p34 2
11 24
12 *Main> p34 5
13 60
```

Karijeve funkcije zato uzimaju argumente jedan po jedan i veoma su pogodne za delimičnu evaluaciju, pri čemu se fiksiraju levi argumenti funkcije.

4.4.5 Normalni oblik

Definicija 4.4.7 *Lambda izraz je u normalnom obliku ako se na njega više ne može primeniti nijedna redukcija.*

Primer 4.4.21 Izvesti normalni oblik, ukoliko on postoji ili pokazati da ne postoji normalni oblik:

$$\begin{aligned}(\lambda x.x+3)((\lambda x.x+5)4) &\to (\lambda x.x+3)(4+5)\\ &\to (\lambda x.x+3)9\\ &\to (9+3)\\ &\to 12\end{aligned}$$

$$\begin{aligned}(\lambda x.x\,x)(\lambda y.y) &\to_\beta (\lambda y.y)(\lambda y.y)\\ &\to_\alpha (\lambda x.x)(\lambda y.y)\\ &\to_\beta \lambda y.y\end{aligned}$$

$$
\begin{aligned}
(\lambda x.x(\lambda x.x))y &\to_\alpha (\lambda x_0.x_0(\lambda x.x))y \\
&\to_\beta y(\lambda x.x)
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.x(\lambda x.x))(\lambda x.x\,x) &\to (\lambda x.x(\lambda x_0.x_0))(\lambda x_{00}.x_{00}x_{00}) \\
&\to (\lambda x_{00}.x_{00}x_{00})(\lambda x_0.x_0) \\
&\to (\lambda x_0.x_0)(\lambda x_0.x_0) \\
&\to (\lambda x_0.x_0)(\lambda x_{000}.x_{000}) \\
&\to (\lambda x_{000}.x_{000}) \\
&\to (\lambda x.x)
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.\lambda y.yx)(\lambda z.u) &\to (\lambda x.(\lambda y.yx))(\lambda z.u) \\
&\to \lambda y.y(\lambda z.u)
\end{aligned}
$$

$$
\begin{aligned}
(\lambda k.k+1)((\lambda m.m-1)2) &\to (\lambda k.k+1)(2-1) \\
&\to (\lambda k.k+1)1 \\
&\to (1+1) \\
&\to 2
\end{aligned}
$$

$$
\begin{aligned}
(\lambda k.k(k\,4))(\lambda y.y-2) &\to ((\lambda y.y-2)((\lambda y.y-2)4)) \\
&\to ((\lambda y.y-2)(4-2)) \\
&\to (\lambda y.y-2)2 \to 2-2 \to 0
\end{aligned}
$$

$$
\begin{aligned}
((\lambda kmn.k-m+n)10)5 &\to ((\lambda k.\lambda mn.k-m+n)10)5 \\
&\to ((\lambda k.(\lambda mn.k-m+n))10)5 \\
&\to (\lambda mn.10-m+n)5 \\
&\to (\lambda m.(\lambda n.10-m+n))5 \\
&\to (\lambda n.10-5+n) \\
&\to (\lambda n.5+n)
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.x\,x)(\lambda x.x\,x) &\to (\lambda x_0.x_0x_0)(\lambda x.x\,x) \\
&\to (\lambda x.x\,x)(\lambda x.x\,x) \\
&\to \cdots
\end{aligned}
$$

Poslednji primer pokazuje izraz koji nema normalni

oblik.

Primer 4.4.22 Izvesti normalni oblik za naredne izraze:

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

Nemaju svi izrazi normalni oblik. Takođe, za neke izraze mogu postojati različite mogućnosti primene redukcija, posebno β-redukcije. Normalni oblik intuitivno odgovara vrednosti polaznog izraza, ali ta vrednost ne mora biti broj; to može biti i neki drugi lambda izraz na koji se više ne može primeniti redukcija.

Primer 4.4.23

$$(\lambda x.5 \cdot x)((\lambda x.x + 1)2)$$

može se redukovati na više načina:

$$(\lambda x.5 \cdot x)(2 + 1) \qquad \text{ili} \qquad 5 \cdot ((\lambda x.x + 1)2).$$

Primer 4.4.24 Još jedan karakterističan primer različitih puteva redukcije jeste:

$$(\lambda y.y\,a)((\lambda x.x)(\lambda z.(\lambda u.u)\,z)).$$

Prvi korak može voditi ka izrazu

$$(\lambda y.y\,a)(\lambda z.(\lambda u.u)\,z)$$

ili ka izrazu

$$((\lambda x.x)(\lambda z.(\lambda u.u)\,z))a,$$

ili ka izrazu

$$(\lambda y.y\,a)((\lambda x.x)(\lambda z.z)).$$

Ovaj primer motiviše pitanje da li različiti putevi redukcije vode ka istom rezultatu.

Teorema 4.4.1 (Svojstvo konfluentnosti, Church–Rosser) *Ako se lambda izraz može svesti na dva različita lambda izraza M i N, onda postoji treći izraz Z do koga se može doći i iz M i iz N.*

Posledica teoreme je da svaki lambda izraz ima najviše jedan normalni oblik, ako normalni oblik postoji. Drugim rečima, iako putevi redukcije mogu biti različiti, konačni rezultat, ukoliko postoji, ne može biti proizvoljan.

Poredak izvođenja redukcija

Aplikativni poredak odgovara pozivu po vrednosti (*call-by-value*): najpre se izračuna argument, pa se tek onda šalje funkciji.

Normalni poredak znači da se β-redukcijom uvek redukuje najlevlji izraz. On odgovara evaluaciji po imenu (*call-by-name*), odnosno evaluaciji po potrebi (*call-by-need*).

Razlika između aplikativnog i normalnog poretka nije samo tehnički detalj implementacije, već ima neposredne posledice po efikasnost i po samu mogućnost završetka izračunavanja.

Primer 4.4.25 Primer primene aplikativnog poretka:

$$(\lambda x.5 \cdot x)(2+1) \rightarrow (\lambda x.5 \cdot x)3 \rightarrow 5 \cdot 3 \rightarrow 15$$

Primer primene normalnog poretka:

$$(\lambda x.5 \cdot x)(2+1) \rightarrow 5 \cdot (2+1) \rightarrow 5 \cdot 3 \rightarrow 15$$

Teorema 4.4.2 (Teorema standardizacije) *Ako je Z normalni oblik izraza E, onda postoji niz redukcija u normalnom poretku koji vodi od E do Z.*

Ova teorema govori da ako normalni oblik lambda izraza postoji i ako primenjujemo redukcije u normalnom poretku, onda će nas taj niz redukcija dovesti do normalnog oblika.

Normalnim poretkom redukcija ostvaruje se lenja evaluacija: izrazi se evaluiraju samo ukoliko su potrebni. Time se izbegavaju nepotrebna izračunavanja i odlaže se rad nad onim delovima izraza čija vrednost možda nikada neće postati potrebna. Normalni poredak, odnosno lenja evaluacija, garantuje završetak izračunavanja uvek kada je to moguće.

Primer 4.4.26 Primena aplikativnog poretka:

$$(\lambda x.1)(12345 \cdot 54321) \rightarrow (\lambda x.1)670592745 \rightarrow 1$$

Primena normalnog poretka:

$$(\lambda x.1)(12345 \cdot 54321) \rightarrow 1$$

Normalnim poretkom preskače se nepotrebno izračunavanje.

Primer 4.4.27 Primena aplikativnog poretka:

$$(\lambda x.1)((\lambda x.xx)(\lambda x.xx)) \rightarrow \cdots$$

Primena normalnog poretka:

$$(\lambda x.1)((\lambda x.xx)(\lambda x.xx)) \rightarrow 1$$

Normalni poredak nas vodi ka normalnom izrazu, dok aplikativnim poretkom ne možemo da dođemo do normalnog oblika, iako on postoji.

Normalni poredak, odnosno lenja evaluacija, odgovara nestriktnoj semantici i omogućava korišćenje beskonačnih struktura.

Primer 4.4.28 Sledeći izraz koristi beskonačnu listu

`[1..]`, ali se računa samo onaj njen početni deo koji je potreban da bi se dobio konačan rezultat.

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

Upravo se na ovakvim primerima najjasnije vidi praktična vrednost lenje evaluacije: beskonačna struktura ne mora biti prepreka ako je potrošnja te strukture konačna i kontrolisana.

Da se izračunavanja ne bi ponavljala, kompajleri koriste tehnike redukcije grafova. Na primer, za izraz

$$(\lambda x.x + x)(12345 \cdot 54321)$$

ne treba dva puta nezavisno računati isti proizvod. Umesto toga, rezultat se izračuna jednom i zatim deli na više mesta gde je potreban. Time lenja evaluacija ne ostaje samo teorijski elegantna, već postaje i implementaciono efikasna.

Rezime

-
-
-
-

Korisni izvori za dalje proučavanje su:

- https://www.haskell.org/
- https://www.haskell.org/documentation
- Real World Haskell, http://book.realworldhaskell.org/
- http://learnxinyminutes.com/docs/haskell/
- http://poincare.matf.bg.ac.rs/~nenad/fp/lambda%20racun%20i%20kombinatori.ps
- materijali o lambda računu sa nastavnih stranica kursa

Pitanja

1. Na koji način je John Backus uticao na razvoj funkcionalnih jezika?
2. Koji su najpoznatiji funkcionalni programski jezici?
3. Koji je domen upotrebe funkcionalnih programskih jezika?
4. Koje su osnovne karakteristike funkcionalnih programskih jezika?
5. Šta je svojstvo transparentnosti referenci i na koji način ovo svojstvo utiče na razumevanje i transformaciju programa?
6. Koje su osobine programa u kojima se poštuje pravilo transparentnosti referenci?
7. Da li je moguće u potpunosti zadržati svojstvo transparentnost referenci?
8. Koji je odnos transparentnosti referenci sa bočnim efektima?
9. Da li je moguće obezbediti promenu stanja programa i istovremeno zadržati svojstvo transparentnosti referenci?
10. Šta su funkcionalni jezici? Šta su čisti funkcionalni jezici?
11. Navesti primere čisto funkcionalnih jezika.
12. Koje su osnovne aktivnosti u okviru funkcionalnog programiranja?
13. Kako izgleda program napisan u funkcionalnom programskom jeziku?
14. Šta je potrebno da obezbedi funkcionalni programski jezik za uspešno programiranje?
15. Šta je striktna, a šta nestriktna semantika?
16. Kakvu semantiku ima jezik Haskell?
17. Kakvu semantiku ima jezik Lisp?
18. Koje su prednosti funkcionalnog programiranja?
19. Koje su mane funkcionalnog programiranja?
20. Šta uključuje definisanje funkcije?
21. Šta su funkcije višeg reda? Navesti primere.
22. Da li matematičke funkcije imaju propratne efekte?
23. Koji je formalni okvir funkcionalnog programiranja?

24. Koji se jezik smatra prvim funkcionalnim jezikom?
25. Koja je ekspresivnost lambda računa?
26. Koji su sve sinonimi za lambda izraz?
27. Navesti definiciju lambda terma.
28. Da li čist lambda račun uključuje konstante u definiciji?
29. Navesti primer jednog lambda izraza, objasniti njegovo značenje i primeniti dati izraz na neku konkretnu vrednost.
30. Koja je asocijativnost primene a koja apstrakcije?
31. Navesti ekvivalentan izraz sa zagradama za izraz koji koristi više primena i apstrakcija.
32. Koje su slobodne a koje vezane promenljive u izrazu po izboru?
33. Navesti definiciju slobodne promenljive. Koje promenljive su vezane?
34. Koja je uloga pojma alfa ekvivalentnosti?
35. Šta su redukcije?
36. Šta je delta redukcija? Navesti primer.
37. Šta je alfa redukcija? Navesti primer.
38. Kada se koristi alfa redukcija?
39. Šta je beta redukcija? Navesti primer.
40. Definisati supstituciju.
