Programske paradigme

Sadržaj

1 Uvod

1.1 Jezici i programski jezici

Jezik je skup pravila za komunikaciju između subjekata. Pomoću njega se predstavljaju i prenose informacije. Prirodni jezik se koristi za komunikaciju između ljudi u govornoj ili pisanoj formi.
Programski jezik služi, prvenstveno, za komunikaciju između čoveka i računara, ali može da se koristi i za komunikaciju između mašina, kao i za komunikaciju između ljudi. Programski jezici se mogu deliti na razne načine. Jedna podela je:

- *mašinski zavisni*
- *mašinski nezavisni*

Na dalje će uglavnom biti reč o mašinski nezavisnim (višim) programskim jezicima.

1.2 Definicije programskih jezika

- Programski jezik je jezik konstruisan formalno da bi se omogućilo zadavanje instrukcija mašinama, posebno računarima. (Wikipedia)
- Programski jezik je jezik za pisanje programa koje računar zna i može izvršiti.
- Programski jezik je veštački jezik koji služi za opis računarskih programa.
- Programski jezik je veštački jezik za opis konstrukcija (pisanje instrukcija) koje mogu biti prevedene u mašinski jezik i izvršene od strane računara. (American Heritage Dictionary)
- Programski jezik je skup sintaktičkih i semantičkih pravila koja se koriste za opis (definiciju) računarskih programa.
- Programski jezik je notacioni sistem čitljiv za računare i ljude, a služi za opis poslova koje treba da obavi računar.

Postoji veliki broj programskih jezika (broji se u hiljadama). Enciklopedija Britanika pominje preko 2.000. Drugi izvori pominju preko 2.500 dokumentovanih programskih jezika (Bill Kinnersley). „Encyclopedia of Computer Languages", autor Diarmuid Pigott, sa Murdoch Univerziteta iz Australije navodi preko 8.000 jezika.
Naravno nisu svi programski jezici jednako važni i zastupljeni. Nemoguće je proučiti sve programske jezike.

1.3 Paradigme i programski jezici

Reč paradigma je grčkog porekla i znači: primer za ugled, uzor, uzorak, obrazac, šablon. Obično se koristi da označi vrstu objekata koji imaju zajedničke karakteristike. Programska paradigma predstavlja programski obrazac, programski stil, programski sablon, način programiranja. Predstavlja fundamentalni stil programiranja. Međusobno slični programski jezici se klasifikuju u jednu ili više programskih paradigmi.
Broj programskih paradigmi nije tako velik kao broj programskih jezika. Izučavanjem programskih paradigmi upoznaju se globalna svojstva svih jezika koji pripadaju toj paradigmi. Dakle, informacija da neki jezik pripada nekoj paradigmi govori nam o osnovnim svojstvima i mogućnostima jezika. Poznavanje ogređene paradigme nam značajno olakšava da savladamo svaki programski jezik koji pripada toj paradigmi.
Programske paradigme su usko povezane sa programskim jezicima. Svakoj programskoj paradigmi pripada više programskih jezika. Na primer, proceduralnoj paradigmi pripadaju programski jezici Pascal i C, objektno-orijentisanoj paradigmi pripadaju Simula, JAVA...
Potrebno je izučiti svojstva najistaknutijiih predstavnika pojedinih programskih paradigmi.

Koliko jezika paradigmi znaš, toliko vredis.
Koliko predstavnika različitih paradigmi znaš toliko vredis!

Sledeći nivo apstrakcije čine koncepti koji su zajednički za različite paradigme. Jedan programski jezik može podržati više paradigmi, na primer C++ podržava klasičan proceduralni stil, ali i objektno-orijentisani i generički stil programiranja.
Za rešavanje nekog konkretnog problema, posebno je bitan izbor programskog jezika.

1.4 Razvoj jezika i paradigmi

Bitni momenti u razvoju računara:

- Jedan od prvih elektronskih računara 1939. ABC za rešavanje sistema linearnih jednačina.
- ENIAC - prvi elektronski računar opšte namene (1946)
- Konceptualna promena krajem 1940. u vidu fon Nojmanove arhitekture.
- Vezuje se za fon Nojmana i računar EDVAC 1951. iako je o nekim elementima ove arhitekture i ranije bilo reči

1.5 Razvoj jezika

Kratka istorija:

- FORTRAN - FORmula TRANslating system, 1957. John Backus i IBM
- LISP - LISt Processing, 1958. John McCarthy
- COBOL - COmmon Business-Oriented language, 1959. Grace Hopper
- 60-te ALGOL (58, 60, 68)
- 70-te C, Pascal, Smalltalk, Prolog
- 80-te C++, Erlang
- 90-te Haskell, Python, Visual Basic, Ruby, JAVA, PHP, OCaml, Lua, JavaScript,...
- C#, Scala, F#, Elixir

Postoji veliki broj programskih jezika, neki su široko rasprostranjeni, neki se više ne koriste. Nastanak i razvoj programskih jezika dosta dobro se može prikazati pomoću *razvojnog stabla*. Ono omogućava da se sagleda vreme nastanka pojedinih programskih jezika, kao i međusobni uticaj. Ne postoji jedinstveno razvojno stablo (od autora zavisi na koje će jezike staviti akcenat i kako će ih međusobno povezati).
Neka od pitanja koja se postavljaju su:

- U kom razdoblju je nastao najveći broj programskih jezika?
- Koji su najutičajniji programski jezici?
- Kada su nastali najuticajnihi programski jezici?
- Zašto postoji veliki broj programskih jezika?

Nove programske paradigme nastajale su uz težnju da se olakša proces programiranja. Istovremeno, nastanak novih paradigmi povezan je sa efikasnim kreiranjem sve kompleksnijeg softvera. Svaka novonastala paradigma, bila je promovisana preko nekog programskog jezika. Razvoj programskih paradigmi kao i programskih jezika skopčan je i sa razvojem hardvera.
Svaka paradigma ima različita shvatanja. Ne postoji jedinstveno mišljenje naučnika o programskim paradigmama (vrstama programskih paradigmi, njihovom značaju, najistaknutijim programskim jezicima pojedinačnih paradigmi itd.). Moguće su različite podele na programske paradigme.

1.6 Vrste programskih paradigmi

Osnovne programske paradigme

- *Proceduralna* paradigma – osnovni zadatak programera je da opiše način (proceduru) kojim se dolazi do resenja problema.
- *Deklarativna* paradigma – osnovni zadatak programera je da precizno opiše problem, dok se mehanizam programskog jezika bavi pronalaženjem resenja problema.

Vrste programskih paradigmi

Osnovne programske paradigme:

- Imperativna paradigma
- Objektno-orijentisana paradigma
- Funkcionalna paradigma
- Logička paradigma

Ostale paradigme se često tretiraju kao podparadigme ili kombinacije osnovnih.

Napomena o imperativnoj i proceduralnoj paradigmi

Postoji više shvatanja proceduralne paradigme:

1.
   - Proceduralna paradigma je podparadigma imperativne paradigme koju karakteriše, pored naredbi, i njihovo grupisanje u podparadigme (funkcije).
   - U ovom slučaju, u literaturi se često imperativna i proceduralna paradigma koriste kao sinonimi.
   - Imperativna paradigma se karakteriše postojanjem naredbi, dok se deklarativna paradigma karakteriše nepostojanjem naredbi.
2.
   - Proceduralna paradigma je svaka paradigma kod koje se u procesu programiranja opisuje algoritam (procedura) rešavanja problema
   - U ovom slučaju je imperativna paradigma podparadigma proceduralne paradigme dok je deklarativna paradigma (pitanje STA) suprotna od proceduralne paradigme (pitanje KAKO).

Programski jezici i paradigme

Programski jezik je sredstvo koje koristi čovek da izrazi proces pomoću kojeg računar rešava nekakav problem. U zavisnosti od toga na kojoj od ovih reči je akcenat, programskim jezikom je podržana dominantna programska paradigma:

- čovek – logička paradigma
- proces – funkcionalna paradigma
- računar – proceduralna paradigma
- problem – objektno-orijentisana paradigma

Prethodna definicija programskog jezika je prilagođena osnovnm programskim paradigmama. Ova definicija se može dopuniti tako da se preko nje mogu obuhvatiti i druge paradigme. Na primer, modifikacija može biti:
Programski jezik je sredstvo koje koristi čovek da izrazi proces pomoću kojeg računar, koristeći paralelnu obradu rešava nekakav problem. Ako je akcenat na paralelnoj obradi, dolazi se do konkurentne (paralelne) paradigme.

Dodatne programske paradigme

- Komponentna paradigma
- Konkurentna paradigma
- Skript paradigma
- Generička paradigma
- Paradigma programiranja ograničenja
- Paradigma upitnih jezika
- Reaktivna paradigma
- Vizuelna paradigma

1.7 Opisi paradigmi

1.7.1 Imperativna (proceduralna) paradigma

Imperativna paradigma nastala je pod uticajem Fon Nojmanove arhitekture računara. Može se reći da se zasniva na tehnološkom konceptu digitalnog računara. Proces izračunavanja se odvija slično kao neke svakodnevne rutine (zasnovan je na algoritamskom načinu rada), kao što je spremanje hrane korišćenjem recepata, popravljanje kola i slično. Može da se okarakteriše rečenicom “prvo uradi ovo, zatim uradi ono”. Proceduralnom se saopštava računaru KAKO se problem rešava, tj navodi se prećizan niz koraka (algoritam) potreban za rešavanje problema.
Osnovni pojam imperativnih jezika je naredba. Naredbe se grupišu u procedure i izvršavaju se sekvencijalno ukoliko se eksplicitno u programu ne promeni redosled izvršavanja naredbi. Upravljačke strukture su naredbe grananja, naredbe iteracije i naredbe skoka (`goto`). Oznake promenljivih su oznake memorijskih lokacija pa se u naredbama često mešaju oznake lokacija i vrednosti – to izaziva **bočne efekte**.
Primeri jezika: C, Pascal, Basic, Fortran, PL, Algol,...

1.7.2 Objektno-orijentisana paradigma

Ovo je jedna od najpopularnijih programskih paradigmi. Sazrela je početkom 80-ih godina prošlog veka, kao težnja da se jednom napisani softver koristi više puta. Spoljasnji svet se simulira (modeluje) pomoću objekata. Objekti intereaguju međusobno razmenom poruka. Ova paradigma bi mogla da se okarakteriše rečenicom: "Uputi poruku objektima da bi simulirao tok nekog fenomena". Podaci i procedure (funkcije) se učauravaju (enkapsuliraju) u objekte. Koristi se skrivanje podataka da bi se zastitila unutrašnja svojstva objekata. Objeki su grupisani po klasama (klasa predstavlja sablon, koncept, na osnovu kojeg se kreiraju konkretni objekti, tj. instance). Klase su najčešće hijerarhijski organizovane i povezane mehanizmom nasleđivanja.
Primeri jezika: Simula 67, SmallTalk, C++, Eiffel, Java, C#

1.7.3 Funkcionalna paradigma

Rezultat težnje da se drugačije organizuje proces programiranja. Izračunavanja su evaluacije matematičkih funkcija. Zasnovana je na pojmu matematičke funckije i ima formalnu strogo definisanu matematičku osnovu u **lambda računu**. Mogla bi da se okarakteriše rečenicom: "Izračunati vrednost izraza i koristiti je"
Eliminisani su bočni efekti što utiče na lakse razumevanje i predviđanje ponašanja programa – izlazna vrednost funkcije zavisi samo od ulaznih vrednosti argumenata funkcije. Najistaknutiji predstavnik ove paradigme je Lisp. Paradigma je nastala 50-ih i početkom 60-ih godina prošlog veka, stagnirala je u razvoju tokom 70-ih i oživela nastankom programskog jezika Haskell.
Primeri jezika: Lisp, Scheme, Haskell, ML, Scala, OCaml

1.7.4 Logička paradigma

Nastaje kao težnja da se u kreiranju programa koristi isti način razmišljanja kao i pri rešavanju problema u svakodnevnom životu. Paradigma je deklarativna. Opisuju se odnosi između činjenica i pravila u domenu problema, koriste se aksiome, pravila izvođenja i upiti. Logička paradigma se dosta razlikuje od svih ostalih po načinu pristupa rešavanju problema. Nije jednako pogodna za sve oblasti istraživanja, osnovni domen je rešavanje problema veštačke inteligencije. Izvrsavanje programa zasniva se na sistematskom pretrazivanju skupa činjenica uz korišćenje određenih pravila zaključivanja. Zasnovana je na matematičkoj logici, tj na predikatskom računu 1. reda. Zasnovana je na automatskom dokazivanju teorema (metod rezolucije). Mogla bi da se okarakteriše rečenicom: "Odgovori na pitanje kroz trazenje resenja".
Primeri jezika: Prolog (najpoznatiji), ASP, Datalog, CLP, ILOG, Solver, ParLog, LIFE

1.7.5 Komponentna paradigma

Ideja je da se softver sklapa od većih gotovih komponenti, kao što se to radi kod sklapanja elektronskih i tehničkih uređaja. Softverska komponenta je kolekcija delova (metoda i objekata) koji obezbeđuju neku funkcionalnost. Kao i tehničke komponente, i softverske komponente mogu biti proste ili kompleksne, mogu delovati samostalno ili u konjunkciji sa drugim jedinicama.
Komponentna paradigma je nova ili je podparadigma objektno-orijentisane paradigme? Nezavisno od toga, pitanje je: da li treba posebno izučavati komponentno programiranje? Stil programiranja koji je u ekspanziji i treba mu pokloniti posebnu pažnju.
Ideja je da se uprosti proces programiranja i da se jednom kreirane komponente mnogo puta koriste. Komponenta je jedinica funkcionalnosti sa "ugovorenim" interfejsom. Interfejs definiše način na koji se komunicira sa komponentom, i on je u potpunosti odvojen od implementacije. Komponente se međusobno povezuju da bi se kreirao kompleksan softver. Način povezivanja komponenti treba da bude jednostavan, po mogućnosti prevlačenjem i spuštanjem na zeljenu lokaciju. Kreiranje programa se vrsi biranjem komponenti i postavljanjem na pravo mesto, a ne pisanjem "linije za linijom". U okviru komponentnog programiranja, važno je razvojno okruženje koje se koristi, dok sama implementacija komponenti i kod koji se komponentnim programiranjem generise može da bude u različitim programskim jezicima, npr JAVA, C++, C#, ...

1.7.6 Konkurentna paradigma

Konkurentnu paradigmu karakteriše više procesa koji se izvršavaju u istom vremenskom periodu, a koji imaju isti cilj. Postoje različite forme konkurentnosti:

Konkurentnost u užem smislu – *jedan procesor, jedna memorija*

Karakterise je preklapajuće izvršavanje više procesa koji koriste isti procesor i koji komuniciraju preko zajedničke memorije. Ovi procesi modeliraju procese spoljašnjeg sveta koji mogu da se dese konkurentno, na primer kod operativnih sistema.

Paralelno programiranje – *više procesora, jedna memorija*

Ukoliko postoji više procesora sa pristupom jedinstvenoj memoriji, onda je u pitanju paralelno programiranje. Procesi međusobno komuniciraju preko zajedničke memorije. Cilj paralelnog izračunavanja je ubrzanje toka izračunavanja.

Distribuirano programiranje – *više procesora, više memorija*

Ukoliko postoji više procesora od kojih svaki ima svoju memoriju, onda je u pitanju distribuirano programiranje. Procesi međusobno šalju poruke da bi razmenili informacije. Distribuirano izračunavanje čine grupe umreženih računara koje imaju isti čilj kao posao koji izvršavaju. Može se shvatiti kao vrsta paralelnog izračunavanja ali sa drugačijom međusobnom komunikacijom koja nameće nove izazove.

Pisanje konkurentnih programa je značajno teže od pisanja sekvencijalnih programa. Nameće nove probleme, po pitanju sinhronizacije procesa i pristupa zajedničkim podacima. Za osnovne koncepte konkurentnog programiranja potrebno je obezbediti odgovarajuću podršku u programskom jeziku.
Primeri jezika: Ada, Modula, ML, Java ...

1.7.7 Paradigma programiranja ograničenja

U okviru ove paradigme zadaju se relacije između promenljivih u formi nekakvih ograničenja. Ograničenja mogu biti raznih vrsta (logička, linearna,...). Ova ograničenja ne zadaju sekvencu koraka koji treba da se izvrše već osobine resenja koje treba da se pronađe. Paradigma je deklarativna. Jezici za programiranje ograničenja čestu su nadogradnja jezika logičke paradigme, na primer PROLOG-a.
Postoje biblioteke za podršku ovoj vrsti programiranja u okviru imperativnih jezika, npr za jezike C, JAVA, C++, Python.
Primeri jezika: BProlog, OZ, Claire, Curry

1.7.8 Skript paradigma

Skript jezik je programski jezik koji služi za pisanje skriptova. To je spisak (lista) komandi koje mogu biti izvršene u zadatom okruženju bez interakcije sa korisnikom. U prvobitnom obliku pojavljuju se kao komandni jezici operativnih sistema (npr Bash). Skript jezici imaju veliku primenu na Internetu. Skript jezici mogu imati specifican domen primene, ali mogu biti i jezici opšte namene (npr Python).
Skript jezici se ne kompajliraju, već interpretiraju. Cesto se koriste za povezivanje komponenti unutar neke aplikacije. Omogućavaju kratak kod. Najčesće nisu strogo tipizirani. Kod i podaci često mogu zameniti uloge. Nije uvek lako napraviti razliku između skript jezika i drugih programskih jezika.
Skript paradigma je često specifična kombinacija drugih paradigmi, kao što su: objektno-orijentisana, proceduralna, funkcionalna (pa je to razlog što se skript paradigma ne prepoznaje uvek kao posebna paradigma). Skript jezici su u ekspanziji.
Primeri jezika: Unix Shell (sh), JavaScript, PHP, Perl, Python, XSLT, VBScript, Lua, Ruby, ...

1.7.9 Paradigma upitnih jezika

Upitni jezici mogu biti vezani za baze podataka ili za pronalaženje inormacija (information retrieval). Paradigma je deklarativna.

Upitni jezici baza podataka

Oni na osnovu struktuiranih činjenica zadatih u okviru struktuiranih baza podataka daju konkretne odgovore koji zadovoljavaju nekakve trazene uslove. Najpoznatiji predstavnih upitnih jezika za relacione baze podataka je SQL. XQuery je jezik za pretrazivanje XML struktuiranih podataka.
Digresija: Jezici za obelezavanje teksta i programske paradigme:

- Poslednjih decenija veliki procvat doživljavaju jezici za obelezavanje teksta, kao što su: SGML, HTML, XML.
- Jezici za obelezavanje teksta nisu programski jezici pa samim tim i ne mogu da generišu neku programsku paradigmu.
- Međutim, paralelno sa razvojem jezika za obelezavanje (posebno XML), razvijeni su specijalizovani programski jezici za razne obrade koje se odnose na jezike za obelezavanje.
- U takve jezike spadaju: XSLT, XQuery, XLS, ... Ovi jezici se mogu pridružiti raznim paradigmama.

Upitni jezici za pronalaženje informacija

To su upitni jezici koji pronalaze dokumenta koji sadrže informacije relevantne za oblast istraživanja. CQL je jezik za iskazivanje upita za pronalaženje infomacija.

1.7.10 Reaktivna paradigma

Reaktivno programiranje je usmereno na tok podataka u smislu prenošenja izmena prilikom promene podataka. Na primer, u proceduralnom programskom jeziku, `a = b + c` je komanda koja se izvršava dodelom vrednosti promenljivoj `a` na osnovu trenutnih vrednosti promenljivih `b` i `c` i kasnija promena vrednosti `b` ili `c` ne utiče na promenu vrednosti promenljive `a`.Kod reaktivnog programiranja, `a = b + c` ima značenje da svaka promena vrednosti `b` i `c` utiče na izmenu vrednosti promenljive `a`.
Koristi se za programiranje u okviru tabela, npr VisiCalc, Excel, LibreOffice Calc. Jezici za opis hardvera pripadaju ovoj paradigmi, jer se izmena jednog kola u dizajnu propagira na celo kolo – Verilog, VHDL, ...

1.7.11 Vizuelna paradigma

Vrsi modelovanje spoljašnjeg sveta (usko povezana sa objektno-orijentisanom paradigmom). Koriste se grafički elementi (dijagrami) za opis akcija, svojstva i povezanosti sa raznim resursima. Vizuelni jezici su dominantni u fazi dizajniranja programa. Postoje razne vrste dijagrama: dijagram klasa, dijagram korišćenja, dijagram stanja, dijagram aktivnosi, dijagram interakcija, ...
Postoje softverski alati za prevođenje "vizuelnog opisa" u neki programski jezik (samim tim i mašinski jezik). Pogodnija za pravljenje "skica" programa, a ne za detaljan opis. Glavni predstavnik ove paradigme je UML.