41. Navesti primer lambda izraza koji definiše funkciju višeg reda koja prima funkciju kao argument.
42. Navesti primer lambda izraza koji definiše funkciju višeg reda koja ima funkciju kao povratnu vrednost.
43. Čemu služi Karijev postupak?
44. Kako se definišu funkcije sa više argumenata?
45. Šta je normalni oblik funkcije?
46. Da li svi izrazi imaju svoj normalni oblik?
47. Navesti svojstvo konfluentnosti.
48. Da li izraz može imati više normalnih oblika?
49. Koja je razlika između aplikativnog i normalnog poretka?
50. Šta govori teorema standardizacije?
51. Šta se dobija lenjom evaluacijom?
52. Koje su osnovne karakteristike Haskella?
53. Šta izračunava dati Haskell program?

Logičko programiranje 5

Pregled

- ►
- ►
- ►
- ►

5.1 ...

5.2 ...

Rezime

- ►
- ►
- ►
- ►

Pitanja

1.
2.
3.
4.
5.

Skript programiranje 6

Pregled

-
-
-
-

Skript programiranje predstavlja značajan stil programiranja koji nalazi primenu u velikom broju različitih domena. Prema aktuelnim pokazateljima kao što je TIOBE indeks, približno trećina najpopularnijih programskih jezika pripada skript paradigmi. Posebno se izdvaja Python kao skript jezik opšte namene čija popularnost kontinuirano raste, kako u industriji, tako i u akademskom okruženju.

6.1 Uloga skript programiranja

Skript jezik se može definisati kao programski jezik namenjen pisanju skriptova, gde skript predstavlja sekvencu komandi koje se izvršavaju u odgovarajućem izvršnom okruženju (engl. *run-time environment*). Skriptovi mogu biti izvršavani bez interakcije sa korisnikom (na primer, skriptovi koji se pišu za servere i zvršavaju na serverima), ali i interaktivno, na primer u okviru REPL okruženja (engl. *read-eval-print-loop*).

Za većinu skript jezika karakteristično je da se interpretiraju, što omogućava fleksibilnije izvršavanje, odnosno izvršavanje kojme ne prethodi faza kompilacije i izgradnje izvršive datoteke. Takođe, moderni skript jezici su često jednostavni za učenje i upotrebu, čak i za korisnike koji nisu profesionalni programeri, iako postoje i stariji skript jezici sa znatno složenijom sintaksom.

6.1.1 Razvoj skript jezika

Prvi skript jezici nastali su u okviru operativnih sistema, pri čemu se kao začetnik često navodi komandni jezik `sh` (Unix shell). Ovaj jezik je inicijalno predstavljao skup komandi koje su se interpretirale kao pozivi sistemskih funkcija, poput upravljanja fajlovima ili filtriranja sadržaja. Vremenom su u takve jezike uvedeni dodatni mehanizmi, uključujući promenljive, kontrolu toka izvršavanja i funkcije, čime su evoluirali u potpune programske jezike. U skladu sa tim razvojem, skript jezici su korišćeni za zapisivanje niza komandi u fajlovima, odnosno skriptova, koji se zatim interpretiraju i izvršavaju.

U savremenom razvoju softvera skript jezici beleže značajnu ekspanziju. Veliki deo inovacija u oblasti programskih jezika tokom poslednjih decenija odnosi se upravo na skript paradigmu. Njihova široka primena obuhvata različite domene, pri čemu posebnu ulogu imaju u veb programiranju, gde se koriste jezici kao što su PHP, JavaScript i Perl. Iako su prvobitno nastali kao komandni jezici operativnih sistema (npr. `bash`), danas skript jezici imaju znatno širi spektar primena. Oni mogu biti specijalizovani za određene domene, ali i univerzalni jezici opšte namene, poput Pythona. Zbog svoje česte upotrebe u povezivanju različitih softverskih komponenti, skript jezici se često nazivaju i *glue languages*.

6.1.2 Povezivanje aplikacija

Tradicionalni programski jezici, kao što su C/C++ i Java, primarno su namenjeni razvoju samostalnih aplikacija koje obrađuju ulazne podatke i na osnovu njih generišu odgovarajući izlaz. Takav pristup podrazumeva jasno definisanu strukturu programa i fokus na efikasno izvršavanje pojedinačnih zadataka.

Međutim, savremena upotreba računara često prevazilazi okvire jedne izolovane aplikacije i zahteva koordinaciju rada većeg broja različitih programa. Ovakvi zadaci

podrazumevaju manipulaciju podacima kroz više alata, njihovu međusobnu integraciju, kao i automatizaciju složenih radnih tokova. Ručno izvršavanje ovih aktivnosti je vremenski zahtevno i sklono greškama, zbog čega se u praksi najčešće koriste skript jezici, koji omogućavaju efikasno povezivanje i orkestraciju različitih softverskih komponenti.

Iako je koordinaciju između programa moguće ostvariti i korišćenjem tradicionalnih programskih jezika, takav pristup često zahteva značajno više napora, dodatnu infrastrukturu i detaljno upravljanje resursima, što ga čini manje pogodnim za brzu automatizaciju i integraciju.

Primer 6.1.1 Fotograf može imati potrebu da preuzme slike sa digitalnog uređaja, konvertuje ih u odgovarajući format, izvrši rotaciju, generiše umanjene verzije za pregled, organizuje ih prema vremenu ili tematici, napravi rezervne kopije na udaljenim sistemima i na kraju ponovo inicijalizuje memoriju uređaja. Za svaki od ovih zadataka postoji odgovarajuća podrška, ali je potrebno sve te zadatke automatizovati što se jednostavno može uraditi pisanjem skriptova.

Primer 6.1.2 Potreba za integracijom različitih poslova postoji i u informacionim sistemima za obračun plata, gde je neophodno objediniti podatke iz različitih izvora, uključujući elektronske kartice, papirne izveštaje i manuelne unose, izvršiti obradu nad bazama podataka, poštovati pravne regulative, izračunati poreze i doprinose, kao i generisati odgovarajuću dokumentaciju.

Primer 6.1.3 U kontekstu razvoja dinamičkih veb aplikacija, neophodno je uskladiti procese autentikacije i autorizacije, komunikaciju sa udaljenim servisima, obradu multimedijalnih sadržaja i generisanje HTML dokumenata.

6.1.3 Poređenje tradicionalnih i skript jezika

Tradicionalni programski jezici, poput C/C++ i Jave, dizajnirani su sa ciljem postizanja visoke efikasnosti izvršavanja, dobre održivosti koda, portabilnosti između različitih platformi i mogućnosti statičkog otkrivanja grešaka u ranim fazama razvoja. Njihovi sistemi tipova zasnovani su na primitivnim konceptima, kao što su celobrojne vrednosti fiksne veličine, brojevi u pokretnom zarezu, karakteri i nizovi, što omogućava preciznu kontrolu nad memorijom i performansama.

Nasuprot tome, skript jezici stavljaju naglasak na fleksibilnost i brzinu razvoja, omogućavajući programerima da se brzo prilagode specifičnim zahtevima problema. Oni podržavaju dinamičke provere tokom izvršavanja i često nude bogat skup ugrađenih struktura podataka i apstrakcija višeg nivoa, kao što su liste, tabele, mape, datoteke i drugi složeni objekti. Ovakav pristup omogućava jednostavniju manipulaciju podacima i efikasniju integraciju različitih sistema, što skript jezike čini posebno pogodnim za automatizaciju i povezivanje aplikacija.

6.2 Karakteristike skript jezika

Skript jezici predstavljaju specifičnu kombinaciju više programskih paradigmi. U njihovoj osnovi često se nalaze elementi imperativne paradigme, kao što su promenljive, naredbe, petlje i uslovne strukture, ali i koncepti objektno-orijentisanog programiranja, uključujući klase, objekte i nasleđivanje. Pored toga, mnogi skript jezici podržavaju i funkcionalne koncepte, kao što su lambda izrazi i funkcije višeg reda. Kroz bogate standardne biblioteke i dodatne module, podrška se može proširiti i na druge paradigme, što dodatno povećava njihovu fleksibilnost.

Iako granica između skript jezika i drugih programskih jezika nije uvek jasno definisana, skript jezici poseduju skup karakteristika koje ih izdvajaju kao posebnu kategoriju. U ovu grupu spadaju jezici kao što su Unix Shell

(sh), Bash, PowerShell, JavaScript, TypeScript, PHP, Perl, Python, XSLT, Tcl, VBScript, Lua i Ruby.

Osnovne karakteristike skript jezika uključuju interaktivno korišćenje i podršku za serijsku obradu, skraćen i ekspresivan zapis, fleksibilna pravila deklaracija i dosega, dinamičko tipiziranje, bogat skup sistemskih funkcija, razvijene mehanizme za manipulaciju stringovima i poklapanje obrazaca, kao i podršku za tipove podataka visokog nivoa.

6.2.1 Interaktivno korišćenje, serijska obrada i skraćeni zapis

Jedna od ključnih odlika skript jezika jeste mogućnost interaktivnog korišćenja, najčešće kroz REPL (*Read-Eval-Print Loop*) okruženje, koje omogućava neposredno izvršavanje komandi i dobijanje rezultata. Ipak, postoje i skript jezici koji su prvenstveno namenjeni za serijsku obradu bez direktne interakcije sa korisnikom, kao što su skriptovi koji se izvršavaju na serverskoj strani.

Većina skript jezika se interpretira, što znači da se program obrađuje liniju po liniju. Ovakav pristup omogućava da se izvršavanje nastavi sve do mesta na kojem se pojavi prva sintaksna greška, za razliku od kompiliranih jezika kod kojih je neophodno da ceo program bude sintaksno ispravan pre nego što se započne njegovo izvršavanje.

Postoje i izuzeci od ovog pravila. Na primer, Perl koristi hibridni pristup u kojem se skript najpre prevodi u međukod, koji se zatim interpretira. Ove dve faze su nerazdvojive i izvršavaju se zajedno pri svakom pokretanju programa, bez čuvanja međukoda u posebnoj datoteci.

Skraćen zapis predstavlja još jednu važnu karakteristiku skript jezika i direktno je povezan sa potrebom za brzim razvojem i jednostavnom upotrebom. U poređenju sa tradicionalnim programskim jezicima, skript jezici često

izbegavaju opširne deklaracije i dodatne strukturne elemente, čime omogućavaju pisanje kraćih i preglednijih programa.

Primer 6.2.1 Jednostavan program za ispis poruke na ekran u jeziku Python:

```
print("Hello, world\n")
```

Nasuprot tome, ekvivalentan program u programskom jeziku Java zahteva znatno više koda i strukture:

```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Ovaj primer ilustruje kako skript jezici omogućavaju bržu realizaciju jednostavnih zadataka.

6.2.2 Deklaracije, pravila dosega i dinamičko tipiziranje

U skript jezicima promenljive se najčešće ne deklarišu eksplicitno, već se uvode samim dodeljivanjem vrednosti. Ovakav pristup pojednostavljuje pisanje programa i doprinosi brzini razvoja, ali istovremeno zahteva pažljiviji pristup organizaciji koda.

Pravila dosega (vidljivosti promenljivih) u skript jezicima su uglavnom jednostavnija nego u tradicionalnim programskim jezicima. Na primer, u nekim jezicima, kao što je Perl, promenljive su podrazumevano globalne, iako je moguće ograničiti njihov doseg. U drugim jezicima, kao što su PHP i Tcl, promenljive su podrazumevano lokalne, ali se po potrebi mogu eksplicitno deklarisati kao globalne. U jeziku Python, promenljive su lokalne za blok u kojem im je dodeljena vrednost, uz jasno definisana pravila za pristup promenljivama iz spoljašnjih opsega.

Za razliku od toga, pravila dosega u tradicionalnim programskim jezicima često su znatno složenija i obuhvataju različite nivoe vidljivosti, uključujući blokovski,

funkcijski, klasni i modulni opseg, kao i dodatne mehanizme poput prostora imena.

Jedna od ključnih karakteristika skript jezika jeste dinamičko određivanje tipova promenljivih. To znači da se tip promenljive ne definiše unapred, već se određuje u toku izvršavanja programa, na osnovu vrednosti koju promenljiva sadrži. Ovaj pristup se često opisuje principom poznatim kao „ako nešto izgleda kao patka, pliva kao patka i oglašava se kao patka, onda je verovatno patka“, što ilustruje fleksibilnost u tretiranju tipova podataka.

U pojedinim jezicima, kao što su PHP, Ruby i Python, tipovi se proveravaju neposredno pre upotrebe, dok u drugim jezicima, kao što su Perl i Tcl, ista vrednost može biti interpretirana na različite načine u zavisnosti od konteksta u kojem se koristi.

Primer 6.2.2 U jeziku Perl, ista vrednost može biti interpretirana na različite načine u zavisnosti od konteksta u kojem se koristi:

```
1 $a = "4";
2 print $a . 3 . "\n";    # nadovezivanje: 43
3 print $a + 3 . "\n";    # sabiranje: 7
```

Dinamičko određivanje tipova posebno dolazi do izražaja kada se skript jezici koriste za povezivanje različitih aplikacija. U takvim scenarijima, skript mora biti sposoban da prihvati, transformiše i prosledi podatke između sistema koji mogu biti implementirani u različitim programskim jezicima i koristiti različite modele tipova podataka. Skriptovi često obrađuju podatke iz raznovrsnih izvora, kao što su korisničke forme, baze podataka, tabele ili veb stranice, zbog čega bi strogo statičko tipiziranje bilo previše ograničavajuće.

Ipak, dinamičko tipiziranje ima i određene nedostatke. Iako eliminiše potrebu za deklaracijama i pojednostavljuje sistem tipova, ono može smanjiti čitljivost programa i otežati razumevanje koda. Takođe, greške povezane sa tipovima mogu se detektovati tek u fazi izvršavanja, a neke od njih mogu ostati neprimećene ukoliko ne

izazivaju neposredan prekid rada programa, već samo suptilne promene u ponašanju.

Sa druge strane, statičko određivanje tipova omogućava kompajleru da unapred identifikuje određene klase grešaka, čime se povećava pouzdanost programa. Uprkos tome, skript jezici zadržavaju dinamičko tipiziranje zbog njegove fleksibilnosti i pogodnosti za brzi razvoj.

Mnogi skriptovi se ne koriste samo jednokratno, već mogu postati deo većih sistema i koristiti se duži vremenski period. U takvim slučajevima, održavanje koda postaje značajan faktor. Da bi skript bio lak za održavanje, potrebno je da bude čitljiv, dobro dokumentovan i modularno organizovan. Dizajneri modernih skript jezika, kao što je Python, nastoje da pomire zahteve za fleksibilnošću i pouzdanošću, iako to često podrazumeva kompromis između jednostavnosti i sigurnosti u radu sa tipovima.

6.2.3 Sistemske funkcije, stringovi i poklapanje obrazaca

Skript jezici obično obezbeđuju direktan i jednostavan pristup funkcionalnostima operativnog sistema. To uključuje rad sa ulazno-izlaznim tokovima, manipulaciju datotekama i direktorijumima, upravljanje procesima, komunikaciju sa bazama podataka, rad sa mrežnim protokolima, kao i mehanizme za interprocesnu komunikaciju, sinhronizaciju i kontrolu pristupa. Iako su ove mogućnosti dostupne i u drugim programskim jezicima, skript jezici ih nude na znatno jednostavniji i pristupačniji način, što ih čini pogodnim za automatizaciju sistemskih zadataka.

Skript jezici su nastali iz jezika koji su namenjeni obradi teksta i generisanju izveštaja. Zbog toga poseduju razvijene mehanizme za manipulaciju stringovima, pretragu i poklapanje obrazaca. Ove funkcionalnosti se najčešće zasnivaju na regularnim izrazima, koji omogućavaju efikasnu obradu složenih tekstualnih struktura.

Primer 6.2.3 Koju osobinu proverava naredna funkcija (Python)?

```
def osobina(n):
 return not re.match(r'^.?$|^(..+?)\1+$', '1'*n)
```

6.2.4 Tipovi podataka visokog nivoa

Pored osnovnih tipova podataka, kao što su celobrojne i realne vrednosti ili stringovi, skript jezici u okviru svoje sintakse i semantike obično pružaju direktnu podršku za tipove podataka višeg nivoa. U takve tipove spadaju skupovi, multiskupovi, rečnici (mape), liste i torke. Ovi tipovi su integrisani u sam jezik i mogu se koristiti na isti način kao i osnovni tipovi, bez potrebe za dodatnim implementacijama.

Nasuprot tome, u tradicionalnim programskim jezicima često je neophodno eksplicitno implementirati ili uključiti biblioteke koje obezbeđuju napredne strukture podataka. Takve strukture se ne koriste uvek ravnopravno sa osnovnim tipovima, što može otežati njihovu primenu. Izuzetak predstavlja programski jezik C++, koji omogućava predefinisanje operatora i time prirodniju integraciju složenih tipova podataka.