2 Skript programiranje

`Unix Shell`	`Bash`	`PHP`	`JavaScript`	`Perl`	`Python`
`XSLT`	`VBScript`	`Lua`	`Ruby`	`...`	

Skript jezici su u ekspanziji. Najviše događanja na polju razvoja programskih jezika sada je u okviru razvoja skript jezika. Tradicionalni programski jezici su namenjeni za razvoj samostalnih aplikacija koje imaju za cilj da prime neku vrstu ulaza i na osnovu nje generišu odgovarajući izlaz. Međutim, upotreba računara često zahteva manipulaciju i koordinaciju različitih programa. Ručno sprovođenje ovih poslova je naporno i sklono greskama. Neki primeri primene:

1: *Sistem za obračun plata.* Obračunavanje vremena sa kartica, papirnih izveštaja i unosa sa tastature, manipulacija bazama podataka, poštovanje pravnih i institucionalnih regulativa, priprema poreza, doprinosa i medicinskog osiguranja, pravljenje papirnih evidencija za arhivu, ...

2: *Fotografija.* Fotograf treba da skine fotografije sa digitalnog fotoaparata, konvertuje u odgovarajući format, rotira slike po potrebi, napravi manje koje su pogodne za brzo razgledanje, indeksira ih po vremenu, temi, napravi bekap na udaljenoj arhivi i ponovo inicijalizuje memoriju, ...

3: *Kreiranje dinamičkih veb stranica.* Autentikacija [1] i autorizacija, komunikacija sa udaljenim uređajem, manipulacija sa slikama, komunikacija sa serverom, čitanje i pisanje HTML-a.

Koordinaciju drugim programima moguće je ostvariti i u tradicionalnim programskim jezicima, kao što su Java ili C, ali to nije lako. Ovi jezici adresiraju efikasnost, lako održavanje, portabilnost i statičko otkrivanje grešaka. Sistem tipova je obično izgrađen oko koncepta kao sto su celobrojne vrednosti fiksnih veličina, brojevi u pokretnom zarezu, karakteri i nizovi. S druge strane, skript jezici imaju za cilj da adresiraju fleksibilnost, brz razvoj, lokalnu prilagodljivost i dinamičke provere. Njihovi sistemi tipova, zbog toga teže da podrže više programske koncepte, kao što su tabele, katalozi, liste, datoteke, ...
Skript jezik je programski jezik koji služi za pisanje skriptova. Skript je spisak (lista) komandi koje mogu biti izvršene u zadatom okruženju bez interakcije sa korisnikom. U prvobitnom obliku pojavljuju se kao komandni jezici operativnih sistema (npr Bash), danas imaju najrazličitije primene. Skript jezici imaju veliku primenu na Internetu (PHP, JavaScript, Dart, Perl, ...). Mogu imati specifičan domen primene, ali mogu biti i jezici opšte namene (npr Python). Pošto se često koriste za povezivanje komponenti, nazivaju se "glue languages".
Skript paradigma je često specifična kombinacija drugih paradigmi, kao što su: objektno-orijentisana, proceduralna, funkcionalna (pa je to razlog sto se skript paradigma ne prepoznaje uvek kao posebna paradigma). Nije uvek lako napraviti razliku između skript jezika i drugih jezika ali ipak imaju određene karakteristike na osnovu kojih se mogu izdvojiti od ostalih programskih jezika.

2.1 Karakteristike skript jezika

Interaktivno korišćenje/serijska obrada

Većina skript jezika omogućava interaktivno korišćenje. Neki skript jezici zahtevaju da se sve komande učitaju pre nego što počne obrada, ali takvi su u manjini (npr Perl). Većina skript jezika je interpretatorskog tipa i mogu da obrađuju liniju po liniju ulaza.

[1] proces određivanja identiteta nekog subjekta, najčešće se odnosi na fizičku osobu. U praksi subjekt daje određene podatke po kojima druga strana može utvrditi da je subjekt upravo taj kojim se predstavlja. Najčešći primjeri su: uz korištenje kartice na bankomatu i upisivanje PIN-a, ili upisivanje (korisničkog) imena i lozinke. Autentikacija se razlikuje od autorizacije; to su dva različita procesa.

Skraćeni zapis

Skraćeni zapis zarad brzog razvoja i interaktivnog korišćenja. Primeri:

Perl, Python, Ruby:

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

Java:

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

Deklaracije i pravila dosega

Promenljive se obično ne deklarišu, postoje jednostavna pravila dosega. U nekim jezicima, sve promenljive su globalne (Perl), u drugim sve su lokalne (php). U Python-u svaka promenljiva je lokalna za blok u kojoj joj je dodeljena vrednost.

Dinamičko tipiziranje

Usled nedostatka deklaracija, većina skript jezika dinamički određuje tipove podataka. "*If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck*". U nekim jezicima (php, Ruby, Python) tip se proverava neposredno pred korišćenje, dok ce se u nekim (npr Perl) tip interpretirati drugačije u različitim kontekstima.

```
$a = "4"
print $a . 3 . "\n" # '.' je nadovezivanje: 43
print $a + 3 . "\n" # '+' je sabiranje: 7
```

Sistemske funkcije

U skript jezicima obično je omogućen lak pristup funkcionalnostima operativnog sistema. Funkcije za ulaz/izlaz, manipulacija fajlovima i direktorijumima, upravljanje procesima, pristup bazama podataka, soketi, interprocesorska komunikacija i sinhronizacija, zaštita i autorizacija, datum i sat, komunikacija preko mreže. I u drugim jezicima je to moguće, ali ne na tako jednostavan način.

Manipulacija stringovima i poklapanje obrazaca

Skript jezici imaju svoje pretke u jezicima za procesiranje teksta i za generisanje izveštaja. Zbog toga imaju bogatu podrsku za rad sa stringovima, za poklapanje obrazaca, pretragu i slično. Ovo se obično bazira na proširenim regularnim izrazima.
Odrediti pravo ime funkciji osobina (tj. naći koju osobinu broja n ispituje naredna funckija u Python-u) `def osobina(n):`

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

Tipovi podataka visokog nivoa

Koriste se i osnovni tipovi podataka ali u okviru same sintakse i semantike skript jezika postoji i direktna podrška za više tipove podataka. Podrška za skupove, rečnike, liste, torke, ... Postoji sakupljač otpadaka.

2.2 Domeni upotrebe skript jezika

Komandni jezici (Shell) – ekspanzija imena fajlova i varijabli, petlje, uslovi, pipe-ovi i redirekcija, funkcije, #! konvencija.

Procesiranje teksta i generisanje izveštaja (Sed, Awk) – orijentisani na rad sa stringovima: pretraga, zamena, ubacivanje, brisanje, poklapanje zagrada, Perl nastao sa idejom da kombinuje Sed, Awk i sh, ali je izrastao u mnogo više od toga.

Matematika (moderni naslednici jezika APL: Maple, Mathematica, Matlab) – podrška numeričkim metodama, simbolčkoj matematici, vizuelizaciji podataka, matematičkom modelovanju

Statistika (S, R - open source) – podrška višedimenzionalnim nizovima, listama, mogu se proširivati infiksnim operatorima, funkcionalno programiranje, ...

Jezici opšte namene (Python, Ruby, ...)

Jezici proširenja (Adobe graficki skup - Illustrator, Photoshop, ... dopuštaju dopune različitim skript jezicima: JavaScript, VisualBasic ili AppleScript; AutoCAD i Flash imaju svoje skript jezike za proširenje, skript jezik Lua se često koristi u razvoju softvera za igrice, Microsoft-ovi alati obično koriste PowerShell, GIMP koristi Scheme, ...) – proširuje korisnost neke aplikacije dozvoljavajući korisniku da daje nove komande koristeći postojeće komande kao gradivne blokove.

Jezici za programiranje veba – skriptovi na strani servera (php, Ruby, PowerShell, Java servlets) i skriptovi na strani klijenta (JavaScript)

3 Programiranje ograničenja - *constraint programming*

Predstavlja savremen pristup rešavanju teških kombinatornih problema. Opšti problem programiranja ograničenja predstavlja se sistemom ograničenja nad upravljačkim (nepoznatim) promenljivama. Zadatak je naći dopustivo rešenje, odnosno odrediti vrednosti promenljivih koje zadovoljavaju sva postavljena ograničenja, ili dokazati da takvo rešenje ne postoji.
Sa aspekta naučnog istraživanja, programiranje ograničenja je multidisciplinarna oblast u kojoj se kombinuju metode i tehnike iz računarskih nauka, veštačke inteligencije, operacionih istraživanja, baza podataka, teorije grafova i logičkog programiranja. Sa aspekta prakse i primena, programiranje ograničenja je softverska tehnologija za deklarativno opisivanje i efektivno rešavanje velikih, prvenstveno kombinatornih, problema.
Osnovna ideja u programiranju ograničenja je da korisnik najpre postavi svoj problem na odgovarajući način, tj pomoću ograničenja, a zatim pronađe rešenje korišćenjem nekog opštenamenskog rešavača ograničenja. Programiranje ograničenja je deklarativno. U njemu se relacije između promenljivih zadaju u obliku različitih ograničenja. Za razliku od imperativne paradigme, gde je postupak pronalaženja rešenja dat u koracima, ovde nije dat postupak već su postavljeni uslovi koje promenljive moraju da ispunjavaju. Od sistema se zatim očekuje da izračuna rešenje, tj da izračuna vrednosti promenljivih koje zadovoljavaju data ograničenja.
Biblioteke programiranja ograničenja mogu imati različite pristupe za rešavanje problema, ali nije neophodno poznavanje algoritma koje ove biblioteke koriste. Rešavanje ograničenja se vrši različitim rešavačima, npr SAT i SMT.
Ograničenja se razlikuju od ograničenja u imperativnoj paradigmi. Na primer, x < y u imperativnoj paradigmi se evaluira u tačno ili netačno, dok u paradigmi ograničenja zadaje relaciju između objekata x i y koja mora da važi. Ograničenja mogu da budu različitih vrsta, na primer, *ograničenja iskazne logike* (A ili B je tačno), *linearna ograničenja* ($x \leq 15$), *ograničenja nad konačnim domenima.*

3.1 Primene ograničenja

Programiranje ograničenja ima primene pre svega u operacionim istraživanjima (tj. u rešavanju kombinatornih i optimizacionih problema). Kriptoaritmetike su zabavne i pogodne za razumevanje programiranja ograničenja, ali **NISU** osnovna primena ove vrste programiranja.
Kriptoaritmetike su matematičke igre u kojima se rešavaju jednacine kod kojih su cifre brojeva zamenjene ogređenim slovima. Primer:

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

```
GREEN + ORANGE = COLORS
MANET + MATISSE + MIRO + MONET + RENOIR = ARTISTS
COMPLEX + LAPLACE = CALCULUS
THIS + IS + VERY = EASY
CROSS + ROADS = DANGER
FATHER + MOTHER = PARENT
WE + WANT + NO + NEW + ATOMIC = WEAPON
EARTH + AIR + FIRE + WATER = NATURE
SATURN + URANUS + NEPTUNE + PLUTO = PLANETS
SEE + YOU = SOON
NO + GUN + NO = HUNT
WHEN + IN + ROME + BE + A = ROMAN
DONT + STOP + THE = DANCE
HERE + THEY + GO = AGAIN
OSAKA + HAIKU + SUSHI = JAPAN
MACHU + PICCHU = INDIAN
SHE + KNOWS + HOW + IT = WORKS
COPY + PASTE + SAVE = TOOLS
```

```
THREE + THREE + ONE = SEVEN
NINE + LESS + TWO = SEVEN
ONE + THREE + FOUR = EIGHT
THREE + THREE + TWO + TWO + ONE = ELEVEN
SIX + SIX + SIX = NINE + NINE
SEVEN + SEVEN + SIX = TWENTY
ONE + ONE + ONE + THREE + THREE + ELEVEN = TWENTY
EIGHT + EIGHT + TWO + ONE + ONE = TWENTY
ELEVEN + NINE + FIVE + FIVE = THIRTY
NINE + SEVEN + SEVEN + SEVEN = THIRTY
TEN + SEVEN + SEVEN + SEVEN + FOUR + FOUR + ONE = FORTY
TEN + TEN + NINE + EIGHT + THREE = FORTY
FOURTEEN + TEN + TEN + SEVEN = FORTYONE
NINETEEN + THIRTEEN + THREE + TWO + TWO + ONE + ONE + ONE = FORTYTWO
FORTY + TEN + TEN = SIXTY
SIXTEEN + TWENTY + TWENTY + TEN + TWO + TWO = SEVENTY
SIXTEEN + TWELVE + TWELVE + TWELVE + NINE + NINE = SEVENTY
TWENTY + TWENTY + THIRTY = SEVENTY
FIFTY + EIGHT + EIGHT + TEN + TWO + TWO = EIGHTY
FIVE + FIVE + TEN + TEN + TEN + TEN + THIRTY = EIGHTY
SIXTY + EIGHT + THREE + NINE + TEN = NINETY
ONE + NINE + TWENTY + THIRTY + THIRTY = NINETY
```

Programiranje ograničenja - primeri:

- Rešiti sistem nejednakosti, npr:

$$
\begin{aligned}
x &\in \{1,\ldots,100\} \\
y &\in \{1,\ldots,100\} \\
x+y &< 100 \\
x &> 10 \\
y &< 50
\end{aligned}
$$

- Rasporediti kraljice na šahovskoj tabli
- Rasporediti topove na šahovskoj tabli
- Odrediti položaje za najmanji broj predajnika tako da pokriva određeni prostor
- Definisati ponašanje semafora tako da protok saobraćaja bude najbolji

3.2 Podrška za programiranje ograničenja

Podrška za ograničenja su ili ugrađena u programski jezik (npr Oz, Kaleidoscope) ili su data preko neke biblioteke. Postoje različite biblioteke za programiranje ograničenja za jezike C, C++, JAVA, Python, za .NET platformu, Ruby. Neke od biblioteka su IBM ILOG CPLEX, Microsoft Z3. Programiranje ograničenja je često raspoloživo u okviru sistema za logičko programiranje. Programiranje ograničenja u logici – *Constraint logic programing* (CPL).

Koreni programiranja ograničenja

Programiranje ograničenja je nastalo u okviru logičkog programiranja (Prolog II, Jaffar i Lassez, 1987). Logičko programiranje i programiranje ograničenja imaju puno zajedničkih osobina. Većina Prolog implementacija uključuje jednu ili više biblioteka za programiranje ograničenja.

B-Prolog	CHIP V5	Ciao	ECLiPSe	SICStus
GNU Prolog	Picat	SWI Prolog	...	

Neke biblioteke – links

- Artelys Lakis (C++, Java, Python)
- Cassowary (C++, Java, JavaScript, Ruby, Smalltalk, Python)
- CHIP V5 (C++, C)
- Choco (Java)
- Cream (Java)
- Disolver (C++)
- Gecode (C++, Python)
- Google or-tools (Python, Java, C++, .NET)
- JaCoP (Java)

- JOpt (Java)
- Numberjack (Python)
- Minion (C++)
- **python-constraint (Python)**
- Z3 (C++, Java, Python, C, C#)

Direktna podrška za programiranje ograničenja

- Claire
- Curry (zasnovan na Haskell-u)
- Kaleidoscope
- Oz
- Wolfram language

3.3 python-constraint

Modul python-constraint podržava programiranje ograničenja na konačnom domenu. Programiranje ograničenja nad konačnim domenom sastoji se od tri dela:

1. Generisanje promenljivih i njihovih domena
2. Generisanje ograničenja nad promenljivama
3. Obeležavanje (*labeling*) – instanciranje promenljivih

U okviru Pythona, rešenja se daju za sve promenljive, tako da je instanciranje podrazumevano.

Primer 1:

```
import constraint

problem = constraint.Problem()

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

resenja = problem.getSolutions()

print resenja

[{'a': 3, 'b': 6}, {'a': 3, 'b': 5},
 {'a': 3, 'b': 4}, {'a': 2, 'b': 6},
 {'a': 2, 'b': 5}, {'a': 2, 'b': 4},
 {'a': 1, 'b': 6}, {'a': 1, 'b': 5},
 {'a': 1, 'b': 4}]
```

Primer 2:

```
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

[{'a': 3, 'b': 5}, {'a': 3, 'b': 4}]
```

Opšta ograničenja

- `AllDifferentConstraint()` – različite vrednosti svih promenljivih
- `AllEqualConstraint()` – iste vrednosti svih promenljivih
- `constraint. MaxSumConstraint(s [, tezine])` – suma vrednosti promenljivih (pomnožena sa težinama) ne prelazi s
- `MinSumConstraint(s [, tezine])` – suma vrednosti promenljivih (pomnožena sa težinama) nije manja od s
- `ExactSumConstraint(s [, tezine])` – suma vrenosti promenljivih (pomnožena sa težinama) je s
- `InSetConstraint(skup)` – vrednosti promenljivih koje se nalaze u skupu `skup`
- `NotInSetConstraint(skup)` – vrednosti promenljivih se ne nalaze u skupu `skup`
- `SomeInSetContraint(skup)` – vrednost nekih promenljivih se nalaze u skupu `skup`
- `SomeNotInSetConstraint(skup)` – vrednosti nekih promenljivih se ne nalaze u skupu `skup`

Primer 3: SEND+MORE=MONEY

```
import constraint

problem = constraint.Problem()

# Definisemo promenljive i njihove vrednosti
# prvo S i M (argument je iterable, i sve iz njega je promenljiva)
problem.addVariables('SM', range(1,10))
problem.addVariables('ENDORY', range(10))

# Definisemo ogranicenje za cifre
def o(s,e,n,d,m,o,r,y):
        if(s*1000 + e*100 + n*10 + d + m*1000 + o*100 + r*10 + e)
                == (10000*m + 1000*o + 100*n + 10*e + y):
        return True

# Dodajemo ograničenja za cifre na svim pozicijama
problem.addConstraint(o, "SENDMORY")

# Dodajemo ogranicenje da su sve cifre razlicite
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'])
```

```
Pokretanje:
python sendmoremoney.py
  9567
 +1085
=10652
```

SEND+MORE=MONEY – Prolog

```
sendmoremoney(Vars) :- Vars = [S,E,N,D,M,O,R,Y],          %generisanje promenljivih
        Vars :: 0..9                                      %definisanje domena
        S #\ 0,                                           %ograničenja
        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).                                   %instanciranje
```

4 Funkcionalna paradigma

4.1 Uvod

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

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

- Formalizam za imperativne jezike – Turingova i URM mašina
- Formalizam za logičke jezike – Logika prvog reda
- Formalizam za funkcionalne jezike – Lambda račun

Funkcionalni jezici su mnogo bliži svom teorijskom modelu nego imperativni jezici, pa je poznavanje lambda izraza važno i ono omogućava fukncionalno programiranje.
Za ekspresivnost vazi:

$$\text{funkcionalni jezici} \Leftrightarrow \text{lambda račun} \Leftrightarrow \text{Tjuringova mašina} \Leftrightarrow \text{imperativni jezici}$$

Svi programi koji se mogu napisati imperativnim stilom, mogu se napisati i funkcionalnim stilom. Lambda račun naglašava pravila za transformaciju izraza i ne zamara se arhitekturom mašine koja to može da ostvari.
Funkcionalna paradigma (ili funkcijska) zasniva se na pojmu matematičkih funkcija – otud potiče i njen naziv. Osnovni cilj ove paradigme je da oponaša matematičke funkcije – rezultat toga je pristup programiranju koji je u osnovi drugačiji od imperativnog programiranja. Osnovna apstrakcija u imperativnim programskim jezicima je apstrakcija kontrole toka (podrutina), u objektno-orijentisanim jezicima je objekat, a u funkcionalnim jezicima to je funkcija. Funkcionalno programiranje je stil koji se zasniva na izračunavanju izraza kombinovanjem fajlova. Osnovne aktivnosti su:

1. *Definisanje funkcije* (pridruživanje imenu funkcije vrednosti izraza pri čemu izraz može sadržati pozive drugih funkcija)
2. *Primena funkcije* (poziv funkcije sa zadatim argumentima)
3. *Kompozicija funkcija* (navođenje niza poziva funkcija) – kreiranje programa

Program u funkcionalnom programiranju je niz definicija i poziva funkcija. Izvršavanje programa je evaluacija funkcija.