Dodatna važna karakteristika skript jezika jeste korišćenje automatskog upravljanja memorijom putem sakupljača otpadaka (engl. *garbage collector*). Ovaj mehanizam oslobađa programera potrebe da ručno upravlja memorijom, čime se smanjuje verovatnoća pojave grešaka, kao što su curenje memorije ili neispravno oslobađanje resursa.

6.3 Domeni upotrebe skript jezika

Skript jezici se koriste u velikom broju različitih domena, zahvaljujući svojoj fleksibilnosti i sposobnosti brzog razvoja. Njihova primena obuhvata komandne jezike, obradu teksta, matematiku i statistiku, razvoj jezika

proširenja, izradu veb aplikacija, kao i opštu programsku namenu.

6.3.1 Komandni jezici

Komandni jezici (engl. *shell languages*), kao što su `sh`, `csh`, `ksh`, `bash` i PowerShell, omogućavaju direktnu komunikaciju sa operativnim sistemom. Njihova osnovna namena je manipulacija fajlovima, upravljanje procesima, prosleđivanje argumenata komandama i povezivanje različitih aplikacija u jedinstvene radne tokove.

Komande u shell jezicima najčešće imaju oblik niza reči, gde prva reč predstavlja ime komande, a preostale reči njene argumente. Većina komandi odgovara izvršnim programima koji se nalaze u direktorijumima definisanim putanjom shell-a, dok postoje i ugrađene komande koje shell izvršava direktno, bez pokretanja spoljnog programa. Primeri često korišćenih komandi u Unix-olikim sistemima uključuju `cd`, `ls`, `cp`, `mv` i `mkdir`.

Shell jezici podržavaju osnovne kontrolne strukture, kao što su petlje, uslovne naredbe i funkcije, slično imperativnim programskim jezicima. Pored toga, oni nude mehanizme za ekspanziju imena fajlova i promenljivih.

Primer 6.3.1 Izraz `*.pdf` odgovara svim fajlovima sa ekstenzijom `.pdf`. Pored operatora `*`, koriste se i drugi obrasci, kao što su `?` za jedan proizvoljan karakter, `[0-9]` za opseg karaktera ili konstrukcije poput `{eps,pdf}` za izbor između više mogućnosti.

Primer 6.3.2 Automatizacija zadataka u shell jezicima često uključuje iteraciju kroz skup fajlova. Na primer, sledeći skript konvertuje sve fajlove u `.eps` formatu u `.pdf`:

```
1 for fig in *.eps
2 do
3    ps2pdf $fig
4 done
```

Isti postupak može se zapisati i u jednoj liniji:

```
for fig in *.eps; do ps2pdf $fig; done
```

Primer 6.3.3 Važan aspekt rada sa stringovima u shell jezicima jeste razlika između jednostrukih i dvostrukih navodnika. Sledeći primer ilustruje njihovo ponašanje:

```
foo=bar
single='$foo'
double="$foo"
echo $single $double
```

Rezultat izvršavanja ovog koda biće:

```
$foo bar
```

Jednostruki navodnici onemogućavaju ekspanziju promenljivih, dok dvostruki navodnici omogućavaju njihovu interpretaciju.

Shell jezici takođe podržavaju napredne mehanizme kao što su cevi (engl. *pipes*) i redirekcija ulaza i izlaza. Cevi omogućavaju povezivanje izlaza jedne komande sa ulazom druge, čime se formiraju složeni tokovi obrade podataka.

Primer 6.3.4 Razmotrimo naredni kod:

```
for fig in *; do echo ${fig%.*}; done | sort -u
    | wc -l
```

Ovaj primer uklanja ekstenzije iz imena fajlova, sortira rezultate, eliminiše duplikate i na kraju prebrojava jedinstvene vrednosti.

Redirekcija omogućava preusmeravanje izlaza u datoteku:

```
for fig in *; do echo ${fig%.*}; done | sort -u
    > all_figs
```

Na kraju, vredi pomenuti i *shebang* konvenciju. Ukoliko skript započinje nizom karaktera `#!`, operativni sistem koristi naredne karaktere kako bi odredio koji interpreter treba da izvrši skript. Na primer:

```
1 #!/bin/bash
```

Ovaj mehanizam omogućava direktno pokretanje skriptova kao izvršnih fajlova, bez potrebe za eksplicitnim navođenjem interpretera prilikom svakog pokretanja.

6.3.2 Jezici za procesiranje teksta

Jezici za procesiranje teksta i generisanje izveštaja predstavljaju domenski specifične skript jezike, čija je osnova rad sa (proširenim) regularnim izrazima. Među prvim jezicima ovog tipa nalaze se **sed** (engl. *stream editor*) i ubrzo potom **awk**.

Sed je dizajniran za primenu akcija skripta na svaku liniju teksta iz datoteka ili na definisane opsege linija. Ovo je mali jezik sa uskim domenom primene, nečitljivom sintaksom, i skriptovi često nisu pogodni za ponovnu upotrebu. Sed nudi podršku za rad sa regularnim izrazima.

Primer 6.3.5 Razmotrimo naredni kod:

```
1 $ echo "Hello, sed" | sed 's/sed/world/'
2 Hello, world
```

U ovom primeru, `sed` se pokreće iz terminala, pri čemu je ulaz u `sed` izlaz komande `echo`. Naredba `s` (substitucija) menja reč `sed` u reč `world`, ilustrujući kako se `sed` koristi u kombinaciji sa shell-om.

Awk je ime dobio po inicijalima autora Alfred Aho, Peter Weinberger i Brian Kernighan. Nastao je kako bi se olakšalo formatiranje i generisanje izveštaja, kao i ispravljanje nedostataka sed-a. Osnovni princip awk programa je kombinacija obrazaca i akcija, pri čemu se akcije izvršavaju kada se obrasci poklope sa linijama ulaza. Obrasci podržavaju rad sa proširenim regularnim izrazima, a akcije imaju sintaksu sličnu jeziku C.

Primer 6.3.6 Naredna dva primera prikazuju štampanje `Hello, world` u awk-u:

```
$ echo 'this line of data is ignored' > test
$ awk '{ print "Hello, world" }' test
Hello, world

$ awk 'BEGIN { print "Hello, world" }'
Hello, world
```

U prvom primeru se kreira datoteka `test` sa jednom linijom teksta, a awk izvršava štampanje `Hello, world` za svaku liniju ulaza. U drugom primeru, komanda `BEGIN` omogućava izvršavanje akcije pre čitanja bilo kakvog ulaza.

Perl je nastao sa idejom da kombinuje funkcionalnosti jezika sed, awk i sh, ali je ubrzo izrastao u mnogo više od toga. Perl je imao značajan uticaj na razvoj modernih skript jezika, uključujući PHP, Ruby, Python i JavaScript. Prva verzija Perla uključivala je i programe `a2p` i `s2p` za konverziju awk i sed skriptova u Perl, sa ciljem da se ovi jezici u potpunosti zamene.

Perl je jedan od najranijih skript jezika nove generacije, sa podrškom za proširene regularne izraze, pristup sistemskim pozivima, rad sa klasama i objektima, funkcionalne koncepte i razne vrste proširenja. Ovaj jezik je imao sličan domen upotrebe kao Python i u mnogome je postao osnova za razvoj modernih skript jezika.

Slika 6.1: Perl logo `https://www.perl.org/`

Perl je nastao 1987. godine, a dizajnirao ga je Leri Vol (eng. *Larry Wall*). Poslednja verzija, 5.32.1, izdata je 23. januara 2021. Ime *Perl* izvedeno je iz engleske reči *pearl* (biser), ali bez slova `a`. Iako je Perl prvobitno nastao kao jezik za procesiranje teksta, danas spada u jezike

opšte namene sa veoma širokim mogućnostima primene, uključujući i primene na vebu.

Primer 6.3.7 Perl skriptovi obično imaju ekstenziju `.pl`. Neka je sadržaj datoteke `hello.pl`:

```
print "Hello, world\n"
```

Pokretanje iz komandne linije:

```
$ perl hello.pl
Hello, world
```

U razmatranju Perla, naročito je važno istaći koncept promenljivih i tipove podataka koje je jezik podržavao. Promenljive u Perlu su statički tipizirane, što ga razlikuje od mnogih drugih skript jezika. Sam jezik je slabo tipiziran a omogućava implicitnu deklaraciju promenljivih, tj nisu neophodne deklaracije promenljivih.

Postoje tri osnovna prostora imena za promenljive, koja se razlikuju prema prvom karakteru imena promenljive. Skalarne promenljive, koje uključuju i stringove i brojeve, označavaju se znakom `$`. Nizovne promenljive, odnosno liste, počinju sa znakom `@`. Treća kategorija su asocijativni nizovi, poznati kao heševi, koji se indeksiraju stringovima i čija je implementacija zasnovana na heš tabelama, a označavaju se znakom `%`. Ova konvencija omogućava čitljivija imena promenljivih u odnosu na mnoge druge skript jezike.

Perl se može posmatrati i kao naziv za familiju jezika. Dok se termin Perl primarno odnosi na Perl 5, između 2000. i 2019. godine odnosio se i na Perl 6, koji se paralelno razvijao, a potom je preimenovan u jezik Raku.

Primer 6.3.8 Raku skriptovi mogu imati različite ekstenzije, uključujući `.raku` i `.p6`. Primer jednostavnog Raku programa koji ispisuje pozdravnu poruku je:

```
print "Hello, world\n"

say "Hello, world"
```

Raku predstavlja modernizovanu verziju Perla sa do-

Slika 6.2: Logo jezika Raku

datnim konceptima i poboljšanom čitljivošću koda, dok zadržava kompatibilnost sa Perl-om u mnogim njegovim osnovnim operacijama.

6.3.3 Matematika i statistika

Jezici poput Maple, Wolfram Mathematica i MATLAB pripadaju grupi domenski specifičnih jezika sa naglaskom na matematičke i statističke primene. Moderni naslednici jezika `APL`* podržavaju numeričke metode, simboličku matematiku, vizuelizaciju podataka i matematičko modelovanje. Njihove primene obuhvataju rešavanje konkretnih problema, izradu simulacija i prototipova.

Primer 6.3.9 `Hello, world` program u MATLAB-u se piše na sledeći način:

```
1 disp('Hello world')
```

Skript programi u MATLAB-u obično imaju ekstenziju `.m`.

U oblasti statistike, posebno značajni su jezici S i R (otvorenog koda), koji omogućavaju statističku obradu podataka u raznim domenima. Ovi jezici podržavaju

* Jezik nastao šezdesetih godina prošlog veka, prvobitno osmišljen kao notacija za podučavanje primenjene matematike. Kao programski jezik, zadržao je akcenat na konciznom i elegantnom izražavanju matematičkih algoritama, ali ga karakteriše neobična i teško čitljiva sintaksa.

rad sa višedimenzionalnim nizovima i listama, funkcionalno programiranje, kao i proširenje infiksnim operatorima.

Primer 6.3.10 Primer jednostavnog programa u jeziku R prikazuje ispis pozdravne poruke u okviru interpretera:

```
> print("Hello, world")
[1] "Hello, world"
> # Quotes can be suppressed in the output
> print("Hello, world", quote = FALSE)
[1] Hello, world
```

Skripte se mogu pisati i u zasebnim datotekama, obično sa ekstenzijom `.r`.

6.3.4 Jezici proširenja

Jezici proširenja omogućavaju korisnicima da dodaju nove funkcionalnosti postojećim aplikacijama koristeći osnovne komande alata kao gradivne blokove. Većina savremenih aplikacija podržava interaktivne komande, koje se mogu izvršavati tekstualno ili putem događaja iniciranih korisničkim interfejsom, kao što je klik mišem na meni. Na primer, komande programa za crtanje mogu uključivati čuvanje ili učitavanje crteža, izbor ili izmenu delova crteža, promenu stila linije, boje ili debljine linije, kao i zumiranje ili rotaciju prikaza. Korišćenjem jezika proširenja, ove operacije mogu se automatizovati i kombinovati, čime se izbegava ručno izvršavanje repetitivnih zadataka.

Da bi aplikacija podržavala izmene putem jezika proširenja, potrebno je da alat sadrži ili komunicira sa interpreterom odgovarajućeg skript jezika, da omogući skriptu pozivanje postojećih komandi alata i da dozvoli povezivanje novodefinisanih komandi sa događajima korisničkog interfejsa.

Primeri primene jezika proširenja uključuju grafičke alate iz Adobe paketa, kao što su Illustrator i Photoshop, koji omogućavaju dopune putem jezika JavaScript, Visual Basic ili AppleScript. AutoCAD i Flash imaju svoje

specifične skript jezike, dok se jezik Lua često koristi za razvoj skripti u igrama. Microsoftovi alati uglavnom koriste Visual Basic. Open-source program GIMP (GNU Image Manipulation Program) može koristiti različite jezike, uključujući Scheme, Tcl, Python i Perl, pri čemu je najčešće korišćenje jezika Scheme.

6.3.5 Jezici za veb

Najznačajnija primena skript jezika danas je u programiranju veb aplikacija. Skriptovi se mogu izvršavati na strani klijenta, u okviru veb pregledača, ili na strani servera, gde omogućavaju implementaciju dinamičkih veb-strana i veb aplikacija. Među najpoznatijim jezicima za veb klijent spada `JavaScript`, dok se za skriptovanje na strani servera koriste jezici poput `PHP`, `Perl`, `Ruby`, `Python` i takođe `JavaScript`.

JavaScript

Upotreba veba značajno je porasla sredinom 1990-ih sa pojavom prvih grafičkih pregledača. JavaScript je originalno razvijen u kompaniji Netscape, a njegov dizajner je Brendan Eich. Jezik je prvobitno nosio ime `Mocha`, zatim je preimenovan u `LiveScript`, da bi krajem 1995. postao zajednički projekat kompanija Netscape i Sun Microsystems i dobio ime `JavaScript`. Iako deli ime sa jezikom Java, JavaScript sa njim nema mnogo zajedničkog. Razvoj JavaScript-a bio je pod uticajem brojnih programskih jezika, uključujući `awk`, `C`, `HyperTalk`, `Java`, `Lua`, `Perl`, `Python`, `Scheme` i `Self`.

Moderan JavaScript, zajedno sa `HTML`-om i `CSS`-om, predstavlja osnovnu tehnologiju za razvoj veb aplikacija. Po nekim merenjima, JavaScript je jedan od najpopularnijih i najkorišćenijih programskih jezika već duži niz godina. Omogućava kreiranje interaktivnih veb stranica i koristi se za definisanje ponašanja na strani klijenta, pri čemu svi najvažniji veb pregledači implementiraju odgovarajuće izvršno okruženje. Pored toga, JavaScript se može koristiti i za razvoj skriptova na strani servera.

JavaScript je multiparadigmatski jezik koji podržava programiranje vođeno događajima, funkcionalno i imperativno programiranje. To je interpretiran, slabo tipiziran jezik. Razvoj JavaScript aplikacija podržan je kroz različite razvojne okvire. Za front-end razvoj najpopularniji su `Angular`, `React`, `Vue` i `Ember`, dok se za serversku stranu koriste `Express JS`, `Next JS`, `Gatsby JS`, `Nuxt JS` i `Node JS`.

Primer 6.3.11 Jednostavan `Hello, world` primer u JavaScript-u, koji se izvršava u okviru HTML dokumenta, može se napisati na sledeći način:

```
<!DOCTYPE HTML>
<html>
<body>
  <p>Before the script...</p>
  <script>
    alert('Hello, world!');
  </script>
  <p>...After the script.</p>
</body>
</html>
```

Skripte takođe mogu biti smeštene u zasebne fajlove sa ekstenzijom `.js` i uključene u HTML dokumente. Alternativno, umesto funkcije `alert`, može se koristiti `document.write('Hello, World!');`.

PHP

PHP (Hypertext Preprocessor) je jezik za skriptovanje na strani servera. Njegov razvoj je započeo 1994. godine, a dizajner jezika bio je Rasmus Lerdorf. Prvobitno je korišćen za potrebe praćenja broja poseta privatnog veb sajta, pod nazivom `PHP Tools` (Personal Home Page Tools). Godine 1997. jezik je kompletno redizajniran i preimenovan u `Hypertext Preprocessor`. Prema nekim statistikama, više od 75% servera koristi PHP.

PHP je skript jezik koji kombinuje osobine imperativnog, funkcionalnog i objektno-orijentisanog programiranja.

Interpretiran je, dinamički i slabo tipiziran. Za razvoj PHP aplikacija koriste se brojni razvojni okviri, uključujući `Laravel`, `CodeIgniter`, `Symfony`, `CakePHP`, `Yii` i `Zend`.

Primer 6.3.12 Primer jednostavnog `Hello, world` PHP skripta smeštenog u datoteci `hello.php`. Datoteka hello.php na serveru, levo. Kada joj se pristupi, ukoliko je sve u redu konfigurisano, generiše se izlaz, desno.

```
<html>
 <head>
  <title>PHP Test</
    title>
 </head>
 <body>
 <?php echo '<p>Hello
     World</p>'; ?>
 </body>
</html>
```

```
<html>
 <head>
  <title>PHP Test</
    title>
 </head>
 <body>
 <p>Hello World</p>
 </body>
</html>
```

6.4 Jezici opšte namene

Skript jezici opšte namene koriste se u različitim domenima primene. Među najpoznatijim jezicima ovog tipa su Perl, Python, Ruby i Lua. Ovi jezici omogućavaju brzo generisanje prototipova, razvoj veb aplikacija, naučne primene, kao i automatizaciju različitih zadataka.

6.4.1 Python

Python (`https://www.python.org/`) je skript jezik otvorenog koda koji se koristi u brojnim domenima. Njegove primene uključuju:

- veb programiranje, posebno skriptove na strani servera,
- nauku o podacima i analitiku,
- veštačku inteligenciju i mašinsko učenje,
- naučne simulacije i računarske aplikacije.

Slika 6.3: Python logo

Python omogućava programerima da se fokusiraju na rešavanje problema, a ne na detalje sintakse jezika. Zbog jednostavne i čitljive sintakse, Python je često preporučen kao prvi jezik za učenje programiranja, pa čak i u školama. Prva verzija Pythona objavljena je 1991. godine, a razvio ga je Guido van Rossum.

Python integriše ideje iz različitih jezika, poput Perla, Haskella i objektno-orijentisanih jezika. Iako pripada skript paradigmi, podržava i imperativni, objektno-orijentisani i funkcionalni stil programiranja. Jezik je kompaktan i oslanja se na bogate biblioteke koje omogućavaju različite funkcionalnosti; na primer, rad sa regularnim izrazima obezbeđuje modul `re`, a ne sam jezik.

Velika dostupnost biblioteka i resursa za učenje doprinosi popularnosti Pythona i širokoj primeni u industriji i nauci.