Na primer, pretpostavimo da imamo funkciju

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

Prethodnu funkciju možemo koristiti za definisanje novih funkcija. Na primer:

$$max3(x,y,z) = max(max(x,y),z)$$

gde je $max3$ kompozicija funkcija max. Prethodno definisane funkcije možemo kombinovati na razne načine. Na primer, ako treba izračunati $max6(a,b,c,d,e,f)$, to možemo uraditi na sledeće načine:

$$\begin{aligned} &= max(max3(a,b,c), max3(d,e,f)) \\ &= max3(max(a,b), max(c,d), max(e,f)) \\ &= max(max(max(a,b,max(c,d)), max(e,f)) \end{aligned}$$

Da bi se uspešno programiralo, treba:

- ugraditi neke osnovne funkcije u sam programski jezik,
- obezbediti mehanizme za formatiranje novih (kompleksnijih funkcija)
- obezbediti strukture za prezentovanje podataka (strukture koje se koriste da se predstave parametri i vrednosti koje funkcija izračunava)
- formirati biblioteku funkcija koje mogu biti korišćene kasnije

Ukoliko je jezik dobro definisan, broj osnovnih funkcija je relativno mali. Iako su na početku funkcionalni jezici obično bili interpreterski, funkcionalni jezici se sada i kompajliraju.
Osnovna apstrakcija je funkcija i sve se svodi na izračunavanje funkcija. Funkcija je ravnopravna sa ostalim tipovima podataka, može biti povratna vrednost ili parametar druge funkcije. Primer funkcije višeg reda:

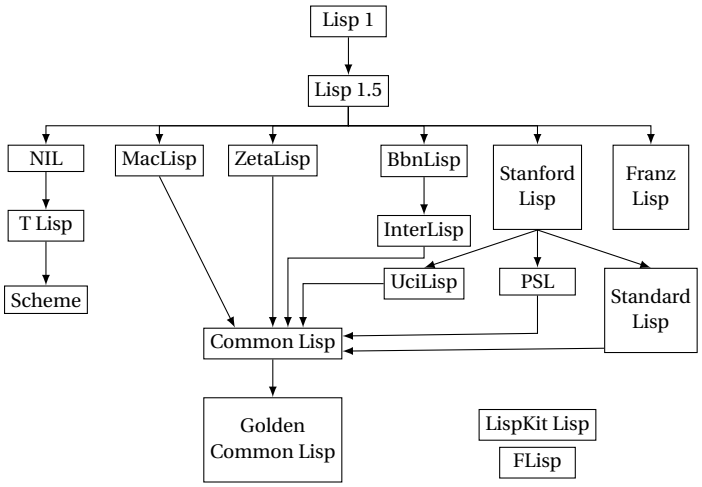
```
duplo f x = f (f x)
```

Postojanje funkcija višeg reda je važna karakteristika funkcionalnih jezika. Posebno važna funkcije višeg reda su `map`, `fold` (`reduce`) i `filter`.

Funkcionalna paradigma nastaje početkom 60-ih godina prošlog veka – najistaknutiji predstavnik paradigme bio je programski jezik Lisp(*LISt Programming – LISP*). Stagnacija u razvoju funkcionalne paradigme javila se 70-ih godina prošlog veka. Tjuringova nagrada za 1977. godinu je dodeljena John Backus-u za njegov rad na razvoju Fortran-a. Svaki dobitnik ove nagrade drži predavanje prilikom formalne dodele, koje se posle štampa u časopisu *Communications of the ACM*. Backus je održao predavanje sa poentom da su čisti funkcionalni programski jezici bolji od imperativnih jer su programi koji se pišu na njima čitljiviji, pouzdaniji i verovatnije ispravni. Suština njegove argumentacije bila je da su funkcionalni jezici lakši za razumevanje, i za vreme i nakon razvoja programa, najviše zbog toga što je vrednost izraza nezavisna od konteksta u kojem se izraz nalazi.
Osnovna karakteristika **čistih** funkcionalnih jezika je *referentna prozirnost* (transparentnost) što kao posledicu ima nepostojanje propratnih (bočnih) efekata. U okviru svog predavanja, Backus je koristio svoj funkcionalni jezik FP kao podršku argumentaciji koju izlaze. Iako jezik FP nije zaživeo, njegovo izlaganje je motivisalo debate i nova istraživanja funkcionalnih programskih jezika.

1924. Kombinatorna logika – Schonfinkel and Curry
1930. Lambda račun – Alonzo Church
1959. Lisp – McCarthy, MIT
1964. SECD – apstraktna mašina i jezik ISWIM (Landin) – prekretnica za razvoj kompajlera i praktičnih rešenja
1977. FP – Backus
1978. Scheme – Sussman and Steel
1978. ML – Robin Milner, University of Edinburgh
1985. Miranda – David Turner (ML)
1986. Erlang (Ericsson) – 1998 open source
1990. SML – Robin Milner, University of Edinburgh
1990. Haskell – Haskell Committee (ML, Miranda), Haskell 98, Haskell 2010
1996. OCaml – Xavier Leroy, Jerome Vouillon, Damien Doligez, Didier Remy (ML)
2002. F# – Don Syme, Microsoft Research, Cambridge (ML)
2003. Scala – Martin Odersky, EPFL
2012. Elixir – Jose Valim

Slika 1: Razvoj Lisp-a



Podrška funkcionalnim konceptima: 2011 C++, JAVA, skript jezici(Python).
Razvojem i zrelošću jezika kao što su ML, Haskell, OCaml, F# i Scala raste interesovanje i upotreba funkcionalnih jezika. Ovi jezici se sada koriste npr u domenima obrade baza podataka, finansijskog modelovanja, statičke analize i bioinformatike, a broj domena upotrebe raste. Funkcionalni jezici su pogodni za paralelizaciju, što ih posebno čini popularnim za paralelno i distribuirano programiranje.

4.2 Svojstva funkcionalnih jezika

4.2.1 Funkcije - *first class citizen*

U okviru programskog jezika za neki gradivni element se kaže da je *građanin prvog reda* ako u okviru jezika ne postoje restrikcije po pitanju njegovog kreiranja i korišćenja. Građani prvog reda se mogu čuvati u promenljivama, prosleđivati funkcijama, kreirati u okviru funkcija i vratiti kao povratna vrednost funkcija. U dinamički tipiziranim jezicima (tj gde se tipovi određuju u fazi izvršavanja programa), građani prvog reda imaju tip koji se proverava u fazi izvršavanja. U funkcionalnom programiranju **funkcije su građani prvog reda**.

Primena funkcije je izračunavanje vrednosti za neke konkretne argumente. Određena je uparivanjem imena funkcije sa određenim elementom iz domena. Rezultat se izračunava evaluiranjem izraza koji definiše preslikavanje funkcije. **Funkcije višeg reda** (funkcijske forme) imaju jednu ili više funkcija kao parametre ili imaju funkciju kao rezultat, ili oba.

α funkcija (`apply to all` ili `map`)

Ova funkcija ima jedan parametar koji je funkcija, i kada se primeni na listu parametara, kao rezultat se dobija lista vrednosti koja se izračunava primenom funkcije parametra na listu parametra. Na primer:

$$f(x) = 2 * x$$
$$\alpha(f,(1,2,3)) = (2,4,6)$$

ϕ funkcija (`filter`)

Ova funkcija ima jedan parametar koji je funkcija (povratna vrednost true/false), i kada se primeni na listu parametara, kao rezultat se dobija lista vrednosti za koje je ispunjen uslov funkcije. Na primer:

$$f(x) = odd(x)$$
$$\phi(f,(1,2,3)) = (1,3)$$

ρ funkcija (`reduce`)

Ova funkcija ima jedan parametar koji je funkcija, i kada se primeni na listu parametara, kao rezultat se dobija odgovarajuća vrednost. Na primer:

$$f(x,y) = x + y$$
$$\rho(f,(1,2,3)) = 6$$

Matematička funkcija je preslikavanje elemenata jednog skupa (domena) u elemente drugog skupa (kodomena). Definicija funkcije uključuje zadavanje domena, kodomena i preslikavanja. Preslikavanje može da bude zadato izrazom ili tabelom. Funkcije se primenjuju na pojedinačne elemente iz domena, koji se zadaju kao parametri (argumenti) funkcija. Domen može da bude Dekartov proizvod različitih skupova, tj funkcija može da ima više od jednog parametra.
Definicija funkcije obično uključuje ime funkcije za kojom sledi lista parametara u zagradama, a zatim i izraz koji zadaje preslikavanje. Na primer: $kub(x) \equiv x * x * x$ za svaki realan broj x. U okviru ove definicije, domen i kodomen su skupovi realnih brojeva, dok se znak $\equiv$ koristi sa značenjem "se definise kao". Parametar x može da bude bilo koji član domena, ali se fiksira kada god se evaluira u okviru izraza funkcije. Kod striktnih jezika, za vreme evaluacije, preslikavanje funkcije ne sadrži nevezane parametre. Svako pojavljivanje parametra se vezuje sa nekom vrednošću i konstantno je za vreme evaluacije. Na primer, za $kub(x)$ imamo $kub(2.0) = 2.0 * 2.0 * 2.0 = 8$. Parametar x je vezan sa vrednošću 2.0 tokom evaluacije i ne postoje nevezani parametri. Dalje, x je konstanta i njena vrednost se ne može promeniti za vreme evaluacije.

Primeri definisanja matematičkih funkcija:

	Funkcija abs	**Funkcija faktorijel**
Domen:	svi realni brojevi	prirodni brojevi
Kodomen:	pozitivni realni brojev	prirodni brojevi
Preslikavanje:	$abs(x) \equiv \begin{cases} x & ,x \geq 0 \\ -x & ,x < 0 \end{cases}$	$n! \equiv \begin{cases} n*(n-1)! & ,n > 0 \\ 1 & ,n = 0 \end{cases}$

Jedna od osnovnih karakteristika matematičkih funkcija je da se izračunavanje preslikavanja kontroliše **rekurzijom** i **kondicionalnim izrazima**, a ne sa sekvencom i iteracijom (kao što je to kod imperativnih jezika).
Matematičke funkcije: **vrednosti preslikavanja elemenata iz domena su uvek isti elementi iz kodomena** (jer ne postoje propratni (bočni) efekti i funkcije ne zavise od drugih spoljašnjih promenljivih).
Kod imperativnih jezika vrednost funkcije može da zavisi od tekućih vrednosti različitih globlanih ili nelokalnih promenljivih.

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

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

4.2.2 Stanje programa i referentna prozirnost

Stanje programa je osnovna karakteristika imperativnih jezika. Njega čine sve vrednosti u memoriji kojima program u toku izvršavanja ima pristup. Imperativni jezici imaju implicitno stanje i izvršavanje programa se svodi na postepeno menjanje tog stanja izvođenjem pojedinačnih naredbi. Ovo stanje se predstavlja programskim promenljivama.