Primer 6.4.1 Python skriptovi imaju ekstenziju `.py`. Sledeći primer prikazuje najjednostavniji program:

Listing 6.1: Hello world program u Pythonu

```
1 print("Hello, world")
```

Ovaj skript se izvršava komandnom linijom:

```
1 python hello.py
```

i daje izlaz:

```
1 Hello, world
```

Python je programski jezik koji se odlikuje jednostavnošću učenja, čitanja i pisanja, zbog čega je posebno pogodan za početnike, ali i za iskusne programere koji

teže preglednom i razumljivom kodu. Njegova izražajnost i konciznost omogućavaju da se složene ideje predstave na jasan i kratak način. Python je slobodan i otvorenog koda, što znači da je dostupan svima za korišćenje, proučavanje i unapređivanje. Kao jezik visokog nivoa, apstrahuje detalje rada sa hardverom i omogućava fokus na rešavanje problema.

Pored toga, Python je portabilan i proširiv, što znači da se programi napisani u ovom jeziku mogu pokretati na različitim platformama uz minimalne izmene, a funkcionalnost jezika se može dodatno proširivati. Podržava objektno-orijentisano programiranje, ali i druge programske paradigme, čime pruža veliku fleksibilnost u razvoju softvera. Python je interpretiran i može se ugraditi u druge sisteme, što ga čini pogodnim za različite vrste primena.

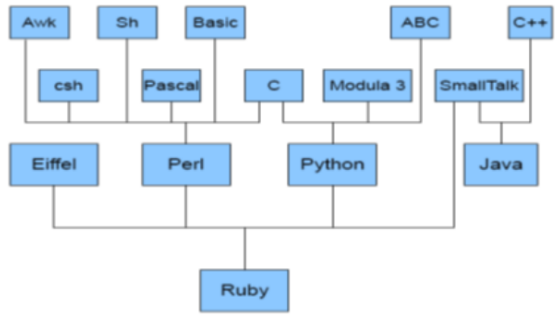
Jedna od njegovih značajnih prednosti je velika standardna biblioteka koja obuhvata širok spektar gotovih modula za različite zadatke, uključujući i razvoj grafičkih korisničkih interfejsa. Na kraju, Python koristi dinamičko, ali strogo tipiziranje, što znači da se tipovi promenljivih određuju tokom izvršavanja programa, ali se pravila tipova dosledno poštuju.

6.4.2 Ruby

Ruby (`https://www.ruby-lang.org/`) je jezik opšte namene otvorenog koda, dinamički tipiziran, sa fokusom na jednostavnost i produktivnost. Ima elegantnu sintaksu koja je prirodna za čitanje i pisanje. Ruby je razvijen 1995. godine od strane Yukihiro "Matz" Matsumoto u Japanu.

Slika 6.4: Ruby logo

Razvoj jezika Ruby je bio pod uticajem mnogih programskih jezika, među kojima su Eiffel, Perl, Python,



Slika 6.5: Razvojno stablo

Smalltalk, Ada, Basic i Lisp. Ruby kombinuje objektno-orijentisano (Eiffel, Smalltalk), funkcionalno (Lisp) i imperativno (Perl, Python, Ada, Basic) programiranje. Jezik je interpretiran, dinamički i strogo tipiziran, a osnovna ideja dizajna bila je da se adresiraju ljudske potrebe – programiranje treba da čini programere produktivnim i zadovoljnim.

Primer 6.4.2 Ekstenzija Ruby skriptova je `.rb`. Najjednostavniji program "Hello, world" prikazan je u nastavku:

Listing 6.2: Hello world program u Ruby-u

```
1 puts "Hello, world"
```

Skripta se izvršava komandnom linijom:

```
1 ruby hello.rb
```

i daje izlaz:

```
1 Hello, world
```

Ruby je programski jezik opšte namene koji se može koristiti za razvoj različitih vrsta aplikacija, uključujući i desktop aplikacije. Implementiran je u programskom jeziku C, a u svom dizajnu inspirisan je pre svega jezicima Perl i Smalltalk, od kojih preuzima određene koncepte i filozofiju razvoja softvera.

Ruby on Rails je razvojni okvir namenjen izradi veb aplikacija zasnovanih na radu sa bazama podataka, implementiran u Ruby-ju. Ovaj okvir značajno pojednostavljuje i ubrzava razvoj veb sistema kroz upotrebu konvencija i gotovih rešenja. Rails je, između ostalog, inspirisan okvirima kao što su Django (Python) i Laravel

(PHP), preuzimajući ideje koje doprinose efikasnijem razvoju savremenih veb aplikacija.

6.4.3 Lua

Lua (`http://www.lua.org/`) je nastala 1993. godine u Brazilu, a autori su Roberto Ierusalimschy, Waldemar Celes i Luiz Henrique de Figueiredo. Ime Lua znači *mesec* na portugalskom.

Slika 6.6: Lua logo

Lua podržava proceduralno, objektno-orijentisano i funkcionalno programiranje, kao i programiranje vođeno podacima. Na razvoj jezika Lua uticali su jezici C++ Modula, Scheme i SNOBOL. Lua je dinamički tipiziran jezik, a bajtkod se izvršava na virtuelnoj mašini napisanoj u C-u, uz automatsko upravljanje memorijom (inkrementalni sakupljač otpadaka). Lua se koristi za pisanje skriptova, konfiguracija i brzo pravljenje prototipova.

Lua je robustan jezik sa dobrim performansama, posebno u industrijskim aplikacijama i industriji igara (npr. *World of Warcraft*, *Angry Birds*). Portabilan je i može se koristiti svuda gde postoji kompajler za standardni C. Takođe se lako ugrađuje u druge jezike, posebno C/C++, ali i u Javu, C#, Fortran, Smalltalk, Ada, Erlang, kao i u druge skript jezike, poput Perla i Ruby. Lua je mali i jednostavan jezik, ali sa velikim mogućnostima, i besplatan je softver otvorenog koda.

Primer 6.4.3 Ekstenzija Lua skriptova je `.lua`. "Hello, world" primer:

```
print("Hello, world")
```

Skripta se izvršava komandnom linijom:

```
lua hello.lua
```

i daje izlaz:

```
Hello, world
```

Listing 6.3: Hello world program u Lua-u

Rezime

-
-
-
-

Pitanja

1.
2.
3.
4.
5.

Konkurentno programiranje 7

Pregled

-
-
-
-

7.1 ...

7.2 ...

Rezime

-
-
-
-

Ispitna pitanja

1.
2.
3.
4.
5.

Dodatne programske paradigme 8

Pregled

- ►
- ►
- ►
- ►

8.1 Komponentno programiranje

Savremeni razvoj softvera sve više teži principima koji su odavno prisutni u inženjerskim disciplinama, poput elektronike i mašinstva, gde se složeni sistemi konstruišu sklapanjem unapred definisanih i testiranih komponenti. Osnovna ideja komponentnog pristupa jeste da se softver gradi od većih, gotovih celina koje predstavljaju zaokružene funkcionalne jedinice, čime se značajno pojednostavljuje proces razvoja i povećava pouzdanost sistema.

Softverska komponenta može se posmatrati kao jedinica funkcionalnosti sa jasno definisanim i ugovorenim interfejsom. Interfejs određuje način komunikacije sa komponentom i u potpunosti je odvojen od njene implementacije, čime se omogućava nezavisnost razvoja i lakša zamena ili unapređenje pojedinačnih delova sistema. Na taj način postiže se veći stepen apstrakcije i približavanje deklarativnom stilu programiranja, gde se fokus pomera sa načina realizacije ka opisu željene funkcionalnosti.

Komponentna paradigma može se posmatrati kao proširenje ili potparadigma objektno-orijentisanog programiranja, budući da se zasniva na sličnim principima enkapsulacije i modularnosti, ali ih podiže na viši nivo apstrakcije. Softverske komponente predstavljaju kolekcije međusobno povezanih elemenata, kao što su objekti

i metode, koje zajedno ostvaruju određenu funkcionalnost. Slično tehničkim komponentama, one mogu biti jednostavne ili složene, mogu funkcionisati samostalno ili u saradnji sa drugim komponentama.

Izgradnja kompleksnih softverskih sistema zasniva se na međusobnom povezivanju komponenti. Da bi ovaj proces bio efikasan, neophodno je da način njihovog povezivanja bude što jednostavniji i intuitivniji. U idealnom slučaju, razvoj softvera podrazumeva izbor odgovarajućih komponenti i njihovo pozicioniranje unutar sistema, umesto tradicionalnog pisanja koda liniju po liniju. Ovakav pristup posebno je vidljiv u razvoju korisničkih interfejsa, gde se elementi poput prozora, tekstualnih polja i dugmadi često kreiraju metodom „prevuci i pusti“ (drag-and-drop).

Važnu ulogu u komponentnom programiranju ima i razvojno okruženje koje podržava ovakav način rada. Iako same komponente mogu biti implementirane u različitim programskim jezicima, kao što su Python, Java, C++ ili C#, njihov način korišćenja i povezivanja ostaje u velikoj meri uniforman. Time se omogućava interoperabilnost i dodatno podstiče ponovna upotreba jednom razvijenih softverskih rešenja, što predstavlja jedan od ključnih ciljeva komponentnog pristupa.

8.2 Paradigma upitnih jezika

Upitni jezici predstavljaju važnu klasu programskih i specifikacionih jezika koji se koriste za izdvajanje informacija iz različitih izvora podataka. Oni mogu biti namenjeni radu sa bazama podataka, ali i širem domenu pronalaženja informacija (engl. *information retrieval*). Osnovna karakteristika ovih jezika jeste njihova deklarativna priroda: korisnik ne opisuje kako se do rezultata dolazi, već definiše koje uslove rezultat treba da zadovolji.

8.2.1 Upitni jezici baza podataka

Upitni jezici baza podataka omogućavaju dobijanje konkretnih odgovora na osnovu strukturiranih podataka organizovanih u okviru baza podataka. Polazeći od unapred definisanih činjenica, korisnik formuliše upit kojim opisuje željene uslove, dok sistem automatski pronalazi sve podatke koji te uslove zadovoljavaju.

Najpoznatiji predstavnik ove grupe jezika je SQL (*Structured Query Language*), koji se koristi za rad sa relacionim bazama podataka. Osnovne klauzule ovog jezika su `SELECT`, `FROM` i `WHERE`, koje omogućavaju izdvajanje podataka iz tabela prema zadatim kriterijumima.

Na primer, sledeći upit izdvaja imena svih studenata čije je prezime Petrovic:

```
SELECT Ime
FROM Studenti
WHERE Prezime='Petrovic';
```

Pored SQL-a, značajni su i drugi upitni jezici prilagođeni različitim modelima podataka. SPARQL je jezik namenjen radu sa podacima predstavljenim u formatu RDF (*Resource Description Framework*). On omogućava postavljanje upita nad semantički strukturiranim podacima.

Primer SPARQL upita kojim se izdvajaju studenti osnovnih studija, kursevi koje pohađaju i njihova imena dat je u nastavku:

```
select ?x ?y ?n
where {
  ?x a :UndergradStudent .
  ?x :takesCourse ?y .
  ?x :name ?n
}
```

Za rad sa XML dokumentima koristi se XQuery, koji omogućava pretraživanje i transformaciju hijerarhijski

strukturiranih podataka. Ovaj jezik koristi konstrukcije poput `for`, `where`, `order by` i `return`.

Na primer, sledeći upit izdvaja naslove knjiga čija je cena veća od 30, sortirane po naslovu:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

8.2.2 Upitni jezici za pronalaženje informacija

Za razliku od upitnih jezika baza podataka, koji rade nad strogo strukturiranim podacima, upitni jezici za pronalaženje informacija koriste se za pretragu kolekcija dokumenata u cilju pronalaženja onih koji su relevantni za određenu temu ili oblast istraživanja. Ovi jezici često rade nad polustrukturiranim ili nestrukturiranim podacima, kao što su tekstualni dokumenti, web stranice ili bibliografski zapisi.

Jedan od predstavnika ove grupe je CQL (*Contextual Query Language*), formalni jezik namenjen iskazivanju upita u sistemima za pretragu informacija, kao što su web indeksi, bibliotečki katalozi ili baze muzejskih zbirki. CQL je dizajniran tako da bude jednostavan za čitanje i pisanje, uz zadržavanje izražajne moći potrebne za formulisanje složenijih upita.

Primeri CQL upita uključuju:

```
title any fish
title any fish sortBy date/sort.ascending
title any fish or creator any sanderson
```

Na ovaj način, upitni jezici omogućavaju efikasno i fleksibilno izdvajanje informacija iz različitih tipova podataka, uz visok nivo apstrakcije i minimalnu potrebu za opisivanjem proceduralnih detalja izvršavanja.

8.3 Generička paradigma

Generička paradigma predstavlja stil programiranja u kojem se algoritmi definišu nad apstrahovanim tipovima podataka, bez vezivanja za konkretne implementacije. Umesto da se isti algoritam implementira više puta za različite tipove, generičko programiranje omogućava da se tipovi prosleđuju kao parametri, čime se postiže veća fleksibilnost i značajno smanjuje dupliranje koda.

Osnovna ideja ove paradigme jeste razdvajanje logike algoritma od konkretnih tipova nad kojima se algoritam izvršava. Na taj način, jednom definisani algoritmi mogu se primeniti na različite strukture podataka, pod uslovom da zadovoljavaju određene zahteve (npr. postojanje operacija poređenja). Tipični primeri uključuju algoritme za određivanje minimuma i maksimuma, sortiranje ili pretragu, koji se mogu koristiti nad različitim tipovima podataka bez potrebe za pisanjem posebnih verzija za svaki tip.

Koreni generičkog programiranja mogu se pronaći još u jeziku ML iz 1973. godine, gde se ova ideja pojavljuje kroz parametarske tipove i polimorfizam. Danas je generička paradigma široko rasprostranjena i implementirana na različite načine u savremenim programskim jezicima.

U mnogim jezicima, kao što su Python, Ada, C#, F#, Java, Nim, Rust, Swift i TypeScript, generičnost je podržana kroz mehanizme poznate kao *generics*, koji omogućavaju parametrizaciju tipova. U jezicima kao što su C++ i D koristi se koncept šablona (engl. *templates*), koji pruža moćan i fleksibilan način za generisanje koda u vreme prevođenja. Sa druge strane, jezici poput ML-a, Scala-e, Julia-e i Haskell-a oslanjaju se na implicitni polimorfizam, gde se generičnost ostvaruje bez eksplicitnog navođenja tipova u mnogim slučajevima.

Zahvaljujući ovim mehanizmima, generičko programiranje doprinosi razvoju apstraktnijeg, ponovo iskoristivog i lakše održivog koda, što ga čini jednim od ključnih principa savremenog softverskog inženjerstva.

8.4 Vizuelna paradigma

Vizuelna paradigma programiranja zasniva se na modelovanju sistema korišćenjem grafičkih elemenata, pri čemu se struktura i ponašanje softvera predstavljaju putem različitih dijagrama. Ovakav pristup omogućava intuitivnije razumevanje složenih sistema i usko je povezan sa objektno-orijentisanom paradigmom, budući da često koristi slične koncepte kao što su objekti, klase i njihove međusobne relacije.

U okviru vizuelne paradigme, za opis akcija, svojstava i veza između različitih delova sistema koriste se grafički simboli i dijagrami. Ovi vizuelni jezici posebno su značajni u fazi projektovanja softvera, gde omogućavaju da se na jasan i pregledan način predstavi arhitektura sistema pre same implementacije. Zbog toga su naročito pogodni za komunikaciju između članova tima, kao i za dokumentovanje sistema.

Najznačajniji predstavnik vizuelne paradigme je UML (*Unified Modeling Language*), standardizovani jezik za modelovanje koji obuhvata širok skup dijagrama za opisivanje različitih aspekata softverskog sistema. UML je pre svega namenjen za izradu konceptualnih i projektnih „skica“ sistema, dok se ređe koristi za detaljan opis koji bi direktno odgovarao implementaciji.

UML definiše ukupno 14 tipova dijagrama, koji se mogu podeliti u dve osnovne grupe: dijagrame koji opisuju strukturne aspekte sistema i dijagrame koji opisuju njegovo ponašanje. Strukturni dijagrami prikazuju statičku organizaciju sistema i odnose između njegovih delova, dok dijagrami ponašanja opisuju dinamiku sistema i tok izvršavanja.

Primeri strukturnih dijagrama uključuju:

- dijagram klasa, koji prikazuje klase, njihove atribute, metode i međusobne veze,
- dijagram objekata, koji predstavlja konkretne instance klasa u određenom trenutku,
- dijagram komponenti, koji opisuje organizaciju i zavisnosti između softverskih komponenti.

Sa druge strane, dijagrami ponašanja obuhvataju:

- dijagram aktivnosti, koji modeluje tok izvršavanja i poslovne procese,
- dijagram komunikacije, koji prikazuje razmenu poruka između objekata,
- dijagram slučajeva upotrebe, koji opisuje interakciju korisnika sa sistemom.

Savremena razvojna okruženja često pružaju podršku za rad sa UML dijagramima, uključujući mogućnost automatskog generisanja izvornog koda na osnovu vizuelnih modela. Na taj način se dodatno smanjuje jaz između faze projektovanja i implementacije, čime vizuelna paradigma dobija značajnu ulogu u modernim procesima razvoja softvera.

8.5 Reaktivna paradigma

Reaktivna paradigma predstavlja pristup programiranju koji je usmeren na tok podataka i propagaciju promena, posebno u kontekstu asinhronih sistema. Osnovna ideja je da se promene vrednosti podataka automatski prenose kroz sistem, tako da svi zavisni delovi reaguju na te promene bez potrebe za eksplicitnim upravljanjem tokom izvršavanja.

Za razliku od imperativnog programiranja, gde izraz poput $a = b + c$ predstavlja jednokratnu dodelu vrednosti promenljivoj a na osnovu trenutnih vrednosti promenljivih b i c, u reaktivnom programiranju isti izraz ima drugačije značenje. Naime, svaka promena vrednosti promenljivih b ili c automatski dovodi do ažuriranja vrednosti promenljive a. Ovakvo ponašanje može se uporediti sa radom tabela u alatima kao što su Excel ili LibreOffice Calc, gde promene u jednoj ćeliji utiču na sve zavisne formule.

Zahvaljujući svojoj sposobnosti da prirodno modeluje asinhrone tokove podataka, reaktivna paradigma postaje sve značajnija u savremenom razvoju softvera. Njena

primena je naročito izražena u razvoju veb servisa, mobilnih aplikacija, kao i sistema koji zahtevaju obradu podataka u realnom vremenu.

Osnovne karakteristike reaktivnih aplikacija mogu se opisati kroz nekoliko ključnih svojstava:

- **Vođenost događajima** (engl. *event-driven*) — sistem reaguje na događaje i promene stanja (oslanja se na asinhronu razmenu poruka), umesto da se oslanja na sekvencijalno izvršavanje instrukcija.
- **Odzivnost** (engl. *responsive*) — sistem brzo reaguje na zahteve korisnika, čak i u uslovima velikog opterećenja ili delimičnih otkaza.
- **Otpornost** (engl. *resilient*) — sistem je sposoban da se oporavi od različitih vrsta otkaza, kao što su softverski izuzeci, hardverski problemi ili prekidi komunikacije. Ova osobina se postiže pažljivim dizajnom arhitekture zasnovane na slabo povezanim komponentama.
- **Skalabilnost** (engl. *scalable*) — sistem se prilagođava povećanom opterećenju efikasnim raspoređivanjem resursa. Sistem ostaje responzivan pod promenljivim opterećenjem (elastičnost (Elastic)).