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

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

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

```
/* Globalna promenljiva y */
int y;

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

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

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

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

Funkcionalni jezici nemaju implicitno stanje, čime je razumevanje efekta rada funkcije značajno olakšano. Izvođenje programa se svodi na evaluaciju izraza i to bez stanja. Posebno, nepostojanje naredbe dodele i promenljivih u imperativnom smislu (tj nepostojanje promeljivih koje menjaju stanje) ima za posledicu da **iterativne konstrukcije nisu moguće** pa se ponavljanje ostvaruje kroz rekurziju.

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

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

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

Rekurzija je u imperativnim jezicima manje prirodna. U funkcionalnim jezicima je prirodna i zapravo je i jedino rešenje. Neki problemi se ne mogu rešiti bez stanja (ili se mogu rešiti komplikovano). Nepostojanje implicitnog stanja nije nedostatak već se upotreba stanja može ostvariti na drugi način, korišćenjem eksplicitnog stanja. Ono se može "napraviti" po potrebi i koristiti. Na primer:

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

Referentna prozirnost (transparentnost, providnost)
Vrednost izraza je svuda jedinstveno određena: ako se na dva mesta referencira na isti izraz, onda je vrednost ta dva izraza ista. Na primer, u sledećem kodu, svako pojavljivanje izraza x u programu možemo zameniti sa `fact 5`.

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

Svojstvo referentne prozirnosti govori da je **redosled naredbi nebitan**. U funkcionalnom programiranju neobavezan je eksplicitni redosled navođenja funkcija – funkcije možemo kombinovati na razne načine, bitno je da se definise izraz

koji predstavlja rešenje problema. To naravno nije slučaj kod imperativnih jeika gde je biran redosled izvođenja operacija, x se nakon inicijalizacije može promeniti, pa stoga imperativni jezici nisu referentno prozirni. (Naredba dodele ima propratni efekat, a ona je u osnovi imperativnog programiranja.) Propratni efekat je svaka ona promena implicitnog stanja koja narušava prozirnost programa. Propratni efekti nemaju smisla sa stanovišta matematičkih izračunavanja. Referentna prozirnost ima niz pogodnosti i korisna je u drugim paradigmama.
Programi sa referentnom prozirnošću su **formalno koncizni**, prikladni za **formalnu verifikaciju, manje podložni greškama** i lakše ih je transformisati, optimizovati i **paralelizovati**. Paralelizacija je moguća zbog referentne prozirnosti jer se mogu delovi izraza sračunati nezavisno, a onda združiti naknadno. Međutim, referentna prozirnost ima cenu.
Ukoliko želimo da imamo u potpunosti referentnu prozirnost, ne smemo dopustiti nikakve propratne efekte. U praksi je to veoma teško jer postoje neki algoritmi koji se suštinski temelje na promeni stanja (npr. random) dok neke funkcije postoje samo zbog svojih propratnih efekata (npr. scanf, tj I/O funkcije). Zbog toga, većina funkcionalnih programskih jezika **dopušta kontrolisane propratne efekte** (tj imperativnost je prisutna u većoj ili manjoj meri i na različite načine). U zavisnosti od prisutnosti imperativnih osobina, postoji podela na čiste funkcionalne jezike (bez propratnih efekata) i na one koji nisu čisti.
Čisto funkcionalni jezici ne dopuštaju bas nikakve propratne efekte, takvi jezici koriste dodatne mehanizme kako bi omogućili izračunavanje sa stanjem, a istovremeno zadržali referentnu prozirnost. Jako je mali broj čistih funkcionalnih jezika (Haskell, Clean, Miranda). Većina funkcionalnih jezika uključuju naredne imperativne osobine: promenljive (mutable variables) i konstrukte koji se ponašaju kao naredbe dodele.

- Funkcionalni jezici su jezici koji podržavaju i ohrabruju funkionalni stil programiranja, npr SML, cisti jezici Haskell, Clean, Miranda
- Moderni višeparadigmatski programski jezici su Common Lisp, OCaml, Scala, Python, Ruby, F#, Clojure...
- Neki imperativni jezici eksplicitno podržavaju neke funkcijske koncepte (C#, Java), dok su u drugima oni ostvarivi (C, C++).

Iako mnogi imperativni jezici podržavaju osnovne koncepte funkcionalnog programiranja, često je funkcionalni podskup takvih jezika vrlo slab i nećemo ih nazivati funkcionalnim jezicima.

4.2.3 Strukture podataka i tipovi podataka

Funkcionlani jezici imaju osnovne tipove podataka (celobrojne vrednosti, realne vrednosti, logičke vrednosti, stringovi). Osnovna struktura podataka koja se javlja u svim funkcionalnim jezicima je *lista*. Funkcionalni jezici podržavaju i *torke* koje mogu da sadrže elemente različitih tipova.
Često: torke zauzimaju kontinualni prostor u memoriji (slično kao nizovi), dok su liste implementirane preko povezanih listi – izbor odgovarajuće strukture zavisi od problema (da li se često obilazi ili se često radi dodavanje i brisanje).
Zaključivanje tipova može biti statičko (u fazi kompilacije) i dinamičko (u fazi izvrsavanja). Statičko zaključivanje tipova je manje fleksibilno ali efikasnije, dok je dinamičko fleksibilnije ali manje efikasno. Funkcionalni jezici mogu imati statičko (npr Haskell) ili dinamičko zaključivanje tipova (npr Elixir). Funkcionalni programski jezici su najčešće jako tipizirani jezici – svi tipovi moraju da se poklapaju i nema implicitnih konverzija. S druge strane, nije neophodno navoditi sve tipove – kompajler je u stanju da često automatski sam zaključi tipove. Kod koji se piše najčešće je sam po sebi polimorfan, a zaključuju se najopštiji mogući tipovi na osnovu tipskih razreda.*Tipski razredi* – tip sa jednakošću, tipovi sa uređenjem, numerički tipovi, celobrojni tipovi, realni tipovi, ...

4.2.4 Sintaksa i semantika

Prvi funkcionalni jezik, LISP, koristi sintaksu koja je veoma drugačija od sintakse koja se koristi u imperativnim jezicima. Najnoviji funkcionalni jezici koriste sintaksu koja je slična sintaksi imperativnih jezika.
Pattern matching
Comprehensions
Semantika programskog jezika opisuje proces izvršavanja programa na tom jeziku. Ona može da se opiše formalno i neformalno. Uloga semantike:

- programer može da razume kako se program izvršava pre njegovog pokretanja kao i šta mora da obezbedi prilikom kreiranja kompilatora
- razumevanje karakteristika programskog jezika
- dokazivanje svojstava određenog programskog jezika

Mogu se razmatrati različita semantička svojstva jezika. Postoje *striktna* i *nestriktna* semantika (izračunavanje vođeno potrebama).
Izraz je **striktan** ako nema vrednost kad bar jedan od njegovih operanada nema vrednost. Semantika je striktna ako je svaki izraz tog jezika striktan. Kod striktne semantike, prvo se izračunaju vrednosti svih operanada, pa se onda izračunava vrednost izraza. Tehnika prenosa parametara koja podržava ovu semantiku je *prenos parametara po vrednosti*.

Većina funkcionalnih jezika ima striktnu semantiku, npr Lisp, OCaml, Scala.

Izraz je **nestriktan** kad ima vrednost čak i ako neki od njegovih operanada nema vrednost. Semantika je nestriktna ukoliko se dozvoli da izraz ima vrednost i ako neki argument izraza nema vrednost (tj ukoliko se dozvole nestriktni izrazi). Kod nestriktne semantike izračunavanje operanada, tj argumenata se odlaže sve dok te vrednosti ne budu neophodne. Ta strategina je poznata kao *zadržano lenjo izračunavanje* (*lazy evaluation*), a prenos parametara je *po potrebi* (*call by need*). Nestriktna semantika omogućava kreiranje beskonačnih struktura i izraza.
Miranda i Haskell imaju nestriktnu semantiku.
Od konkretnog izraza zavisi da li će i na osnovu koliko poznatih argumenata biti izračunata njegova vrednost. Na primer: `a & b` ima vrednost `False` ako `a` ima vrednost `False`, tj. nije važna vrednost izraza `b` (nestriktna semantika). U striktnoj semantici, ako vrednost za `b` nije poznata, vrednost konjunkcije se ne može izračunati.

4.2.5 Prednosti i mane funkcionalnog programiranja

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

- **Stanje i propratni efekti** – za velike programe teško je pratiti stanje i razumeti propratne efekte, lakše je kada imamo uvek isto ponašanje funkcija, ali svet koji nas okružuje je pun promena i različitih stanja i nije prirodno da koncepti jezika budu u suprotnosti sa domenom koji se modeluje.
- **Paralelno programiranje** – jednostavno i bezbedno konkurentno programiranje je posledica referentne prozirnosti, međutim, rad sa podacima koji se ne menjaju dovodi do mogućeg rada sa bajatim podacima, dok rad sa podacima koji se menjaju jeste komplikovaniji i zahteva kodiranje kompleksne logike, ali omogućava rad sa svežim podacima (sto je bitnije?).
- **Stil programiranja** – programi su često kraći i lakši za čitanje, treba znati pročitati funkcionalni kod.
- **Produktivnost programera** – produktivnost je veća, ali produktivnost mora da bude mnogo veća da bi se opravdao trošak zapošljavanja programera koji znaju funkcionalno programiranje; takođe, većina programera ne gradi nove sisteme već rade na održavanju starih sistema koji su pisani u drugim (imperativnim) jezicima.
- Za funkcionalne programe lakše je konstruisati matematički dokaz ispravnosti.
- Stil programiranja nameće razbijanje koda u manje delove koji imaju jaku koheziju i izraženu modularnost.
- Pisanje manjih funkcija omogućava veću čitljivost koda (?) i lakšu proveru grešaka.
- Testiranje i debagovanje je jednostavnije:
  - Testiranje je jednostavnije jer je svaka funkcija potencijalni kandidat za unit testove. Funkcije ne zavise od stanja sistema što olakšava sintezu test primera i proveru da li je izlaz odgovarajući.
  - Debagovanje je jednostavnije jer su funkcije uglavnom male i jasno specijalizovane. Kada program ne radi, svaka funkcija je interfejs koji se može proveriti tako da se brzo izoluje koja funkcija je odgovorna za grešku.
- Mogu se jednostavnije graditi biblioteke funkcija.
- Efikasnost se dugo navodila kao mana, ali zapravo efikasnost odavno nije problematična
- Debagovanje ipak može da bude komplikovano(kada je nestriktna semantika u pitanju)
- Nije pogodno za svaku vrstu problema (mada se i ovde sve brže brišu granice)
- Često se navodi da je funkcionalno programiranje teško za učenje (diskutabilno!)
- U radu *"Why no one uses functional languages?"*, Philip Wadler 1998. godine diskutuje razne mane funkcionalnog programiranja, ali je većina ovih mana u međuvremenu uklonjena, iako se i dalje (neosnovano) često pominju. Tu se pominju: kompatibilnost sa drugim jezicima (u složenim sistemima se kombinuju različite komponente), biblioteke, portabilnost, dostupnost i podrška, nedostatak profajlera i dibagera, period učenja, popularnost. On već tada navodi da efikasnost nije ključni razlog.
- Ono što je svakako izvesno je da se velike pare ulazu u razvoj i podršku funkcionalnog stila programiranja (npr Scala).
- Postoje domeni u kojima treba koristiti funkcionalno programiranje i domeni za koje funkcionalno programiranje nije elegantno rešenje
- Postoji velika cena prelaska na funkcionalno programiranje i za to treba vremena.
- Pomeranje prema programiranju koje je više u funkcionalnom stilu mora da ide polako i postepeno.

4.3 Lambda račun

4.3.1 Istorijski pregled

Formalni model definisanja algoritma

Lambda račun (λ-*calculus*) je formalni model izračunljivosti funkcija. Alonzo Church 1930. godina. Lambda račun se zasniva na apstrakciji i primeni funkcija korišćenjem vezivanja i supstitucije (zamene). Funkcije se tretiraju kao izrazi koji se postepeno transformišu do rešenja, tj. funkcija definiše algoritam. Lambda račun je fomalni model definisanja algoritama.

Prvi funkcionalni jezik

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

λ-račun sa i bez tipova

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

4.3.2 Sintaksa

Zadatak definisanja funkcije može se razdvojiti od zadatka imenovanja funkcije. Na primer:

Funkcija $sum(x,y) = x+y$ moze da se definiše i bez njenog imenovanja kao funkcija koje promenljive x i y preslikava u njihov zbir, tj $(x,y) \rightarrow x+y$.
Funkcija $id(x) = x$ moze da se definiše kao $x \rightarrow x$

λ-račun daje osnove za definisanje bezimenih funkcija. λ-izraz definiše parametre i preslikavanje funkcije, ne i ime funkcije, pa se takođe zove *anonimna funkcija* (jos sinonima: bezimena funkcija, funkcijska apstrakcija).

Sintaksa λ-izraza: $\lambda promenljiva.telo$
sa značenjem $promenljiva \rightarrow telo$

$\lambda x.x+1 \quad , x \rightarrow x+1$
$\lambda x.x \quad , x \rightarrow x$
$\lambda x.x*x+3 \; , x \rightarrow x \cdot x+3$

λ-izraz se moze primenjivati na druge izraze.
Sintaksa: $(\lambda promenljiva.telo)izraz$
- intuitivno, primena odgovara pozivu funkcije.

$(\lambda x.x+1)5$
$(\lambda x.x*x+3)((\lambda x.x+1)5)$

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

Promenljive Promenljiva x je validni λ term

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

λ-primena Ako su t i s λ termovi, onda je ts λ term

λ termovi se mogu konstruisati samo konačnom primenom prethodnih pravila.
Prirodni brojevi se mogu definisati korišćenjem ove definicije λ-računa. Ukoliko λ-račun ne uključuje konstante u definiciji, onda se naziva **čist**. Radi jednostavnosti, numerali se često podrazumevaju i koriste već u okviru same definicije λ termova:

<con> ::= konstanta <id> ::= identifikator <exp>
::= <id> | λ <id>.<exp> | <exp><exp> | (<exp>) | <con>

Ukoliko λ-račun ukljucuje konstante u difiniciji, onda se naziva **primenjen**. Isto tako, korišćenjem osnovnog λ-računa mogu se definisati i aritmetičke funkcije. Radi jednostavnosti, u okviru λ termova koristimo aritmetičke funkcije imenovane na standardni nacin, kao sto su $+, -, *$ i slično.

U okviru lambda izraza zagrade su važne. Na primer, termovi $\lambda x.((\lambda x.x+1)x)$ i $(\lambda x.(\lambda x.x+1))x$ su različiti termovi. Da bi se smanjila upotreba zagrada, postoje pravila asocijativnosti za *primenu* i *apstrakciju*. **Primena** funkcije je **levo** asocijativna, tj umesto $(e_1e_2)e_3$ možemo kraće da pišemo $e_1e_2e_3$. **Apstrakcija** je **desno** asocijativna, tj umesto $\lambda x.(e_1e_2)$ pišemo skraćeno $\lambda x.e_1e_2$. I sekvenca apstrakcija može da se skrati, npr $\lambda x.\lambda y.\lambda z.e$ se skraćeno zapisuje kao $\lambda xyz.e$.

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

4.3.3 Slobodne i vezane promenljive

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

Promenljive Slobodna promenljiva terma x je samo x

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

Primena Skup slobodnih promenljivih terma ts je unija skupova slobodnih promenljivih terma t i terma s

Na primer, term $\lambda x.x$ nema slobodnih promenljivih, dok term $\lambda x.x*y$ ima slobodnu promenljivu y.
α ekvivalentnost – definise se za λ termove, sa ciljem da se uhvati intuicija da izbor imena vezane promenljive u λ računu nije vazan.

Termovi $\lambda x.x$ i $\lambda y.y$ su α-ekvivalentni jer oba predstavljaju istu funkciju, tj. identitet.
Termovi $\lambda x.x$ i $\lambda x.y$ nisu α-ekvivalentni jer prvi predstavlja funkciju identiteta, a drugi konstantnu funkciju.
S druge strane, termovi x i y nisu α-ekvivalentni jer nisu vezani u okviru λ apstrakcije.
Zaokružiti slovo ispred α-ekvivalentnih termova:

(a) x i y

(b) $\lambda z.z*y-1$ i $\lambda x.x*z-1$

(c) $\lambda a.a*y-1$ i $\lambda b.b*z-1$

(d) $\lambda ij.i-j*3$ i $\lambda mn.m-n*3$

(e) $\lambda k.5+k/2$ i $\lambda h.5+h/2$

4.4 Redukcije

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

δ redukcija

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

α redukcija ili preimenovanje

Dozvoljava da se promene imena vezanim promenljivama. Na primer, α redukcija izraza $\lambda x.x$ moze da bude u $\lambda y.y$. Termovi koji se razlikuju samo po α konverziji su α *ekvivalentni*. Na primer, $\lambda xy.x=\lambda zy.z=\lambda ay.a$... ($y$ nije vezana promenljiva i za nju ne možemo da vršimo preimenovanje. α preimenovanje je nekada neophodno da bi se izvrsila β redukcija.
α redukcija nije u potpunosti trivijalna, treba voditi računa. Na primer, $\lambda x.\lambda x.x$ može da se svede na $\lambda y.\lambda x.x$ ali ne i na $\lambda y.\lambda x.y$. Ukoliko naredne funkcije primenimo na broj 3

$\lambda x.\lambda x.x-$ dobijemo preslikavanje kojim se 3 se preslikava u funkciju identiteta

$\lambda y.\lambda x.x-$ dobijemo preslikavanje kojim se 3 se preslikava u funkciju identiteta

$\lambda y.\lambda x.y-$ dobijemo preslikavanje kojim se 3 se preslikava u fukciju konstantnog preslikavanja u broj 3

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

β redukcija – primena funkcije

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

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

β redukcija u telu λ izraza formalni argument zamenjuje aktuelnim argumentom i vraća telo funkcije (dakle svako pojavljivanje promenljive u telu se zamenjuje datim izrazom). Na primer:

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

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

Visestruka primena β funkcije (oznaka $\twoheadrightarrow_\beta$), na primer:

$$(\lambda x.x*x+3)((\lambda x.x+1)5) \twoheadrightarrow_\beta 39$$

Primenjujemo β redukciju sve dok možemo – to odgovara izračunavanju vrednosti funkcije.

Prethodni primeri su bili jednostavni jer je primena obuhvatala konstante i jednostavne λ izraze. Da bi se izmene vršile na ispravan način, potrebno je precizno definisati pojam zamene – supstitucije. Ukoliko postoji problem kolizije imena, potrebno je uraditi α preimenovanje kako bi se izbegla kolizija. Na primer, $\overline{\lambda x.((\lambda y.\lambda x.x+y)x)}5$. Supstitucija se definiše rekurzivno po strukturi terma.
Supstitucija $[I/P]T$ je proces zamene svih slobodnih pojavljivanja promenljive P u telu λ izraza T izrazom I na sledeći način (x i y su promenljive, a M i $N\lambda$ izrazi):

Promenljive $[N/x]x = NN/xy = y$ pri cemu je $x \neq y$

Primena $[N/x](M_1M_2) = ([N/x])(M_1)([N/x])(M_2)$

Apstrakcija $[N/x](\lambda x.M) = \lambda x.M[N/x](\lambda y.M) = \lambda y.([N/x]M)$ ukoliko je $x \neq y$ i y ne pripada skupu promenljivih za N

β redukcija se definiše preko naredne supstitucije:

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

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

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

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

4.5 Funkcije všeg reda i funkcije sa više argumenata

Funkcije višeg reda su funkcije koje kao argument ili kao povratnu vrednost imaju funkciju. Funkcija koja očekuje funkciju tu funkciju će primeniti negde u okviru svog tela. Na primer: $\lambda x.(x2)+1-$ λ izraz kod kojeg x primenimo na dvojku (dakle x je nekakva funkcija), a onda na to dodamo broj 1

$$(\lambda x.(x2)+1)(\lambda x.x+1) \rightarrow_\beta (\lambda x.x+1)2+1 \twoheadrightarrow_\beta 4(\lambda x.(x2)+1)(\lambda x.x) \twoheadrightarrow_\beta 3$$

Funkcija koja vraća funkciju će u svom telu sadržati drugi lambda izraz. Na primer:

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

Dakle, kada se početni lambda izraz primeni na 5, onda 5 ulazi u izraz na mesto x, i dobijamo novu funkciju kao rezultat. Kako je ovakva upotreba česta, uvedena je sintaksna skraćenica koju smo ranije pominjali, tj $\lambda x.(\lambda y.2*y+x)$ pišemo kao $\lambda xy.2*y+x$. Dakle umesto zagrada, nižemo argumente.
Lambda izrazi ograničeni su samo na jedan argument. Kako definisati funkcije sa više argumenata, npr $f(x,y) = x+y$? Bilo koja funkcija sa više argumenata može se definisati pomoću funkcije sa samo jednim argumentom (rezultat iz 1924. godine). Postupak se naziva *Curryjev postupak* (po američkom matematičaru Haskell Brooks Curry). Ideja je da funkcija koja treba da uzme dva argumenta, prvo uzme jedan argument, od njega napravi funkciju koja će onda da uzme drugi argument.
Funkcija oblika $f(x_1,x_2,...,x_n) = telo$ u lambda računu se definiše kao $\lambda x_1.(\lambda x_2.(...(\lambda x_n.telo)))$ odnosno skraćeno kao $\lambda x_1x_2...x_n.telo$. Na primer, $f(x,y) = x+y$ se definise kao $\lambda xy.x+y$

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

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

1. $(\lambda k.k*k+1)((\lambda m.m+1)2)$
2. $(\lambda k.k4)(\lambda y.y-2)$
3. $((\lambda kmn.k*m+n)2)3$

4.6 Normalni oblik

Višestrukom β redukcijom izračunavamo vrednost izraza i zaustavljamo se tek onda kada dalja β redukcija nije moguća. Tako dobijen λ izraz naziva se normalni oblik i on intuitivno odgovara vrednosti polaznog izraza. Bilo bi dobro da je normalni oblik jedinstven i da ga mi mozemo uvek pronaći. Ali nemaju svi izrazi svoj normalni oblik, na primer, $(\lambda x.xx)(\lambda x.xx)$. Za neke izraze mogu da postoje različite mogućnosti primene β redukcije, primer :

$$(\lambda x.5*x)((\lambda x.x+1)2) \rightarrow_\beta (\lambda x.5*x)(2+1)$$

$$(\lambda x.5*x)((\lambda x.x+1)2) \rightarrow_\beta 5*((\lambda x.x+1)2)$$

Postavlja se pitanje da li je važno kojim se putem krene?

Teorema 4.1 (Church–Rosser (svojstvo konfluentnosti)) *Ako se λ izraz može svesti na dva različita izraza M i N, onda postoji treći izraz Z do kojeg se može doći iz M i iz N.*

Posledica teoreme je da svaki λ izraz ima najvise jedan normalni oblik (dakle, ako postoji, on je jedinstven). To znači da nije bitno kojim putem se dolazi do normalnog oblika, ukoliko do normalnog oblika dođemo, znamo da smo došli do jedinstvenog normalnog oblika. Kako da dođemo do normalnog oblika?

Aplikativni poredak

β redukcijom uvek redukovati najdublji najlevlji izraz. Na primer:

$$(\lambda x.5*x)(2+1) \rightarrow_\beta (\lambda x.5*x)3 \rightarrow_\beta 5*3 \rightarrow 15$$

Ovo odgovara pozivu po vrednosti (call-by-value) – izračunavamo vrednost argumenta i tek kada ga izračunamo šaljemo ga u funkciju i funkcija dočeka u svom telu izračunati argument.

Normalni poredak

β redukcijom uvek redukovati najlevlji izraz. Na primer:

$$(\lambda x.5*x)(2+1) \rightarrow_\beta 5*(2+1) \rightarrow 5*3 \rightarrow 15$$

Ovo odgovara evaluaciji po imenu (call-by-name) ili evaluaciji po potrebi (call-by-need).

Teorema 4.2 (Teorema standardizacije) *Ako je Z normalni oblik izraza E, onda postoji niz redukcija u normalnom poretku koji vode od E do Z.*

Lenja evaluacija

Normalnim poretkom redukcija ostvaruje se lenja evaluacija – izrazi se evaluiraju samo ukoliko su potrebni. Lenjom evaluacijom se izbegavaju nepotrebna izracunavanja. Na primer:

Aplikativni poredak: $(\lambda x.1)(12345*54321) \rightarrow_\beta (\lambda x.1)670592745 \rightarrow_\beta 1$

Normalni poredak: $(\lambda x.1)(12345*54321) \rightarrow_\beta 1$

Dakle, evaluiramo izraz tek onda kad nam njegova vrednost treba. Lenja evaluacija nam garantuje završetak izračunavanja uvek kada je to moguće. Na primer:

Aplikativni poredak: $(\lambda x.1)((\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \ldots$ (ne završava)

Normalni poredak: $(\lambda x.1)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta 1$

Postoje tehnike koje primenjuju kompajleri, a koje obezbeđuju da se izračunavanja ne ponavljaju, ovo je važno sa stanovišta efikasnosti izračunavanja.

Za $(\lambda x.x+x)(12345*54321) \rightarrow_\beta (12345*54321)+(12345*54321)$ ne bi bilo dobro dva puta nezavisno računati proizvod $(12345*54321)$ već je potrebno to samo jednom uraditi, i za to postoje *tehnike redukcije grafova.*

5 Haskell

Razvoj
Haskell Brooks Curry(1900-1982) logičar i matematičar.

1987	međunarodni odbor počinje sa dizajnom novog, zajedničkog funkcionalnog jezika
1990	odbor najavljuje specifikaciju Haskell 1.0
1990-1997	4 izmene standarda
1998	Haskell 98
2010	Haskell Prime, tj Haskell 2010

Karakteristike Haskella (Zvanicni sajt)
Haskell je *čist funkcionalni jezik*. Zasniva se na lenjoj evaluaciji – izbegavaju se nepotrebna izračunavanja. Ima moćni sistem tipova – automatsko zaključivanje tipova. Strogo je tipiziran jezik (svi tipovi moraju da se poklapaju, *nema implicitnih konverzija*. Ima podršku za paralelno i konkurentno programiranje.
Podržava *parametarski polimorfizam* (višeobličje) i *preopterećivanje* – što omogućava sažeto i generičko progamiranje. Podržava kompaktan i ekspresivan način definisanja *listi* kao osnovnih struktura funkcionalnog programiranja. Naglašava upotrebu rekurzije.
Funkcije višeg reda omogućavaju visok nivo apstrakcije i korišćenja funkcijskih oblikovnih obrazaca (uočavanje obrazaca izračunavanja koja se često sprovode i njihovo izdvajanje u funkcije višeg reda). Haskell ima podršku za monadičko programiranje koje omogućava da se propratni efekti izvedu bez narušavanja referentne transparentnosti.
Ima razrađenu biblioteku standardnih funkcija (Standard Library) i dodatnih modula (Hakage). Standardizaciju sprovodi međunarodni odbor (Haskell Commitee).

- Haskell dokumentacija
- Real world Haskell – knjiga
- Wiki – Haskel u industriji
- GHC – Glasgow Haskell Compiler – interaktivni interpreter i kompajler
- Razvojno okruženje za Haskell

GHC je kompajler koji proizvodi optimizovan kod koji se može upotrebljavati za stvarne primene. Ekstenzija za Haskell kod je `.hs`. Interpreter je `ghci`.

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

Biblioteka `Prelude` definiše osnovne funkcije.
`:help, :?, :h` – prikazivanje komandi interpretera
`:quit, :q` – izlazak iz interpretera

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

<statement>                 evaluate/run <statement>
:                           repeat last command
:{\n ..lines.. \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 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 ''Hello, World!''
Hello, World!

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

Prelude> sqrt 2
1.4142135623730951

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

Primena funkcije se piše bez zagrada, kao u lambda računu, ali zagrade mogu da budu potrebne za grupisanje argumenata. Primena funkcije je *levo asocijativna*, tako da bez zagrada, poslednji izraz u primeru bi se tumačio kao primena `sqrt` na funkciju `abs` (sto interpreter prijavi kao grešku jer se ne poklapaju tipovi). Ako funkcija uzima više argumenata, koristi se Curryjev postupak.

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

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

Liste su posebno bitne u deklarativnim jezicima.

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

Funkcije za rad sa listama:

```
Prelude> head [1,2,3,4,5]
1
Prelude> length [1..5]
5
Prelude> take 2 [1,2,3,4,5,6]
[1,2]
Prelude>sum [1,6..90]
783
Prelude>head ['a', 'b', 'c']
'a'
```

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

Lenjo izracunavanje – lista jeste beskonačna, ali nama treba 50-ti element i lista će samo dotle biti sračunata.

```
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
```

```
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> 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> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
Prelude> [ x^2 | x <- [1..5]]
[1,4,9,16,25]
Prelude> [(x,y) | x<-[1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
Prelude> [(x,y) | x<-[1..10], y <- [1..10], x+y==10]
[(1,9),(2,8),(3,7),(4,6),(5,5),(6,4),(7,3),(8,2),(9,1)]
```

Definisanje funkcije:

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

Osnovni tipovi:

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

Prethodni primeri su mogli da se isprobaju u interpreteru jer su bili kratki i jednostavni. Za pisanje programa u Haskell-u koristi se kompajler. Kod pišemo u datoteci sa ekstenzijom `.hs`. Ukoliko želimo da napravimo izvršnu verziju, potrebno je da definišemo main funkciju od koje će početi izvršavanje.
Prevođenje: `ghc 1.hs`
Pokretanje: `./1`
Možemo napisati i kod koji sadrži samo definicije funkcija koje učitavamo u interpreter kao modul i koristimo:
`Prelude> :load 1.hs`
Izlazak sa `*Main> :module`

Tipovi funkcija
Funkcija argumente jednog tipa preslikava u argumente drugog tipa. Na primer: ⟶
Tipovi se mogu navesti prilikom definicije funkcije.

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

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

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

Prethdno smo upotrebljavali kada imamo različite tipove nad istom strukturom podataka. *Preopterećivanje* koristimo kada su nad različitim strukturama podataka definisane iste operacije (na primer + za cele i realne brojeve). Preopterećivanje se ostvaruje preko *tipskih razreda*.
Tipski razred definiše koje funkcije neki tip mora da implementira da bi pripadao tom razredu.

- EQ – Tipovi sa jednakošću == / =
- ORD – Tipovi sa uređenjem (nasleđuje EQ) <, >, <=, ...
- NUM – Numerički tipovi (nasleđuje ORD) $+, -, *, ...$
- INTEGRAL – Celobrojni tipovi (nasleđuje NUM) div, mod
- FRACTIONAL – Razlomački tipovi (nasleđuje NUM) $/, recip$

Tipskim razredim se ograničavaju funkcije. Na primer, `sum :: Num a => [a] -> a` sumiranje može da se definiše samo nad numeričkim tipovima.

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

<interactive>:52:1:
No instance for (Num Char)
arising for a use of 'sum'
Possible fix: add an instance declaration for (Num Char)
In the expresion: 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''
```

Karijeve funkcije argumente uzimaju jedan po jedan što ih čini veoma fleksibilnim. Mogu se delimično evaluirati, tako da se definišu nove funkcije kojima su neki argumenti početne funkcije fiksirani. Delimičnom evaluacijom se fiksiraju levi argumenti funkcije.

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

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

6 Konkurentno programiranje

6.1 Uvod

Konkurentnu paradigmu karakteriše više procesa koji se izvršavaju u istom vremenskom periodu, a koji imaju zajednički cilj.
Konkurentnost nije nova ideja. Veliki deo teorijskih osnova datira još iz 1960-ih godina, a već Algol68 sadrži podršku za konkurentno programiranje. Međutim, široko je rasprostranjen interes za konkurenost poslednjih 20-ak godina. Uzroci tome, kao i motivacija za korišćenje konkurentnog programiranja, su:

Podrška logičkoj strukturi problema

Porast broja grafičkih, multimedijalnh i veb-zasnovanih aplikacija koje se sve prirodno predstavljaju konkurentnim nitima.