Da bi se ovakva svojstva ostvarila, reaktivne aplikacije se oslanjaju na specifične apstrakcije i arhitektonske obrasce. Reaktivna paradigma se zasniva na naprednim konceptima konkurentnog i asinhronog programiranja, ali nije vezana za konkretan programski jezik. Ona predstavlja opšti stil programiranja koji se može primeniti u različitim jezicima i tehnologijama, često uz pomoć specijalizovanih biblioteka i okvira. Zbog toga danas postoji širok spektar alata i pristupa koji podržavaju razvoj reaktivnih sistema u različitim okruženjima. Usled rastućih zahteva za skalabilnim, pouzdanim i responzivnim sistemima, reaktivno programiranje postaje jedan od ključnih pristupa u savremenom softverskom inženjerstvu.

8.6 Programiranje ograničenja

Programiranje ograničenja (eng. *constraint programming*) predstavlja opšti pristup rešavanju problema u kome se problem modeluje kao sistem ograničenja nad upravljačkim (nepoznatim) promenljivama. Cilj je pronaći dopustivo rešenje, odnosno odrediti vrednosti promenljivih koje zadovoljavaju sva postavljena ograničenja, ili dokazati da takvo rešenje ne postoji.

Programiranje ograničenja pripada deklarativnim programskim paradigmama. Za razliku od imperativne paradigme, gde se postupak rešavanja problema eksplicitno opisuje kroz niz koraka, u programiranju ograničenja definišu se uslovi koje promenljive moraju da zadovolje, dok se sam postupak pronalaženja rešenja prepušta sistemu.

Ograničenja mogu biti različitih vrsta, uključujući ograničenja iskazne logike (npr. $A \vee B$), linearna ograničenja (npr. $x \leq 15$), kao i ograničenja nad konačnim domenima.

Primer 8.6.1 Važna semantička razlika u odnosu na imperativnu paradigmu ogleda se u tumačenju ograničenja. Na primer, izraz $x < y$ u imperativnim jezicima predstavlja logički izraz koji se evaluira kao tačan ili netačan. Nasuprot tome, u programiranju ograničenja ovaj izraz definiše relaciju između promenljivih x i y koja mora biti zadovoljena u svakom dopustivom rešenju.

Programiranje ograničenja predstavlja savremen pristup rešavanju složenih kombinatornih problema. Njegova primena je naročito izražena u oblasti operacionih istraživanja, gde se koristi za rešavanje optimizacionih i kombinatornih problema, kao što su raspoređivanje resursa, planiranje i optimizacija.

Primer 8.6.2 Pekara *Kiflica* proizvodi hleb i kifle. Za mešenje i pečenje hleba potrebno je 10 minuta, dok je

za kiflu potrebno 12 minuta. Testo za hleb sadrži 300g brašna, a testo za kiflu 120g brašna. Zarada po hlebu iznosi 7 dinara, a po kifli 9 dinara. Pekara raspolaže sa ukupno 20 radnih sati i 20kg brašna dnevno. Potrebno je odrediti broj proizvedenih hlebova i kifli tako da se maksimizuje ukupna zarada, pod pretpostavkom da će svi proizvodi biti prodati.

Primer 8.6.3 Kompanija raspolaže sa 250 zaposlenih i budžetom od 26000 evra za obuku. Obuka za programski jezik Elixir košta 100 evra po zaposlenom i donosi produktivnost od 150 projekat-sati mesečno uz dobit od 5 evra po satu. Obuka za Dart košta 105 evra i donosi 170 projekat-sati uz dobit od 6 evra po satu. Maksimalni kapacitet iznosi 51200 projekat-sati mesečno. Potrebno je odrediti optimalnu raspodelu obuke radi maksimizacije dobiti.

Sa praktičnog aspekta, programiranje ograničenja predstavlja softversku tehnologiju za deklarativno modelovanje i efikasno rešavanje problema. Korisnik najpre formuliše problem pomoću ograničenja, nakon čega se koristi odgovarajući rešavač ograničenja za pronalaženje rešenja. Sa naučnog aspekta, programiranje ograničenja je multidisciplinarna oblast koja obuhvata metode iz računarskih nauka, veštačke inteligencije, operacionih istraživanja, baza podataka, teorije grafova i logičkog programiranja.

Programiranje ograničenja potiče iz oblasti logičkog programiranja, posebno iz rada na sistemima kao što je Prolog II (Jaffar i Lassez, 1987). Često se implementira u okviru logičkih programskih sistema kroz pristup poznat kao *constraint logic programming* (CLP).

Podrška za programiranje ograničenja može biti ugrađena u programske jezike ili dostupna putem specijalizovanih biblioteka. Među jezicima koji imaju direktnu podršku izdvajaju se Claire, Curry, Kaleidoscope, Oz i Wolfram jezik. Takođe, postoji veliki broj biblioteka za jezike kao što su C, C++, Java i Python, koje imple-

mentiraju različite pristupe rešavanju (npr. SAT i SMT rešavače).

Programiranje ograničenja nad konačnim domenima zasniva se na tri osnovna koraka:

1. generisanje promenljivih i njihovih domena,
2. definisanje ograničenja nad promenljivama,
3. instanciranje promenljivih (labeling).

Primer 8.6.4 Kriptoaritmetički problemi predstavljaju matematičke zadatke u kojima su cifre zamenjene slovima, a cilj je pronaći odgovarajuće cifre koje zadovoljavaju zadatu jednačinu. Na primer:

```
SEND
+MORE
-----
MONEY
```

Ovakvi problemi su pogodni za ilustraciju principa programiranja ograničenja, ali ne predstavljaju tipičnu primenu ove paradigme.

Primer 8.6.5 U programskom jeziku Python, modul *python-constraint* omogućava rad sa ograničenjima nad konačnim domenima.

```
import constraint

problem = constraint.Problem()

problem.addVariable("a", [1,2,3])
problem.addVariable("b", [4,5,6])

resenja = problem.getSolutions()
print(resenja)
```

Uvođenje ograničenja vrši se definisanjem funkcije koja opisuje relaciju između promenljivih:

Primer 8.6.6

```
import constraint

problem = constraint.Problem()

```

```
problem.addVariable("a", [1,2,3])
problem.addVariable("b", [4,5,6])

def o(a,b):
    if(2*a > b):
        return True

problem.addConstraint(o, "ab")

resenja = problem.getSolutions()
print(resenja)
```

Primer 8.6.7 Primer: SEND + MORE = MONEY

```
import constraint

problem = constraint.Problem()

problem.addVariables('SM', range(1,10))
problem.addVariables('ENDORY', range(10))

def o(s,e,n,d,m,o,r,y):
    return (s*1000 + e*100 + n*10 + d +
            m*1000 + o*100 + r*10 + e) == \
           (10000*m + 1000*o + 100*n + 10*e + y)

problem.addConstraint(o, "SENDMORY")
problem.addConstraint(constraint.
    AllDifferentConstraint())

resenja = problem.getSolutions()
for r in resenja:
    print("  "+str(r['S'])+str(r['E'])+str(r['N
    '])+str(r['D']))
    print(" +"+str(r['M'])+str(r['O'])+str(r['R
    '])+str(r['E']))
    print("="+str(r['M'])+str(r['O'])+str(r['N
    '])+str(r['E'])+str(r['Y']))
```

Primer 8.6.8 B-Prolog je logički programski jezik sa bogatom podrškom za programiranje ograničenja. Podržava rad sa različitim domenima, uključujući konačne domene.

Primer za problem `SEND + MORE = MONEY`:

```
sendmoremoney(Vars) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars :: 0..9,
    S #\= 0,
    M #\= 0,
    all_different(Vars),
    1000*S + 100*E + 10*N + D +
    1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y,
    labeling(Vars).
```

Programiranje ograničenja predstavlja moćnu i široko primenjivu programsku paradigmu. Većina savremenih programskih jezika poseduje odgovarajuću podršku kroz biblioteke ili ugrađene mehanizme.

Ključ uspešne primene ove paradigme leži u pravilnom modelovanju problema pomoću sistema ograničenja. Nakon toga, izbor konkretne implementacije ili biblioteke postaje sekundaran, jer se osnovni principi lako prenose između različitih tehnologija.

Najvažnije je prepoznati probleme koji se mogu efikasno modelovati kao problemi ograničenja, kako bi se iskoristile prednosti savremenih rešavača i omogućilo brzo i efikasno pronalaženje rešenja.

Rezime

-
-
-
-

Pitanja

1. Šta je programska paradigma i koje je značenje pojma paradigma uopšte?
2. Koja je uloga programskih paradigmi u razvoju softvera?
3. Šta je programski jezik?

4. Koji je odnos između programskih jezika i programskih paradigmi?
5. Kako su se programski jezici razvijali kroz vreme?
6. Koje su osnovne programske paradigme?
7. Koje su osnovne četiri programske paradigme?
8. Nabroj bar četiri dodatne programske paradigme.
9. Zašto su nastajale i nastaju nove programske paradigme?
10. Šta karakteriše proceduralnu paradigmu?
11. Šta karakteriše imperativnu paradigmu?
12. Šta karakteriše deklarativnu paradigmu?
13. Koje su osnovne karakteristike posmatrane paradigme?
14. Nabroj tri programska jezika koji pripadaju datoj paradigmi.