Dobijanje na brzini

Dostupnost jeftinih viseprocesorskih masina

Rad sa fizički nezavisnim uređajima

Potreba za umrežavanjem računara (frame računara) i različitih uređaja.

6.1.1 Podrška logičkoj strukturi problema

Konkurentni mehanizmi su originalno izmišljeni za rešavanje određenih problema u okviru operativnih sistema, ali se sada koriste u raznim aplikacijama. Mnogi programi moraju da vode računa istovremeno o više nego jednom zadatku koji je u velikoj meri nezavisan, pa je u takvim situacijama logično da se zadaci podele u različite kontrolne niti.

Primer aplikacije čiji se dizajn oslanja na konkurentnost je veb pregledač (eng. web browser). On mora da izvršava više različitih funkcionalnosti istovremeno (konkurentnost na nivou jedinica), npr primanje i slanje podataka serveru, prikaz teksta i slika na ekranu, reagovanje na korisničke akcije mišom i tastaturom... U ovom slučaju konkurentnost može da se odnosi i na *konkurentnost u užem smislu* koja obuhvata jedan procesor, a konkurentnost se ostvaruje implementiranjem izvršavanja različitih niti ili procesa.

6.1.2 Dobijanje na brzini

Ukoliko imamo više procesora, treba ih iskoristiti da bi se dobilo na brzini. Više procesora odgovara *paralelnom programiranju*. Paralelno programiranje može da se odnosi i na više procesora na različitim mašinama, ali se najčešće misli na višeprocesorsku mašinu. Procesi međusobno komuniciraju preko zajedničke memorije (ukoliko je višeprocesorska mašina u pitanju) ili slanjem poruka.
Dobijanje na brzini može da se ostvari i kod jednoprocesorske mašine konkurentnim izvršavanjem. Program, po prirodi, ne mora da bude u potpunosti sekvencijalan i to se moze iskoristiti.

6.1.3 Zajednički rad fizički nezavisnih uređaja

Aplikacije koje rade distribuirano korišćenjem različitih mašina, bilo da su u pitanju lokalno povezane mašine ili internet – *distribuirano programiranje*. Procesi međusobno salju poruke da bi razmenili informacije. Može se shvatiti kao vrsta paralelnog izračunavanja, ali sa drugačijom međusobnom komunikacijom koja nameće nove izazove. Postoje jezici dizajnirani za distribuirano programiranje, ali oni nisu u širokoj upotrebi.

6.1.4 Veza sa programskim jezicima

Na prvi pogled, konkurentnost može da izgleda kao jednostavan koncept. Pisanje konkurentnih programa je značajno teže od pisanja sekvencijalnih programa. Za osnovne koncepte konkurentnog programiranja potrebno je obezbediti odgovarajuću podršku u programskom jeziku.
Postoje dve vrste konkurentnosti, *fizička* konkurentnost (podrazumeva postojanje vše procesora) i *logička* konkurentnost. Sa stanovišta programera i dizajna programskog jezika, logička konkurentnost je ista kao i fizička.

Zadatak implementacije jezika je da korišćenjem operativnog sistema preslika logičku konkurentnost u odgovarajući hardver. Osnovni problem se odnosi na pitanja komunikacije i sinhronizacije procesa, kao i pitanje pristupa zajedničkim podacima. Veliki izazov za programere, dizajnere programskih jezika i dizajnere operativnih sistema (dobar deo podrške za konkurentnost obezbeđuje operativni sistem).

6.1.5 Hijerarhijska podela konkurentnosti

- *Konkurentna paradigma* je najširi pojam, tj konkurentnu paradigmu karakteriše više procesa koji se izvršavaju u istom vremenskom periodu.
- *Paralelna paradigma* je specijalizacija koja obuhvata postojanje više procesora.
- *Distribuirana paradigma* je specijalizacija paralelne paradigme u kojoj su procesori i fizički razdvojeni

Međutim, sa stanovišta semantike ove podele nemaju značajnu ulogu, one su bitne po pitanju implementacije i performansi.
Postoje 4 osnovna nivoa konkurentnosti:

1. Nivo instrukcije – izvršavanje dve ili više instrukcija istovremeno.
2. Nivo naredbe – izvršavanje dve ili više naredbi jezika višeg nivoa istovremeno.
3. Nivo jedinica – izvršavanje dve ili više jedinica (potprograma) istovremeno.
4. Nivo programa – izvršavanje dva ili više programa istovremeno.

Prvi i četvrti nivo konkurentnosti ne utiču na dizajn programskog jezika.

6.1.6 Skalabilnost i portabilnost

Konkurentnim programiranjem treba da se proizvedu skalabilni i portabilni algoritmi. *Skalabilnost* se odnosi na ubrzavanje izvršavanja porastom broja procesora. Ovo je važno jer se broj dostupnih procesora stalno povećava. Naravno, u razmatranju skalabilnosti mora se uzeti u obzir i priroda problema i njegova prirodna ograničenja (porast broja procesora nakon neke granice ne mora da ima pozitivan efekat). Idealna bi bila *linearna skalabilnost*, ali ona je retka.
Amdahlov zakon – mali delovi programa koji se ne mogu paralelizovati ce ograničiti mogućnost ukupnog ubrzavanja paralelizacijom (npr ako je $\alpha = 10\%$ koda koji se ne može paralelizovati, onda je maksimalno ubrzanje paralelizacijom $1/\alpha$, tj u ovom slucaju 10 puta, bez obzira na broj procesora koji se dodaju).
Komunikacija predstavlja dodatno usporavanje.
Portabilnost se odnosi na nezavisnost od konkretne arhitekture – životni vek hardvera je kratak, stalno izlaze nova hardverska rešenja i dobar algoritam mora da ne zavisi od hardvera.

6.2 Osnovni koncepti

Za konkurentno programiranje potrebna je podrška u okviru programskog jezika, ili u okviru biblioteka. Da bi se razmatrala podrška koju je potrebno da programski jezik pruži, najpre je potrebno razumeti osnovne koncepte konkurentnosti. Nažalost, terminologija u okviru različitih autora, programskih jezika i operativnih sistema nije konzistentna.

6.2.1 Zadatak, nit, proces

Zadatak (eng. *task*), ili posao, je jedinica programa, slična potprogramu, koja može da se izvrši konkurentno sa drugim jedinicama istog programa. Zadatak se razlikuje od potprograma na 3 načina:

- Zadaci mogu da počnu implicitno, ne moraju eksplicitno da budu pozvani.
- Kada program pokrene neki zadatak, ne mora uvek da čeka na njegovo izvršavanje pre nego što nastavi sa svojim.
- Kada se izvršavanje zadatka završi, kontrola ne mora da se vrati na mesto odakle je počelo izvršavanje.

Svaki zadatak u programu moze da bude podržan od strane jedne kontrolne niti ili procesa.
Zadaci se dele na dve opšte kategorije: **heavyweight** i **lightweight**. Teški imaju svoj sopstveni adresni prostor dok laki dele isti adresni prostor.

Teškim zadacima upravlja operativni sistem obezbeđujući deljenje procesorskog vremena, pristup datotekama, adresni prostor. Prelaz sa jednog teškog zadatka na drugi vrši se posredstvom operativnog sistema uz pamćenje stanja prekinutog procesa (to je skupa operacija). Stanje teškog zadatka obuhvata:

- Podatke o izvršavanju (stanje izvršavanja koje može biti spreman, radi, čeka,...), vrednosti registara, brojač instrukcija.
- Informacije o upravljanju resursima (informacije o memoriji, datoteke, ulazno-izlazni zahtevi i ostali resursi)

Promena konteksta je promena stanja procesora koja je neophodna kada se sa izvršavanjem jednog teškog zadatka prelazi na izvršavanje drugog teškog zadatka: potrebno je zapamtiti u memoriji stanje zadatka koji se prekida i na osnovu informacija u memoriji rekonstruisati stanje zadatka koji treba da nastavi izvršavanje. Promena konteksta je skupa, a to nije zanemarljivo. Dešava se veoma često, od nekoliko puta do par stotina ili hiljada puta u jednoj sekundi. Podaci o stanju procesa zauzimaju memoriju koja nije zanemarljiva, a bitna je i u kontekstu promašivanja u kešu koje značajno

utiče na performanse sistema.

Koncept lakih zadataka se uvodi kako bi se omogućio efikasniji prelaz sa jednog na drugi zadatak. Za razliku od teških zadataka, laki ne zahtevaju posebne računarske resurse koje on obuhvata. Podaci o lakom zadatku obuhvataju samo njegovo stanje izvršavanja, brojač instrukcija i vrednost radnih registara. Prelaz sa jednog lakog zadatka na drugi je stvar izmene sadržaja radnih registara i pamćenja brojača instrukcija i stanja izvršavanja i vrši se brzo.
Upravljanje nitima ostvaruje se preko korisničkih biblioteka. Tri najpoznatije biblioteke su:

- POSIX Pthreads
- Java threads
- Win32 threads

Kada je reč o operativnim sistemima, niti su podržane u:

- Windows XP/2000, Vista, 7, ...
- Linux
- Solaris, Tru64 UNIX, Mac OS X

Terminologija je nekonzistentna. Na primer, u C-u teški zadaci su procesi (koji imaju svoj adresni prostor), laki zadaci su niti (koje imaju zajednički adresni prostor). Kreiranje lakih zadataka je efikasnije od kreiranja teških. Sistemski poziv `fork` je za kreiranje novih procesa. Biblioteka `pthread` je za kreiranje niti.

6.2.2 Izbor nivoa konkurentnosti

Osnovna odluka koju programer mora da donese kada piše paralelni program je kako da podeli posao. Šta paralelizovati?

Za svaki broj iz niza brojeva odrediti koja od narednih svojstava ispunjava:

- broj je prost,
- broj je savršen,
- broj je deljiv sumom svojih cifara,
- broj ima paran broj delilaca,
- broj je jednak zbiru kubova svojih cifara.

Najčešća strategija, koja dobro radi na malim mašinama, je da se koriste različite niti za svaki od glavnih programerskih zadataka ili funkcija. Na primer, kod procesora reči, jedan zadatak bi mogao da bude zadužen za prelamanje paragrafa u linije, drugi za određivanje strana i raspored slika, treći za pravopisnu proveru i proveru gramatičkih gresaka, četvrti za renderovanje slika na ekranu,... Ova strategija se obično naziva **paralelizam zadataka**. Njen nedostatak je da ne skalira prirodno dobro ukoliko imamo veliki broj pocesora.
Za dobro skaliranje na velikom broju procesora, potreban je **paralelizam podataka**. Kod paralelizma podataka, iste operacije se pimenjuju konkurentno na elemente nekog velikog skupa podataka. Na primer, program za manipulaciju slikama, može da podeli ekran na n manjih delova i da koristi različite niti da bi procesirao svaki taj pojedinačni deo. Ili, igrica može da koristi posebnu nit za svaki objekat koji se pokreće.

Program koji je dizajniran da koristi paralelizaciju podataka najčešće se zasniva na paralelizaciji petlji: Svaka nit izvršava isti kod, ali korišćenjem različitih podataka. Za ovu vrstu paralelizma se najčešće koristi paralelizam na nivou naredbi. Na primer, u C#, ukoliko se koristi *Parallel FX Library*:

```
Parallel.For(0, 100, i => { A[i] = foo( A[i] ) ; } );
```

Neophodno je da programer zna da su pozivi funkcije `foo` međusobno nezavisni, ukoliko nisu, neophodna je **sinhronizacija**. Idealno bi bilo kada bi kompajler mogao sam da zaključi da je nešto nezavisno i da sam obavi paralelizaciju umesto nas, ali, nažalost u opštem slučaju to nije moguće.
Paralelizacija podataka je prirodna za neke vrste problema, ali nije za sve.

- Proizvod komponenti vektora
- Skalarni proizvod dva vektora
- Množenje matrica
- Pronalaženje prostih brojeva u nizu
- Pronalaženje brojeva koje imaju određenu osobinu (prosti, savršeni, blizanci, Armstrongovi,...) i pripadaju intervalu $[n, m]$.

6.2.3 Komunikacija

Zadaci međusobno moraju da komuniciraju. Komunikacija se odnosi na svaki mehanizam koji omogućava jednom zadatku da dobije informacije od drugog. Može se ostvariti preko zajedničke memorije(ukoliko zadaci dele memoriju) ili slanjem poruka.

Ukoliko imamo zajedničku memoriju, izabranim promenljivama se može pristupiti iz različitih zadataka (ovo se, pre svega, odnosi na komunikaciju između lakih zadataka). Da bi dva zadatka komunicirala, jedan upiše vrednost promenljive, a drugi je jednostavno pročita. U ovom slučaju, važan je redosled čitanja i pisanja promenljivih i potrebno je starati se o ispravnom korišćenju resursa i eventualnim sukobima.

Slanje poruka se može upotrebljavati uvek, i kada zadaci imaju i kada nemaju zajedničku memoriju. U tom slučaju, da bi se ostvarila komunikacija, jedan zadatak mora eksplicitno da pošalje podatak drugom. Slanje poruka može da se ostvari na razne načine. Mogu se koristiti tokovi, signali, cevi, soketi, kanali (mehanizmi međuprocesne komunikacije)... Ukoliko je slanje poruka u okviru iste mašine, onda se ono smatra pouzdanim. Slanje poruka u distribuiranim sistemima nije pouzdano jer podaci putuju kroz mrežu gde mogu da se zagube i tu su potrebni dodatni mehanizmi i protokoli komunikacije. Za slanje poruka u distribuiranim sistemima koriste se različiti protokoli.

6.2.4 Sinhronizacija

Sinhronizacija se odnosi na mehanizam koji dozvoljava programeru da kontroliše redosled u kojem se operacije dešavaju u okviru različitih zadataka. Sinhronizacija je obično implicitna u okviru modela slanja poruka: poruka prvo mora da se pošalje da bi mogla da se primi, tj ako zadatak pokuša da primi poruku koja još nije poslata, mora da sačeka pošiljaoca da je najpre pošalje.

Sinhronizacija nije implicitna u okviru modela deljene memorije. Postoje dve vrste sinhronizacije kada zadaci dele podatke: *saradnja* i *takmičenje*. Sinhronizacija saradnje je neophodna između zadatka A i zadatka B kada A mora da čeka da B završi neku aktivnost pre zadatka A da bi zadatak A mogao da pošne ili da nastavi svoje izvršavanje. Primer: proivođač-potrosač.

Sinhronizacija takmičenja je neophodna između dva zadatka kada oba zahtevaju nekakav resurs koji ne mogu istovremeno da koriste. Na primer ako zadatak A treba da pristupa deljenom podatku x dok B pristupa x, tada zadatak A mora da čeka da zadatak B završi procesiranje podatka x.

Ukoliko zadatak A treba da vrednost deljene promenljive uveća za 1, dok zadatak B treba da vrednost deljene promenljive uveća 2 puta, onda ukoliko nema sinhronizacije, kao rezultat rada ovih vrednosti mogu da se dese različite situacije.

```
x = 3;
A : x = x + 1;
B : x = 2 * x;
```

Na mašinskom nivou, imamo sledeća tri koraka: uzimanje vrednosti, izmena, upisivanje nove vrednosti. Bez sinhronizacije, ukoliko je početna vrednost 3, mogući ishodi su:

- 8 – ako se A prvo izvrši, pa zatim B
- 7 – ukoliko se B prvo izvrši, pa zatim A
- 6 – ukoliko A i B uzmu istovremeno vrednost, ali je A prvi upiše nazad, pa je zatim B prepiše
- 4 – ukoliko A i B uzmu istovremeno vrednost, ali je B prvi upiše nazad, pa je zatim A prepiše

Situacija koja vodi do ovih problema naziva se **uslov takmičenja** (eng.*race condition*) jer se dva ili više zadataka takmiče da koriste deljene resurse i ponašanje programa zavisi od toga ko pobedi na takmičenju, tj ko stigne prvi. Osnovna uloga sinhronizacije je da za sekvencu instrukcija koju nazivamo *kritična sekcija* obezbedi da se izvrši atomično, tj da se sve instrukcije kritične sekcije izvrše bez prekidanja.

Za kontrolisanje pristupu deljenim resursima, koristi se **zaključavanje** ili uzajamno isključivanje. Za uzajamno isključivanje mogu se koristiti npr *semafori* ili *monitori*.

Semafori su prvi oblik sinhronizacije, implementiran vec u Algol68, i dalje prisutni, npr u Javi. Imaju dve moguće operacije, P i V (ili wait i release). Nit koja poziva P atomično umanjuje brojač i čeka da n postane nenegativan. Nit koja poziva V atomično uvećava brojač i budi čekajuću nit, ukoliko postoji. Iako se široko koriste, semafori se takođe smatraju kontrolom niskog nivoa, nepogodnom za dobro struktuiran i lako održiv kod. Korišćenje semafora lako dovodi do *grešaka* i do *uzajamnog blokiranja*.

Drugi koncept su monitori koji enkapsuliraju deljene strukture podataka sa njihovim operacijama, tj čine deljene podatke apstraktnim tipovima podataka sa specifičnim ograničenjima. Monitori su prisutni npr u Javi (modifikator *synchronized*, C#(klasa *Monitor*),... I monitori se takođe smatraju kontrolom niskog nivoa.

Mutex – *mutual exclusion* (postoje u C++, ne postoje u Javi). Atomično zaključavanje, samo jedna nit može zaključati podatak, ako je muteks već zaključan, nit koja želi da ga zaključa mora da sačeka na njegovo otključavanje.

Semantika katanca – osnovni način bezbednog deljenja podataka u konkurentnom okruženju. Ima različit pristup pod različitim uslovima (npr više njih može da čita, samo jedan može da piše, tako da niko drugi ne sme da čita).

I kod deljene memorije i kod slanja poruka, sinhronizacija može da se implementira na dva načina: **zauzetim čekanjem** ili **blokiranjem**.
Busy-waiting synchronization – zadatak u petlji stalno proverava da li je neki uslov ispunjen (da li je poruka stigla ili da li deljena promenljiva ima neku određenu vrednost). Nema smisla na jednom procesoru.
Blokirajuća sinhronizacija – zadatak svojevoljno oslobađa procesor, a pre toga ostavlja poruku u nekoj strukturi podataka zaduženoj za sinhronizaciju. Zadatak koji ispunjava uslov u nekom trenutku naknadno, pronalazi poruku i sprovodi akciju da se prvi zadatak odblokira, tj da nastavi sa radom.

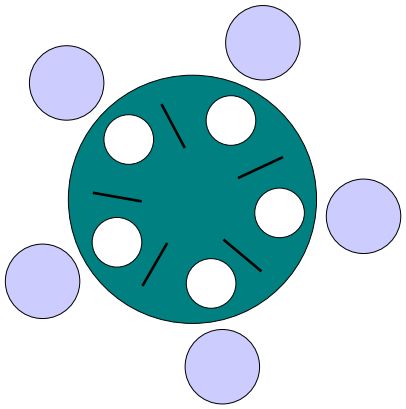
6.2.5 Koncept napredovanja (eng. *liveness*)

Kod sekvencijalnog izvrsavanja programa, program ima karakteristiku napredovanja ukoliko nastavlja sa izvršavanjem, dovodeći do završetka rada programa u nekom trenutku (ukoliko program ima osobinu da se završava, npr nema beskonačnih petlji). Opštije, koncept napredovanja govori da ukoliko neki događaj treba da se desi (npr završetak rada programa) da će se on i desiti u nekom trenutku, odnosno da se stalno pravi nekakav progres tj napredak.
U konkurentnoj sredini sa deljenim objektima, napredak zadatka može da prestane, odnosno može da se desi da program ne može da nastavi sa radom i da zbog toga nikada ne završi svoj rad.

> Na primer, pretpostavimo da oba zadatka, A i B, zahtevaju resurse X i Y da bi mogli da završe svoj posao. Ukoliko se desi da zadatak A dobije resurs X, a da zadatak B dobije Y, tada da bi nastavili sa radom, zadatku A treba resurs Y, a zadatku B treba X, i oba zadatka čekaju onaj drugi da bi nastavili sa radom. Na taj način, oba gube napredak i program ne moze da završi sa radom normalno.

Prethodno opisan način gubitka napredka naziva se **uzajamno blokiranje**, smrtonosno blokiranje (eng.**deadlock**). Deadlock je ozbiljna pretnja pouzdanosti programa i zahteva ozbiljna razmatranja i u jeziku i u dizajnu programa.
Livelock (živo blokiranje) – kada svi zadaci nešto rade, ali nema progresa.
Lockout (*individual starvation*) – mogućnost da jedan zadatak sprečava izvršavanje drugog.

Slika 2: Problem filozofa koji večeraju, Hoare, 1985.



Treba obezbediti:

- uzajamno isključivanje kritičnih sekcija
- pristupačnost resursima
- poštenost u izvršavanju

Communicating Sequential Processes, Hoare – http://www.usingcsp.com/cspbook.pdf

6.2.6 Odnos konkurentnosti, potprograma i klasa

- Jedinstveno pozivanje – ne sme se istovremeno upotrebljavati u različitim nitima (npr koristi globalne promenljive ili globalne resurse na način koji nije bezbedan)
- Ulazne (eng. *reentrant*) – može se upotrebljavati na različitim nitima ali uz dodatne pretpostavke (da se ne koriste nad istim podacima)
- Bezbedne po niti (eng. *thread-safe*) – sme se bezbedno upotrebljavati bez ograničenja.
- Slično važi i za klase.
- Zašto je teško naći gresku u konkurentnim programima?
- Preporuke bezbednog konkurentnog programiranja.

7 Logičko programiranje

Imperativnu paradigmu karakteriše postojanje naredbi, sekvenci, selekcija, iteracija i propratnih efekata. Funkciopnalnu paradigmu karakteriše evaluacija funkcija korišćenjem redukcija. U logičkoj paradigmi imamo logičke metode (na primer metod rezolucije).
Pomenute paradigme se temelje na različitim teorijskim modelima:

Formalizam za imperativne jezike – Tjuringova mašina

Imperativni jezici imaju svu izražajnost Tjuringove mašine (uz ograničenje memorije i resursa).

Formalizam za funkcionalne jezike – Lambda račun

Funkcionalni jezici imaju svu izražajnost lambda računa (uz ograničenje memorije i resursa).

Formalizam za logičke jezike – Logika prvog reda

Logika prvog reda *nije* formalizam izračunljivosti, tako da je teško vršiti poređenje ove vrste. Ipak, većina logičkih jezika je Tjuring-kompletno.

7.1 Rešavanje problema

U logičkom programiranju, logika se koristi kao deklarativni jezik za opisivanje problema, a dokazivač teorema kao mehanizam za rešavanje problema. Rešavanje problema je podeljeno između programera koji opisuje problem i dokazivača teorema koji problem rešava.
Sintaksa opisa problema je dosta slična u jezicima logičke paradigme, ali dokazivač teorema može da se razlikuje. Najznačajniji predstavnik logičke paradigme je Prolog, koji se često koristi i kao sinonim za logičku paradigmu.
U procesu rešavanja problema, Prolog koristi metod rezolucije.

7.2 Razvoj logičkog programiranja

1879.	G.Frege – Predikatski račun (sredstvo za analizu formalne strukture čistog mišeljenja)
1915-1936.	Gedel, Erban, Skulem, Turing,... Osnovi teorije izračunljivosti
1965.	J.A.Robinson, Metod rezolucije
1967.	Prvi logički jezik - ABSYS - Aberdeen SYStem M.Foster, T.Elkok, Group for Computing Research, University of Aberdeen (UK) Radi sa tvrđenjima (aksiomama) preko kojih se, nakon postavljanja pitanja, deduktivnim putem generiše odgovor. ABSYS je antipicira razne koncepte koji se kasnije koriste u logičkom programiranju.
Razvoj Prologa	
1971.	Pod rukovodstvom A.Colmerauer-a u Marselju kreiran Q-System (obrada prirodnih jezika).
1972.	Q-System preimenovan u PROLOG (PROgramming in LOGic). Saradnja sa Robertom Kovalskim iz Edinburga (automatsko dokazivanje). Implementiran prvi interpretator za Prolog. (Saradnici: Ph. Roussel, R.Pasero, J.Trudel)
1974.	Na kongresu IFIP-a (International Federation for Information Processing) R.Kavalski predstavio Prolog široj javnosti.
1977.	David Warren napravio efikasnu implementaciju kompajlera za Prolog za DEC-10 u Edinburgu. (Edinburški Prolog). Osnova sintakse za moderan Prolog.
1981.	Seminari u Sirakuzi i Los Anđelesu.
1982.	Prva međunarodna konferencija o Prologu u Marselju.
1983.	Japanski projekat o razvoju računara 5. generacije.
1986.	The Association for Logic Programming http://www.logicprogramming.org/
1993.	Završen Japanski projekat razvoja računara 5. generacije.
1995.	ISO Prolog standard
2007,2012.	Korekcije standarda, dodavanje modula.

Prolog se često koristi kao sinonim za logičko programiranje. Nakon početnih godina intenzivnog razvoja jezika, sada je razvoj usmeren ka integracijama jezika sa drugim programskim jezicima. I dalje se istražuju efikasniji algoritmi za prevođenje Prolog programa u izvršni kod. Razne implementacije Prologa: BProlog, GNU Prolog, JIProlog, Visual Prolog, SWI Prolog,... Prolog je uticao na razvoj raznih programskih jezika: ALF, Fril, Godel, Mercury, Oz, Ciao, Visual Prolog, XSB, λProlog,...

7.3 Drugi predstavnici logičke paradigme

Pored Prologa, značajni predstavnici logičke paradigme su i **ASP** (Answer Set Programming) i **Datalog**. Sintaksa Dataloga i jezika koji pripadaju ASP podparadigmi su veoma slične Prologu, ali se proces rešavanja problema razlikuje. ASP za izračunavanje ne koristi metod rezolucije već rešavače za iskaznu logiku. Koristi se najčešće za pretragu kod NP teških problema, npr bojenje grafova, hamiltonovi ciklusi, velike klike,...
Datalog (logic and databases) je u sintaksnom smislu podskup Prologa koji se koristi za integraciju podataka, izvlačenje podataka, umrežavanje, analizu programa, cloud computing. Datalog može da koristi različite efikasne algoritme za određivanje vrednosti upita. Datalog nije Tjuring kompletan.

7.4 Primena

Logičko programiranje je *pogodno* za:

- rešavanje problema matematičke logike
- obradu prirodnih jezika
- podršku relacionim bazama podataka
- automatizaciju projektovanja
- simboličko rešavanje jednačina
- razne oblasti veštačke inteligencije

Nije pogodno za: I/O algoritme, grafiku, numeričke algoritme.

7.5 Logika prvog reda - osnovni pojmovi

7.5.1 Sintaksa

Logički deo jezika prvog reda čine:

- skup promenljivih V
- skup logičkih veznika $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$
- skup kvantifikatora $\{\exists, \forall\}$
- skup logičkih konstanti $\{\top, \bot\}$
- skup pomoćnih simbola $\{(,)\}$

Elementi ovih skupova nazivaju se logički simboli.

Rečnik ili signatura

Rečnik ili signatura sastoje se od najviše prebrojivih skupova $\sum$ i $\prod$ koje redom nazivamo skupom **funkcijskih simbola** i skupom **predikatskih simbola**, kao i od funckije ar koja preslikava uniju ovih skupova u nenegativne cele brojeve i koju nazivamo **arnost**.
Presek svaka dva od prethodno nabrojanih skupova je prazan. Funkcijske simbole arnosti 0 zovemo simbolima konstanti. Skupovi $\sum$ i $\prod$ čine nelogički deo jezika prvog reda, a sve njihove elemente zovemo **nelogičkim simbolima**.

Term

Skup **termova** nad signaturom $\mathscr{L} = (\sum, \prod, ar)$ i skupom promenljivih V je skup za koji važi:

- svaki simbol konstante (tj. svaki funkcijski simbol arnosti 0) je term
- svaki simbol promenljive je term
- ako je f funkcijski simbol za koji je $ar(f) = n$ i $t_1, t_2, \ldots, t_n$ su termovi, onda je i $f(t_1, t_2, \ldots, t_n)$ term

Termovi se mogu dobiti samo konačnom primenom prethodnih pravila.

Atomičke formule

Skup **atomičkih formula** nad signaturom $\mathscr{L} = (\sum, \prod, ar)$ i skupom promenljivih V je skup za koji važi:

- logičke konstante $\top$ i $\bot$ su atomičke formule
- ako je p predikatski simbol za koji je $ar(p) = n$ i $t_1, t_2, \ldots, t_n$ su termovi, onda je $p(t_1, t_2, \ldots, t_n$ atomička formula.

Dobro zasnovane formule (ili samo formule)

Skup **dobro zasnovanih formula** nad signaturom $\mathscr{L} = (\sum, \prod, ar)$ i skupom promenljivih V je skup za koji važi:

- svaka atomička formula je dobro zasnovana formula
- ako je A dobro zasnovana formula, onda je i $(\neg A)$ dobro zasnovana formula

- ako su A i B dobro zasnovane formule, onda su i $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$ i $(A \Leftrightarrow B)$ dobro zasnovane fomule
- ako je A dobro zasnovana formula i x je promenljiva, onda su i $((\forall x)A)$ i $((\exists x)A)$ dobro zasnovane formule.

Dobro zasnovane formule se mogu dobiti samo konačnom primenom prethodnih pravila.

Osnovni pojmovi

Literal je atomička formula ili negacija atomičke formule.
Klauza je disjunkcija literala.
Pod terminom **izraz** podrazumevaju se termovi i (dobro zasnovane) formule.

7.5.2 Supstitucija i unifikacija

Supstitucija za term

Term dobijen zamenom (supstitucijom) promenljive x termom t_x u termu t označavamo sa $t[x \mapsto t_x]$ i definišemo na sledeći način:

- ako je t simbol konstante, onda je $t[x \mapsto t_x] = t$
- ako je $t = x$, onda je $t[x \mapsto t_x] = t_x$
- ako je $t = y$, gde je $y \neq x$, onda je $t[x \mapsto t_x] = t$
- ako je $t = f(t_1, t_2, \ldots, t_n)$, onda je $t[x \mapsto t_x] = f(t_1[x \mapsto t_x], t_2[x \mapsto t_x], \ldots, t_n[x \mapsto t_x])$

Supstitucija za formule

Formulu dobijenu zamenom (supstitucijom) promenljive x termom t_x u formuli $\mathscr{A}$ označavamo sa $\mathscr{A}[x \mapsto t_x]$ i definišemo na sledeći način:

- $\top[x \mapsto t_x] = \top$
- $\bot[x \mapsto t_x] = \bot$
- ako je $\mathscr{A} = p(t_1, t_2, \ldots, t_n)$, onda je $\mathscr{A}[x \mapsto t_x] = p(t_1[x \mapsto t_x], t_2[x \mapsto t_x], \ldots, t_n[x \mapsto t_x])$
- $(\neg\mathscr{A})[x \mapsto t_x] = \neg(\mathscr{A}[x \mapsto t_x])$
- $(\mathscr{A} \vee \mathscr{B})[x \mapsto t_x] = (\mathscr{A}[x \mapsto t_x] \vee \mathscr{B}[x \mapsto t_x])$
- $(\mathscr{A} \wedge \mathscr{B})[x \mapsto t_x] = (\mathscr{A}[x \mapsto t_x] \wedge \mathscr{B}[x \mapsto t_x])$
- $(\mathscr{A} \Rightarrow \mathscr{B})[x \mapsto t_x] = (\mathscr{A}[x \mapsto t_x] \Rightarrow \mathscr{B}[x \mapsto t_x])$
- $(\mathscr{A} \Leftrightarrow \mathscr{B})[x \mapsto t_x] = (\mathscr{A}[x \mapsto t_x] \Leftrightarrow \mathscr{B}[x \mapsto t_x])$
- $(\forall x\mathscr{A})[x \mapsto t_x] = (\forall x\mathscr{A})$
- $(\exists x\mathscr{A})[x \mapsto t_x] = (\exists x\mathscr{A})$
- ako je $x \neq y$, neka je z promenljiva koja se ne pojavljuje ni u $(\forall y)\mathscr{A}$ ni u t_x;
  tada je $(\forall y\mathscr{A})[x \mapsto t_x] = (\forall z)\mathscr{A}[y \mapsto z][x \mapsto t_x]$
- ako je $x \neq y$, neka je z promenljiva koja se ne pojavljuje ni u $(\exists y)\mathscr{A}$ ni u t_x;
  tada je $(\exists y\mathscr{A})[x \mapsto t_x] = (\exists z)\mathscr{A}[y \mapsto z][x \mapsto t_x]$

Primetimo da poslednja dva pravila u prethodnoj definiciji obezbeđuju, na primer, da $((\forall y)p(x, y))[x \mapsto y]$ ne bude $(\forall y)p(y, y)$ već $(\forall z)p(y, z)$.

Uopštena zamena (supstitucija)

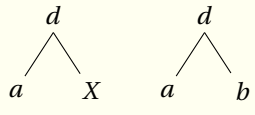
Uopštena zamena (supstitucija) σ je skup zamena $[x_1 \mapsto t_1], [x_2 \mapsto t_2], \ldots, [x_n \mapsto t_n]$ gde su x_i promenljive i t_i su proizvoljni termovi i gde je $x_i \neq x_j$ za $i \neq j$. Takvu zamenu zapisujemo kraće $[x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n]$
Uopštena zamena primenjuje se simultano na sva pojavljivanja promenljivih $x_1, x_2, \ldots, x_n$ u polaznom izrazu i samo na njih (tj. ne primenjuje se na podtermove dobijene zamenama). Izraz koji je rezultat primene zamene σ nad izrazom E, označavamo sa $E\sigma$.

- Za $\sigma = [x \mapsto f(y)]$ i $s = g(a, x)$ važi $s\sigma = g(a, f(y))$
- Za $\sigma = [x \mapsto f(x)]$ i $s = g(a, x)$ važi $s\sigma = g(a, f(x))$
- Za $\sigma = [x \mapsto f(y), y \mapsto a]$, $s = g(a, x)$ i $t = g(y, g(x, y))$ važi $s\sigma = g(a, f(y))$ i $t\sigma = g(a, g(f(y), a))$

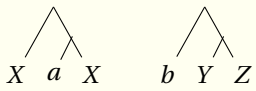
Unifikacija

Problem unifikacije je problem ispitivanja da li postoji supstitucija koja čini dva izraza (dva terma ili dve formule) jednakim.
Ako su e_1 i e_2 izrazi i ako postoji supstituija σ takva da važi $e_1\sigma = e_2\sigma$ onda kažemo da su izrazi e_1 i e_2 **unifikabilni** i da je supstitucija σ **unifikator** za ta dva izraza. Dva unifikabilna izraza mogu da imaju više unifikatora.

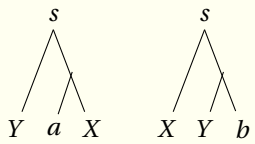
Primeri unifikacije:



$\{X = b\}$



$\{X = b, Y = a, Z = b\}$



Nemoguća unifikacija

Neka je term t_1 jednak $g(x,z)$, neka je term t_2 jednak $g(y,f(y))$ i neka je σ supstitucija $[y \mapsto x, z \mapsto f(x)]$. Tada je i $t_1\sigma$ i $t_2\sigma$ jednako $g(x,f(x))$, pa su termovi t_1 i t_2 unifikabilni, a σ je (jedan) njihov unifikator.

Unifikator termova t_1 i t_2 je npr. i $[x \mapsto a, y \mapsto a, z \mapsto f(a)]$
Termovi $g(x,x)$ i $g(y,f(y))$ nisu unifikabilni.

Među svim unifikatorima za dva izraza postoji jedan koji je najopštiji, tj. svi drugi se mogu dobiti iz njega primenom dodatne supstitucije.

Ako imamo termove $f(x)$ i $f(y)$ i supstitucije $\sigma_1 = [x \mapsto a, y \mapsto a]$, $\sigma_2 = [x \mapsto b, y \mapsto b]$ i $\sigma = [x \mapsto y]$ u ovom slučaju σ je najopštiji unifikator.

Postoji algoritam koji pronalazi **najopštiji unifikator** i vraća neuspeh ukoliko identifikator za dva izraza ne postoji.

7.5.3 Metod rezolucije

Metod rezolucije je metod za izvođenje zaključaka (logičkih posledica) u logici prvog reda koji se zasniva na pravilu rezolucije. Na primer, posledica formula $A \Rightarrow B$ i $B \Rightarrow C$ je formula $A \Rightarrow C$. Navedeni zaključak ne bi mogao da se izvede iz formula $A \Rightarrow B'$ i $B'' \Rightarrow C$. Međutim, ukoliko su B' i B'' unifikabilni, onda se oni mogu učiniti jednakim za neko σ pa je posledica formula $A\sigma \Rightarrow B'\sigma$ i $B''\sigma \Rightarrow C\sigma$ upravo $A\sigma \Rightarrow C\sigma$. Zapravo, pravilo rezolucije se iskazuje u terminima klauza, tj. $\neg A \vee B$ i $\neg B \vee C$ daje $\neg A \vee C$.

8 Prolog

8.1 Teorijske osnove

Kao što se Haskell oslanja na lambda račun, tako se Prolog oslanja na logiku prvog reda i metod rezolucije. Potrebno je poznavanje pojmova iskazne logike i logike prvog reda. Prološki zapis programa je izveden iz zapisa formula logike prvog reda. Ovaj zapis je redukovani zapis formula logike prvog reda – ne mogu se sve formule logike prvog reda izraziti u Prologu. U Prologu se mogu izraziti samo **Hornove klauze**. Hornova klauza je *disjunkcija literala sa najviše jednim nenegiranim literalom.*

Hornova klauza odgovara implikaciji

$$(A_1 \wedge A_2 \wedge \ldots \wedge A_n) \Rightarrow B$$
$$\Longleftrightarrow$$
$$\neg(A_1 \wedge A_2 \wedge \ldots \wedge A_n) \vee B$$
$$\Longleftrightarrow$$
$$\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n \vee B$$

Hornove klauze u Prologu se zapisuju u obliku:

$$B \Leftarrow (A_1 \wedge A_2 \wedge \ldots \wedge A_n)$$

pri čemu se znak $\Leftarrow$ zapisuje kao `:-`

```
B:-(A1, A2,..., An)
```

Hornove klauze omogućavaju efikasnu primenu metoda rezolucije. One predstavljaju ograničenje: ne može se izraziti tvrđenje oblika $\neg A \Rightarrow B$. Prethodnom tvrđenju odgovara formula $A \vee B$, dakle formula koja sadrži dva nenegativna literala.

Supstitucija je preslikavanje koje promenljive preslikava u termove. Supstitucija se sastoji od konačnog broja pravila preslikavanja, npr. $x \to a, y \to f(a,b), u \to v$.
Dva terma t i u se mogu unifikovati ako i samo ako postoji supstitucija σ tako da vazi $t\sigma = u\sigma$. Unifikacija se koristi prilikom traženja rešenja u okviru Prologa.

Programiranje u Prologu se sastoji u

- obezbeđivanju činjenica o objektima i odnosima među njima
- definisanju pravila o objektima i odnosima među njima
- formiranju upita o objektima i odnosima među njima

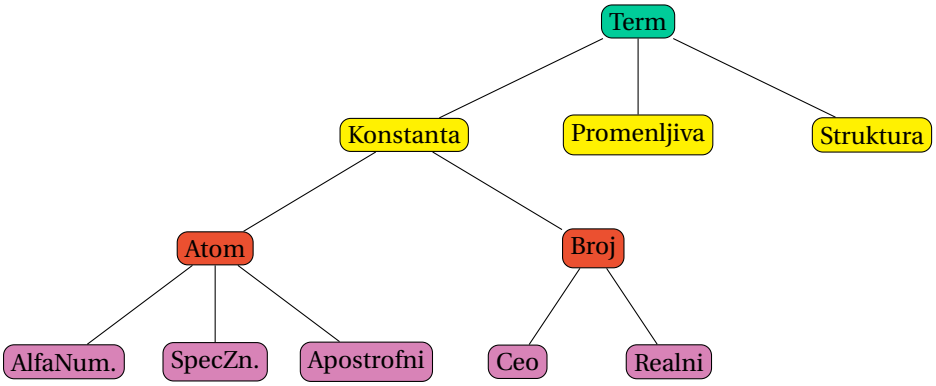
```
?- write('Hello world!'), nl.
Hello world!
true.
```

8.2 Sintaksa

Termovi su *osnovni gradivni elementi* Prologa i širi su od pojma terma logike prvog reda, tj.

$$Term\_Prologa \neq Term\_logike\_prvog\_reda$$

.



Slika 3: Sintaksa

Promenljive se zapisuju velikim početnim slovom ili simbolom _ (simboim _ počinju *imena anonimnih promenljivih čije vrednosti nisu bitne*). Strukture ili predikati se grade od atoma i termova. Ako je p atom, a $t_1, t_2, \ldots, t_n$ termovi, onda je $p(t_1, t_2, \ldots, t_n)$ term.

8.3 Programi

Činjenice su Hornove klauze bez negiranih literala (dakle, samo B). Pomoću činjenica opisuju se svojstva objekata, odnosno relacije između objekata.

```
zivotinja(panda) .
biljka(bambus) .
jede(panda, bambus) .
```

Činjenice su konstrukcije Prologa koje se sastoje iz **funktora** iza kojih slede argumenti između malih zagrada. Ukoliko ima više argumenata, razdvajaju se zapetama. Funktor predstavlja *naziv relacije*, a argumenti nazive objekata. Imena činjenica treba zadavati u skladu sa njihovim značenjem(semantikom). Prolog prihvata činjenice kao *apsolutne istine* (aksiome) i ne proverava njihovu tačnost.

U tom smislu činjenice u Prologu su:

```
biljka(panda) .
zivotinja(bambus) .
jede(bambus, panda) .
```

Skup činjenica formira *bazu činjenica* (bazu podataka).

Pravila su Hornove klauze u punom obliku. Pravila predstavljaju opšta tvrđenja o objektima i relacijama među objektima. To su konstrukcije koje se sastoje iz glave i tela povezanih vrat-simbolom (koji odgovara implikaciji).

```
GLAVA :- TELO.
```

Pravila su uslovne rečenice oblika: Važi GLAVA ako važi TELO.

Ako želimo da saopštimo da sve što poseduje Ana poseduje i Pavle, ne moramo navoditi sve činjenice o posedovanju predmeta za Pavla, već je dovoljno navesti pravilo:

```
poseduje(pavle, X) :- poseduje(ana, X).
```

```
sunce(beograd).
kisa(beograd).
duga(X) :- kisa(X), sunce(X).
```

Pravila mogu uključivati više različitih predikata razdvojenih zarezom (*zarez odgovara logičkoj konjunkciji*).

```
roditelj(A,B) :- dete(B,A).
dete(A,B) :- roditelj(B,A).
```

Za razliku od činjenica, u pravila su uključene promenljive. U definisanju pravila, može se koristiti i rekurzija, ali se ne mogu koristiti beskonačni ciklusi.

Takođe, levostrane rekurzije **nisu** dozvoljene.
Na primer, *pravilo 1.* je u redu: Ali *pravilo 2.* (iako deklarativno ima isti smisao) će dovesti do greške prilikom generisanja rešenja. Ovo je posledica načina na koji Prolog dolazi do rešenja (o čemu će biti više reči kasnije).

Pravilo 1.

```
macka(X) :- majka(Y,X), macka(Y).
```

Pravilo 2.

```
macka(X) :- macka(Y), majka(Y,X).
```

Baza znanja

Pomoću činjenica i pravila saopštavaju se Prologu informacije o relacijama i objektima. Činjenice i pravila se zajedničkim imenom nazivaju **tvrđenja**. Skup činjenica i pravila određuje **bazu znanja** (bazu podataka u širem smislu).

Jedino 'znanje' kojim raspolaže Prolog, u vezi sa problemom koji rešava, nalazi se u bazi znanja. **Pretpostavka zatvorenosti**(eng. *Closed-World Assumption*) govori da je netačno sve što nije eksplicitno navedeno kao tačno. Prolog nalazi rešenja (daje odgovore na postavljene upite) korišćenjem činjenica i pravila iz baze znanja.

Upiti su Hornove klauze bez nenegativnog literala ($A_1, A_2, \ldots, A_n$). To su konstrukcije Prologa preko kojih korisnik komunicira sa bazom znanja. Upiti se najčešće realizuju u interaktivnom radu korisnika sa računarom. To je moguće ako korisniku stoji na raspolaganju Prolog-mašina (interpretator za Prolog sa pratećim softverskim komponentama).

Interpretator

`?-`	Odzivni znak (prompt) interpretatora
`help`	Pomoć
`halt`	Izlazak iz interpretatora
`assert`	Ubacivanje činjenica i pravila, npr `assert(zivotinja(panda))`
`listing`	Proveravanje spiska postojećih činjenica i pravila

Najjednostavniji upiti služe za ispitivanje da li se činjenice nalaze u bazi podataka. Upiti su konstrukcije koje se **završavaju tačkom**.

```
?-zivotinja(panda).
yes
?-zivotinja(bambus).
no
```

Za postavljanje složenijih upita, tj. za dobijanje i drugačijih odgovora od `yes` i `no`, potrebno je koristiti *promenljive*.

```
?-zivotinja(X).
X=panda
?-biljka(Y).
Y=bambus
```

Svaki upit Prolog "shvata" kao cilj koji treba ispuniti (dostići). Prolog pokušava da instancira promenljive i prikaže njihovu vrednost. Generiše odgovore upoređujući upit sa bazom podataka od početka ka kraju.
Jedan cilj (upit) može se sastojati iz nekoliko podciljeva. Na taj način postavljaju se složeniji upiti. Ukoliko cilj sadrži više podciljeva, podciljevi se razdvajaju *zapetom* (,). Ovde zapeta ima ulogu *operatora konjunkcije*.

```
?- jede(panda, Nesto), jede(medved, Nesto).
```

Ovo se shvata kao pitanje: da li postoji `Nesto` što jede i `panda` i `medved` i šta je primer toga?

Prolog ispunjava cilj tako što ispunjava **svaki potcilj** i to **s leva u desno**.

Primer 1.

```
planeta(merkur).
planeta(venera).
planeta(zemlja).
planeta(mars).
planeta(venera).
planeta(saturn).
planeta(jupiter).
planeta(uran).
planeta(neptun).
veca(venera, merkur).
veca(zemlja, venera).
veca(uran, zemlja).
veca(saturn, uran).
manja(saturn, jupiter).
veca(X,Y) :- zvezda(X), planeta(Y).
zvezda(sunce).
veca(X,Y) :- planeta(Z), veca(X,Z), veca(Z,Y).
veca(X,Y) :- manja(Y,X).
manja(venera, saturn).
manja(zemlja, jupiter).
```

8.4 Sistemski predikati i operatori

Unifikacija (ujednačavanje) je jedna od najvažnijih operacija nad termima. Simbol za ovu operaciju u Prologu je =. Neformalno, unifikacija nad termima T i S vrši se na sledeći način:

- Ako su T i S konstante, unifikuju se ako predstavljaju istu konstantu
- Ako je S promenljiva, a T proizvoljan objekat, unifikacija se vrši tako što se termu S dodeli T. Obrnuto, ako je S proizvoljan objekat, a T promenljiva, onda T primi vrednost terma S.
- Ako su S i T strukture, unifikacija se može izvršiti ako:
  - imaju istu arnost i jednake dominantne simbole (*funktore*) **i**
  - sve odgovarajuće komponente se mogu unifikovati.

Logika prvog reda je veoma apstraktna notacija koja nema algoritamsku prirodu. Zato je prirodno da jezici, kao sto je Prolog, moraju da sadrže proširenja koja jeziku daju algoritmičnost, kao i proširenja potrebna za komunikaciju sa spoljašnjim svetom. *Sistemski predikati* su predikati koji su ugrađeni u sam jezik.
Bez sistemskih predikata Prolog je čist deklarativni jezik. Svaka verzija Prologa ima neke specifične sistemske predikate. Dodavanjem predikata dodaju se propratni efekti. Najveći broj operatora zahteva konkretizovane (instancirane) promenljive. Na primer, $X < Y$, zahteva konkretizovane promenljive X i Y.

Operatori jednakosti i nejednakosti		<, >, =<, >=	relacijski operatori
=, \=	(unifikacija)	+ -, *, /, mod, is	aritmetički operatori
is	(dodela)	true, fail, not, !	
=:=, =/=	(aritmetika)	repeat, call	Predikati
==, \==	(identitet)	op	Definisanje korisničkih tipova
user consult reconsult	Rad sa datotekama (učitavanje baze znanja)	var nonvar atom integer atomic	Klasifikacija terma
listing , asserta , assertz retract,clause	Rad sa klauzama	functor , arg ..= , name	Rad sa strukturama
read, get , write , nl tab , put , display	Rad sa ulazom i izlazom	see , seen , seeing tell , told , telling	Rad sa datotekama

Primer 2.

```
dana_u_mesecu(31, januar, _).
dana_u_mesecu(29, februar, G):- prestupna(G).
dana_u_mesecu(28, februar, G):- not(prestupna(G)).
dana_u_mesecu(31, mart, _).
dana_u_mesecu(30, april, _).
dana_u_mesecu(31, maj, _).
dana_u_mesecu(30, juni, _).
dana_u_mesecu(31, juli, _).
dana_u_mesecu(31, avgust, _).
dana_u_mesecu(30, septembar, _).
dana_u_mesecu(31, oktobar, _).
dana_u_mesecu(30, novembar, _).
dana_u_mesecu(31, decembar, _).
prestupna(G) :- je_deljivo(G,400).
prestupna(G) :- not(je_deljivo(G,100)), je_deljivo(G,4).
je_deljivo(X,Y) :- 0 is X mod Y.
```

Primer 3.

```
vojnik('Aca Peric', 183, 78).
vojnik('Milan Ilic', 192, 93).
vojnik('Stanoje Sosic', 173, 81).
vojnik('Sasa Minic', 162, 58).
vojnik('Dragan Sadzakov', 180, 103).
vojnik('Pera Peric', 200, 80).
vojnik('Rade Dokic', 160, 56).
zadovoljava(Ime) :- vojnik(Ime, Visina, Tezina),
                    Visina>170, Visina<190, Tezina =< 95,
                    Tezina>60.
otpada(Ime) :- vojnik(Ime, _, Tezina),Tezina =< 60.
otpada(Ime) :- vojnik(Ime, Visina, _), Visina>200.
```

Metaprogramiranje u Prologu

Prolog programi su jednoobrazni – podaci i programi izgledaju isto. To daje mogućnost jednostavnog metaprogramiranja: kreiranje programa koji u fazi izvršavanja mogu da stvaraju ili menjaju druge programe, ili da sami sebe proširuju. Najjednostavnije metaprogramiranje u Prologu je dodavanje činjenica u bazu znanja prilikom izvršavanja programa (`asserta`, `assertz`), ili njihovo izbacivanje iz baza podataka (`retract`, `retractall`). Predikati `functor`, `arg`, `=..` omogućavaju i zaključivanje o novim predikatima za vreme izvršavanja programa.

Liste

Liste su važne u Prologu. Prestavljaju niz uređenih elemenata proizvoljne dužine. Element liste može biti bilo koji term pa čak i druga lista. Lista je: prazna lista u oznaci `[]`, struktura `.(G,R)` gde je `G` ma koji term, a `R` lista.
Funktor u strukturi liste je tačka (`.`), prvi argument naziva se glava, a drugi rep. Drugi argument je uvek lista. Zbog rekurzivne definicije liste, rekurzija je najpogodniji način za obradu listi. Ako pođemo od definicije liste, imamo:

- `[]` prazna lista (lista koja ne sadži ni jedan element)
- `.(a, [])` je jednočlana lista, gde je `a` nekakav term
- `.(b, .(a,[]))` je dvočlana lista (`a` i `b` su termi)
- `.(c, .(b, .(a,[])))` je tročla (`a`, `b` ic su termi)
- ...

Zapis liste `.(c, .(b, .(a,[])))` je glomazan i nepregledan, zato se koristi zapis: `[]`, `[a]`, `[a,b]`, `[a,b,c]`, ...

```
Lista                        Glava                    Rep
[iva, mara, dara]            iva                      [mara, dara]
[[voz, tramvaj],trolejbus]   [voz, tramvaj]           [trolejbus]
[a]                          a                        []
[]                           -                        -
```

```
?- duzina(Lista, Duzina).
duzina([],0).
duzina([G|R],D) :- duzina(R,D1), D is D1+1.

?- suma(Lista, ZbirElemenataListe).
suma([], 0).
suma([Glava|Rep],S) :-suma(Rep,S1), S is Glava+S1.
```

8.5 Deklarativna i proceduralna interpretacija

Prolog-konstrukcije se mogu interpretirati deklarativno i proceduralno.

Činjenica	`p(a)`
Deklartivno	p(a) je istinito
Proceduralno	zadatak p(a) je izvršen

Pravilo	`p(a,X)`
Deklartivno	Za svako X istinito je p(a,X)
Proceduralno	Za svako X, zadatak p(a,X) je izvršen

Pravilo	`p(X) :- q(X), r(X)`
Deklarivno	Za svako X, p(X) je istinito ako je istinito q(X) i r(X)
Proceduralno	Da bi se izvršio zadatak p(X), prvo izvrši zadatak q(X), a zatim izvrši zadatak r(X)

U proceduralnom tumačenju se određuje i redosled ispunjavanja cilja.

Upit	`?- p(X)`
Deklartivno	Da li postoji vrednost promenljive X za koji važi p(X)
Proceduralno	Izračunavanjem ostvari cilj p(X) i nadi vrednost promenljive X za koju važi svojstvo p

8.6 Stablo izvođenja

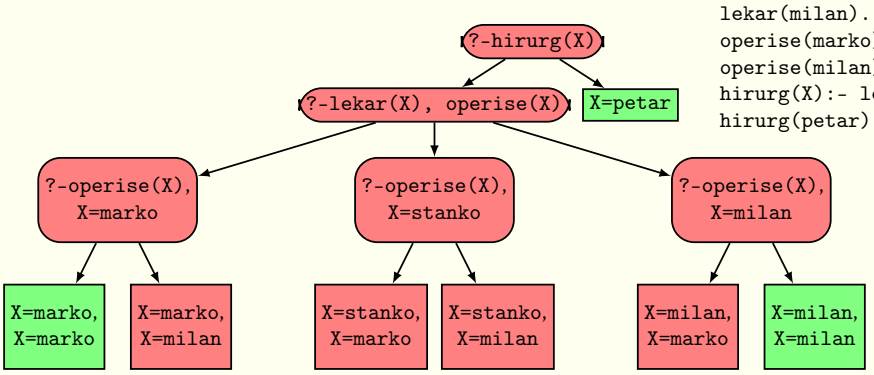
Svaki upit Prolog tretira kao cilj koji treba dostići (ostvariti, ispuniti) – proceduralna interpretacija. Ostvarivanje cilja Prolog-mašina čini pokušavajući da dokaže saglasnost cilja sa bazom znanja. U tom procesu baza znanja se pregleda od vrha ka dnu i moguće su dve situacije:

- pronađeno je tvrđenje koje se uparuje sa postavljenim ciljem (uspešno je ostvaren cilj – uspeh)
- nije pronađeno tvrđenje koje se uparuje sa postavljanjim ciljem (cilj nije ispunjen – neuspeh)

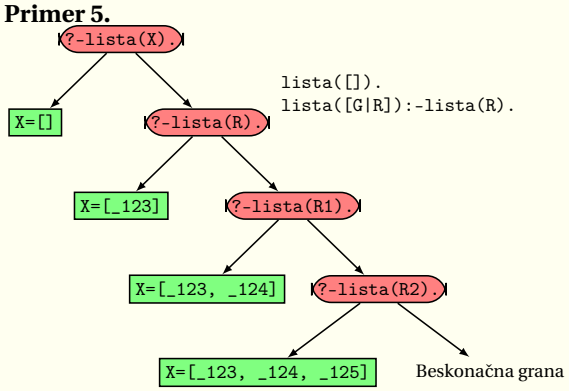
U slučaju uspeha, korisnik može zahtevati da se ponovo dokaže saglasnost cilja sa bazom podataka (pronalaženje novog rešenja).
Stablo izvođenja omogućava slikovit prikaz načina rešavanja problema u Prologu.

Primer 4.

```
lekar(marko).
lekar(stanko).
lekar(milan).
operise(marko).
operise(milan).
hirurg(X):- lekar(X), operise(X).
hirurg(petar).
```



Primer 5.



Terminologija

Stablo izvođenja – stablo pretrage. Ono se sastoji iz grana i čvorova. Grana u stablu izvođenja može biti konačna ili beskonačna. Stablo izvođenja je konačno ako su sve grane u njemu konačne, u suprotnom je beskonačno.

Čvorovi koji su označeni upitom (ciljem) nazivaju se **nezavršenim**. (Iz njih se izvode sledeći čvorovi). *Listovi* u stablu izvođenja nazivaju se **završnim** čvorovima. *Grane* koje vode do završnih čvorova su **konačne**. Grane koje nemaju završne čvorove su **beskonačne**.

Ako završni čvor *daje rešenje*, naziva se **čvor uspeha** (označen zelenom bojom), a odgovarajuća grana je **grana uspeha**. Završni čvor koji ne predstavlja rešenje je **čvor neuspeha** (označen crvenom bojom), a odgovarajuća grana je **grana neuspeha**.

Veza stabla izvođenja i metoda rezolucije

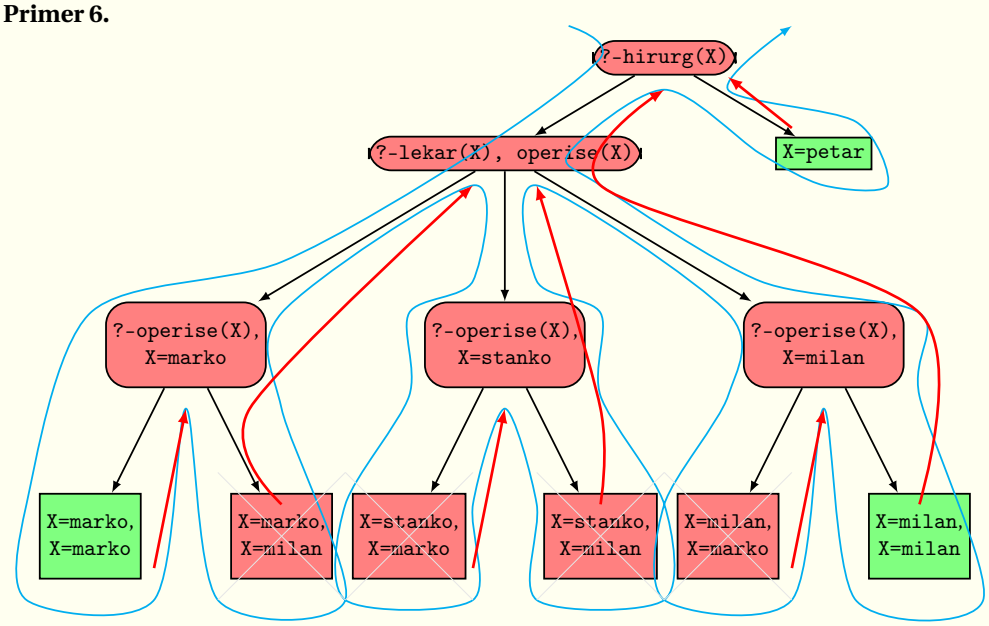
U smislu *deklarativne* semantike, stablo izvođenja odgovara poretku primene pravila rezolucije na postojeći skup činjenica i pravila. U smislu *proceduralne* semantike, stablo izvođenja odgovara procesu ispunjavanja ciljeva i podciljeva. Redosled činjenica i pravila u bazi znanja određuje redosled ispunjavanja ciljeva i izgled stabla izvodjenja.

S obzirom da su sve klauze Hornove, proces izvođenja je relativno jednostavan, ali efikasnost može i dalje da bude problem. Implementacija rezoluije može da se ostvari izvođenjem u širinu i izvođenjem u dubinu. Izvođenjem u širinu, radi se paralelno na svim podciljevima datog cilja, dok se izvođenjem u dubinu najpre izvede jedan podcilj, pa se dalje nastavlja sa ostalim podciljevima. Koja implementacija bi bila bolja?

U Prologu je implementacija ostvarena izvođenjem u dubinu, jer su na taj način manji memorijski zahtevi. To znači da redosled tvrđenja u bazi znanja utiče i na efikasnost pronalaženja rešenja. Takođe, redosled može da utiče i na konačnost najlevlje grane, pa time i na to da li će rešenje uopšte biti pronađeno.

Način obilaska stabla izvođenja i nalaženje rešenja Pomoću stabla izvođenja potpuno je određen prostor izvođenja za jedan cilj. Njega čine svi putevi koji vode od korena stabla do njegovih čvorova.

Primer 6.



Kako je redosled izvođenja poznat i deterministički određen, to se može iskoristiti za dobijanje na efikasnosti Prolog programa time što se činjenice i pravila u bazi znanja ređaju u odgovarajućem redosledu. Takođe, na efikasnost utiče i davanje pogodnog redosleda podciljeva u pravilima. Ovo narušava pravila deklarativnosti po kojima redosled ne bi trebao da utiče na izvršavanje programa.

8.7 Operator sečenja

Sinonimi: operator sečenja, `cut` operator, rez operator, operator odsecanja.

To je sistemski operator koji omogućava brže izvršavanje programa i uštedu memorijskog prostora kroz eksplicitnu kontrolu backtracking-s. Ovaj operator je zapravo **cilj koji uvek uspeva**. Operator sečenja odseca pojedine grane na stablu pretraživanja, pa samim tim smanjuje prostor pretraživanja. To omogućava brže nalaženje rešenja. U isto vreme ne moraju se pamtiti mnogoprojne tačke prilikom traženja sa vraćanjem, što dovodi do uštede memorijskog prostora.

Neka imamo tvrđenje: `A :- B1, B2, ..., Bk, !, ..., Bm`
Kada se neki cilj G unifikuje sa A, aktivira se prethodno tvrđenje. Tada se cilj G naziva roditeljski cilj za A. Kada se dođe do reza, uspeli su podciljevi: B1, B2, ..., Bk. Sečenje uspeva i rešenje B1, B2, ..., Bk se "zamrzava", tj. predikat onemogućava traženje alternativnih rešenja. Takođe se onemogućava ujedinjavanje cilja G sa glavom nekog drugog predikata koji je u bazi podataka iza navedenog tvrđenja.

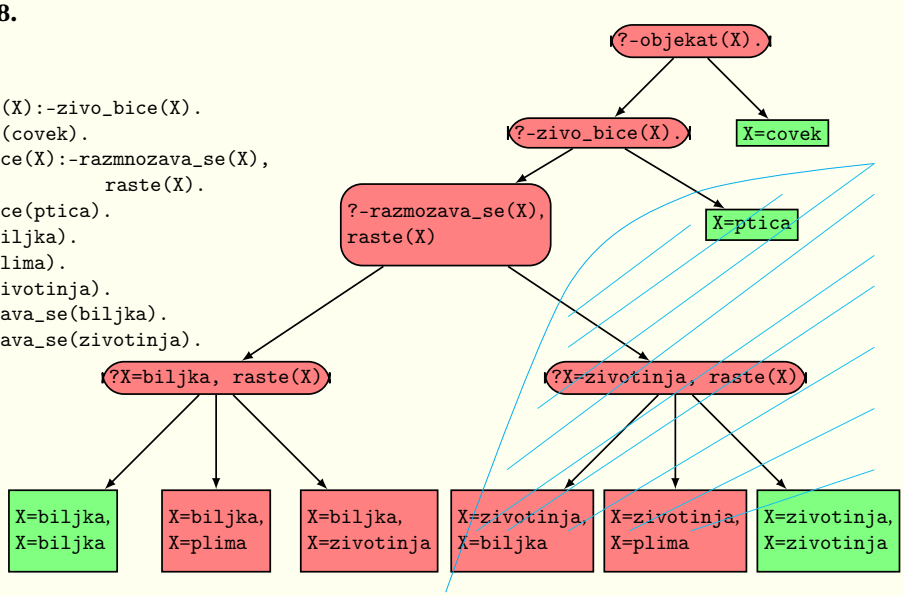
Primer 7.

```
A :- P, Q, R, !, S, T.
A :- U.
G :- L, A, D.
?- G.
```

U startu, traženje sa vraćanjem moguće je samo za P, Q, R. Kad uspe R uspeva i sečenje i alternativna rešenja se više ne traže. Alternativno rešenje: `A :- U`, takođe se ne razmatra. Alternativna rešenja su moguća između S i T, tj desno od operatora sečenja je moguć backtracking. Ukoliko se ne nađe rešenje za fiksirane vrednosti P, Q, R, onda je dozvoljen backtracking.

Primer 8.

```
objekat(X):-zivo_bice(X).
objekat(covek).
zivo_bice(X):-razmnozava_se(X),
              raste(X).
zivo_bice(ptica).
raste(biljka).
raste(plima).
raste(zivotinja).
razmnozava_se(biljka).
razmnozava_se(zivotinja).
```



Predikat sečenja može se uporediti sa GOTO-naredbom u proceduralnim jezicima. Narušava deklarativni stil programiranja i može da proizvede neželjene efekte (greške). Pogrešna upotreba operatora sečenja je najčešći uzrok grešaka programiranja u Prologu. Ipak, ovaj predikat je često neophodan za dobijanje efikasnih rešenja i treba ga koristiti (ali oprezno!). Ako se predikat sečenja upotrebljava tako da ne narušava deklarativno svojstvo predikata tj. ne menja njegovu semantiku niti skup rešenja (već samo pojačava njegova deterministička svojstva i ubrzava proces izračunavanja), onda se naziva **zeleni predikat sečenja**. Treba biti veoma oprezan sa upotrebom predikata sečenja.

Primer 9. Primena operatora sečenja
Kada želimo da saopštimo Prologu: "Nađeno je potrebno rešenje, ne treba dalje tražiti!"

```
auto(mercedes, 200000, 10000).
auto(skoda, 100000, 5000).
auto(fiat, 50000, 4000).
auto(bmv, 12000, 12000).
zadovoljava(M) :- auto(M, K, C), C<5000, !.
```

Primer 10. Primena operatora sečenja
Kada želimo da saopštimo Prologu:"Nađeno je jedinstveno rešenje i ne treba dalje tražiti!"

```
kolicnik(N1, N2, Rez):-ceo_broj(Rez),
                P1 is Rez*N2,
                P2 is (Rez+1)*N2,
                P1=<N1, P2>N1, !.
ceo_broj(0).
ceo_broj(X) :- ceo_broj(Y), X is Y+1.
```

Primer 11. Primena operatora sečenja
Kada želimo da saopštimo Prologu:"Na pogrešnom si putu, završiti pokušaj zadovoljenja cilja!".

```
inostrani(dzon).
inostrani(matrin).
dohodak(dzon, 5000).
dohodak(matrin, 10000).
dohodak(pera, 20000).
placa_porez(X) :- inostrani(X), !, fail.
placa_porez(X) :- dohodak(X,D), D>10000.
```

Koristi se u kombinaciji sa `fail`-predikatom. Može samo da se koristi za proverne svrhe, ne i za generisanje rešenja.

8.8 Svojstva Prologa

Prolog nije čist deklarativni jezik (kontrola rezolucije i backtracking-a). On je restrikcija logike prvog reda na formule koje se mogu svesti na Hornove klauze. Hornovim klauzama ne mogu se opisati sva tvrđenja logike prvog reda.
Prolog može da dokaže da je neki cilj ispunjen, ali ne i da neki cilj nije ispunjen. To je posledica oblika Hornovih klauza $A:-B_1,B_2,\ldots,B_n$ ako su sve B_i tačne, možemo da zaključimo da je A tačno, ali ne postoji način da zaključimo da A nije tačno. Prema tome, ukoliko Prolog ne može da dokaže da je neki cilj ispunjen, on prijavljuje da cilj nije ispunjen, iako je to zapravo nemogućnost dokazivanja tačnosti, a ne dokazivanje netačnosti. Preciznije, Prolog je **true/fail** sistem, a ne true/false i to treba imati na umu.
Operator NOT nije operator negacije u smislu logičke negacije. Naziva se **"negacija kao neuspeh"**, `not(cilj)` uspeva ukoliko `cilj` ne uspeva. Zapravo, logičko NOT ne može da bude sastavni deo Prologa što je posledica oblika Hornovih klauza. To znači da, ukoliko imamo na primer naredni oblik `not(not(cilj))` to ne mora da bude ekvivalentno sa `cilj`, što može da dovede do raznih problema i neočekivanih rezultata. Zbog toga i sa upotrebom operatora NOT treba biti veoma pažljiv i koristiti ga samo za instancirane promenljive.

Primer 12. operator not

```
vozac(janko).                                  ?-dobar_vozac(X).
vozac(marko).                                  no
vozac(petar).                                  ?-dobar_vozac(petar).
pije(janko).                                   yes
pije(marko).                                   ?-pije(X).
dobar_vozac(X) :- not(pije(X)), vozac(X).      X=janko;
                                               X=marko;
                                               no
                                               ?-not(not(pije(X))).
                                               yes
```

Primer 13. operator NOT

```
vozac(janko).                                  ?-dobar_vozac(X).
vozac(marko).                                  X=petar;
vozac(petar).                                  no
pije(janko).                                   ?-dobar_vozac(petar).
pije(marko).                                   yes
dobar_vozac(X) :- vozac(X),not(pije(X)).
```

Objasniti naredne rezultate:

```
?- not(X==1), X=1.
X = 1
yes
?- X=1, not(X==1).
no
```

X nije instancirano, pa stoga nije identično jedinici, i not operator uspeva. Da bi uspeo i drugi cilj, X se unifikuje sa 1.

```
?- not(X==1), X=1.
X = 1
yes
```

U ovom slučaju, X se najpre unifikuje sa 1, i kako je 1 == 1 not ne uspeva.

```
?- X=1, not(X==1).
no
```

Generisanje efikasnih algoritama

Osnovni cilj logičkog programiranja je da se obezbedi programiranje takvo da programer da specifikaciju šta program treba da uradi bez specifikacije na koji način to treba da se ostvari. Visok nivo apstrakcije i deklarativnosti nosi sa sobom cenu neefikasnosti.

Sortiranje može jednostavno da se definiše kao generisanje permutacije i provera da li je permutacija sortirana. Na primer, kriterijum da li je niz sortiran može u Prologu da se napise:

```
sorted([]).
sorted([X]).
sorted([X, Y | List]) :- X =< Y, sorted([Y|List]).
```

Međutim, sam algoritam sortiranja je potpuno neefikasan: problem je što nije dat način na koji da se izvrši sortiranje i jedini način da se to ostvari je enumeracijom svih permutacija liste dok se ne kreira ona permutacija koja zadovoljava svojstvo sortiranosti liste. Zapravo, ne postoji način da se opis sortiranosti liste prevede u efikasan način sortiranja elemenata liste – rezolucija ne može da generiše efikasan algoritam. Prema tome, ako želimo efikasno izvršavanje, moramo da damo specifikaciju algoritma i u Prologu. Možda ćemo jednog dana doći do tačke da su sami programski sistemi u mogućnosti da otkriju dobre algoritme iz deklarativnih specifikacija, ali to još uvek nije slučaj.

Moduli

Kreiranje mogula u Prologu propisano je ISO standardom, ali ne podržavaju svi kompajleri ovo svojstvo što značajno otežava razvoj kompleksnog softvera. Postoje nekompatibilnosti između sistema modula različitih Prolog kompajlera. Takođe, većina kompajlera implementira i dodatne funkcionalnosti koje nisu podržane standardom, što onemogućava kompatibilnost između kompajlera.

- Prolog je Tjuring-kompletan jezik, to se može pokazati korišćenjem Prologa da simulira Tjuringovu mašinu.
- Prolog je netipiziran jezik, postoje razni pokušaji uvođenja tipova u Prolog.
- Prolog ima mehanizam prepoznavanja i optimizacije repne rekurzije, tako da se ona izvršava jednako efikasno kao i petlje u proceduralnim jezicima.
- Predikati višeg reda su predikati koji uzimaju jedan ili više predikata kao svoje argumente. To izlazi iz okvira logike prvog reda, ali ISO standard propisuje neke predikate višeg reda (na primer predikat `call`)

8.9 Programiranje ograničenja u logičkom programiranju

Programiranje ograničenja je deklarativno programiranje. U njemu se relacije između promenljivih zadaju u vidu ograničenja. Od sistema se zatim očekuje da izračuna rešenje, tj da izračuna vrednosti promenljivih koje zadovoljavaju data ograničenja. Ograničenja se razlikuju od ograničenja u imperativnoj paradigmi. Na primer, $x < y$ u imperativnoj paradigmi se evaluira u tačno ili netačno, dok u paradigmi ograničenja zadaje relaciju između objekata x i y koja mora da važi.

Ograničenja mogu da budu različitih vrsta, na primer, ograničenja iskazne logike (A ili B je tačno), linearna ograničenja ($x < 15$), ograničenja nad konačnim domenima. Rešavanje ograničenja vrši se različitim rešavačima, npr SAT i SMT. Programiranje ograničenja ima primene pre svega u operacionim istraživanjima (tj u rešavanju kombinatornih i optimizacionih problema).

8.9.1 B-Prolog

B-Prolog (http://www.picat-lang.org/bprolog/)

Nastao je 1994. Komercijalni je proizvod, ali se može koristiti za učenje i istraživanja besplatno. Implementira ISO standard Prologa, ali i razna proširenja. B-Prolog ima bogat sistem za programiranje ograničenja. Podržava programiranje ograničenja na različitim domenima, npr na konačnom i iskaznom domenu.

Programiranje ograničenja se sastoji od tri dela:

- Generisanje promenljivih i njihovih domena.
- Generisanje ograničenja nad promenljivama
- Obeležavanje (labeling) – instanciranje promenljivih

Za programiranje ograničenja uvode se novi predikati, koje ćemo razmotriti kroz primere.
Generisanje promenljivih i njihovih domena:

- Definisanje domena: `Vars in D` ili `Vars :: D`
- `D` se definiše kao `pocetak..korak..kraj`
- `1..5..20` definiše domen `1, 6, 11, 16`
- Korak nije obavezan, podrazumeva se da je jedan.
- Postoje predikati za opšta ograničenja, a mogu se zadavati numerička ograničenja.
- Numerička počinju sa #, npr #<, #<=,...
- Labeling se odnosi na instanciranje promenljivih i na njihovo prikazivanje.

```
?- X in 1..10, Y in 5..2..30, X#>=Y, labeling(X).
X = 5
Y = 5 ?;
X = 6
Y = 5 ?;
X = 7
Y = _0490::[5,7] ?;
X = 8
Y = _0490::[5,7] ?;
X = 9
Y = _0490::[5,7,9] ?;
X = 10
Y = _0490::[5,7,9] ?;
no
```

Promenljive: X i Y
Domeni: X je bilo koji broj od 1 do 10
Y su neparni brojevi od 5 do 30
Ograničenje: $X \geq Y$
Instanciranje: po X

```
?- X in 1..10, Y in 5..2..30, X#>=Y, labeling(X), labeling(Y).
X = 5
Y = 5 ?;
X = 6
Y = 5 ?;
X = 7
Y = 7 ?;
X = 8
Y = 5 ?;
...
```

Promenljive: X i Y
Domeni: X je bilo koji broj od 1 do 10
Y su neparni brojevi od 5 do 30
Ograničenje: $X \geq Y$
Instanciranje: po X i Y

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

| ?- consult(primer1).
| ?- sendmoremoney(Vars).
Vars = [9,5,6,7,1,0,8,2] ?;
no
```

```
|X1 X2 | |
|X3 X4 X5|
| | X6 X7|
crossword(Vars):-
        Vars=[X1,X2,X3,X4,X5,X6,X7],
        Words2=[(0'I,0'N),
                (0'I,0'F),        %IF
                (0'A,0'S),        %AS
                (0'G,0'O),        %GO
                (0'T,0'O)],       %TO
        Words3=[(0'F,0'U,0'N),    %FUN
                (0'T,0'A,0'D),    %TAD
                (0'N,0'A,0'G),    %NAG
                (0'S,0'A,0'G)],   %SAG
        [(X1,X2),(X1,X3),(X5,X7),(X6,X7)] in Words2,
        [(X3,X4,X5),(X2,X4,X6)] in Words3,
        labeling(Vars),
        format("~s~n",[Vars]).
```

```
| ?- consult(primer2).
| ?- crossword(Vars).
ASSAGGO
Vars = [65,83,83,65,71,71,79] ?;
INNAGGO
Vars = [73,78,78,65,71,71,79] ?;
no
```

```
queens(N):-
        length(Qs,N),
        Qs :: 1..N,
        foreach(I in 1..N-1, J in I+1..N,
                (Qs[I] #\= Qs[J],
                abs(Qs[I]-Qs[J]) #\= J-I)),
        labeling([ff],Qs),
        writeln(Qs).
```

```
| ?- consult(primer3).
| ?- queens(5).
[1,3,5,2,4]
yes
| ?- queens(8).
[1,5,8,6,3,7,2,4]
yes
| ?- queens(10).
[1,3,6,9,7,10,4,2,5,8]
yes
| ?- queens(15).
[1,3,5,14,11,4,10,7,13,15,2,8,6,9,12]
yes
```

9 Imperativna paradigma

Imperativna paradigma je prvonastala programska paradigma. Nastala je pod uticajem Fon Nojmanove arhitekture računara (tj. ova arhitektura nameće ovaj stil programiranja, prilagođen mašini, a ne čoveku). U rešavanju problema prednost se daje algoritmima pa se imperativno programiranje naziva i *Algoritamski orijentisano programiranje.* Podaci i algoritmi postoje nezavisno.

$$Podaci + Algoritmi = Programi$$

Kao što se u govornom jeziku zapovedni način (*imperativ*) koristi za izražavanje naredbi, tako se imperativni programi mogu posmatrati kao niz naredbi koje računar treba da izvrši. Pored naredbi, ključan je i redosled izvršavanja naredbi – procedura. Imperativna paradigma i proceduralna paradigma se često koriste kao sinonimi. Imperativno programiranje karakteriše izračunavanje u terminima naredbi koje menjaju *stanje* programa.

Stanje programa čine sve sačuvane informacije, u datom trenutku vremena, kojima program ima pristup (preko memorije). Izvršavanjem programa generiše se niz stanja. Prelaz iz jednog stanja u sledeće je određen komandama koje se izvršavaju. Svaki imperativni jezik obezbeđuje raznovrsne komande za modifikaciju stanja (manipulisanje) memorije.

Osnovna komanda za modifikaciju stanja memorije je **naredba dodele**. Ona vrši povezivanje imena i neke vrednosti, odnosno upisivanje konkretne binarne reči na odgovarajuću memorijsku lokaciju. Analogija između memorijskih i (proceduralno) jezičkih elemenata:

memorija	binarna reč	mem. registar	adresa
jezik	vrednost	promenljiva	ime

Deklaracijom promenljive u proceduralnom jeziku određuje se veličina memorijskog prostora za zapis promenljive. Na primer:

```
Pascal:  var name :  Type;
C:       type name;
```

Promenljiva se povezuje sa nekom vrednošću preko izraza, što se različito opisuje u različitim jezicima. Na primer:

```
Pascal:   V := E
C:        V = E
APL:      V <- E
```

Redosled povezivanja imena i vrednosti (dodeljivanje) utiče na vrednosti izračunavanja:

```
a = 5; b = 3; c = 7;
b = a+1;
c = a+b;        /* Redosled izvrsavanja ovih naredbi je bitan! */
```

Bitan je i redosled povezivanja (asocijativnost, prioriteti).

```
b = a + b * c / 2;
```

Kontrola toka se ostvaruje kroz različite instrukcije i apstrakcije. Najveći broj konstrukcija u imperativnim jezicima je odraz hardverske implementacije. Imperativna paradigma prolazi kroz različite faze razvoja, a svaku fazu karakteriše viši nivo apstrakcije u odnostu na arhitekturu računara, i udaljavanje od asemblerskog jezika. Neki autori svaku fazu imperativne paradigme izdvajaju u posebnu paradigmu, a neki ih tretiraju kao potparadigme imperativne paradigme.

- Operaciona (pod)paradigma
- Strukturna (pod)paradigma
- Proceduralna (pod)paradigma
- Modularna (pod)paradigma

9.1 Operaciona paradigma

FORTRAN ALGOL COBOL BASIC ...

Prva faza programiranja. Programiranje je zasnovano na dosetkama, trikovima, naziva se i *trik programiranje*. Programi su pisani bez opštih pravila, korišćene su specifičnosti u radu računara, trikovi za uštedu memorije, pisanje samomodifikujućh programa. Dolazi do ogromne produkcije softvera krajem 60-ih godina prošlog veka.
Kontrola toka se sastoji od minimalnog skupa komandi (koji je često korišćen):

```
komanda ::=
        identifikator = izraz |                  (naredba dodele)
        komanda; komanda |                        (sekvenca)
        labela : komanda |                       (obelezavanje)
        GOTO labela |                            (naredba skoka)
        IF (log_izraz) [THEN] GOTO labela        (selekcija)
```

Ove upravljačke strukture su nastale kao odraz (analogoni) struktura u programu na mašinskom jeziku. Prema Fon Nojmanovom konceptu računara, instrukcije slede jedna za drugom (sekvenca naredbi u imperativnom jeziku), a redosled se može promeniti korišćenjem GOTO (Jump) naredbe. U imperativnim jezicima javljaju se 2 oblika GOTO naredbe (u IF naredbi i bez IF naredbe). Korišćenje prethodnih naredbi dovodi do pisanja nepreglednih programa, teških za modifikaciju (*"Špageti-programi"*). To je posledica prilagođavanja jezika mašini, a ne čoveku.

Špageti programiranje u C-u

```
#include <stdio.h>

main(){
     float x, y;
     int n;
     n = 1;
    l20: scanf(''%f'', &x);
         y=x;
         if(x<0) goto 130;
         y=-x;
    l30: printf(''y = %f\n'', y);
         n=n+1;
         if (n!=5) goto 120;
}
```

Špageti programiranje u Pascal-u

```
program idina;
     label 20, 30;
     var x,y:real;
         n:integer;
begin
    n:=1;
20: readln(x);
    y:=x;
    if x<0 then goto 30;
    y:=-x;
30: writeln(y);
    n:=n+1;
    if n<>5 then goto 20
end.
```

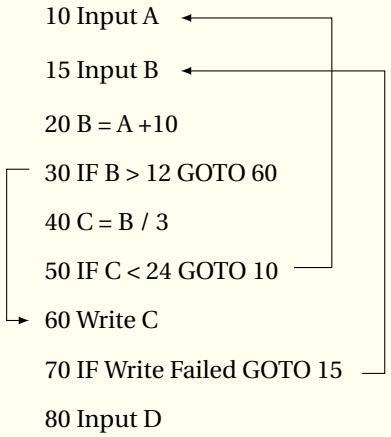
Špageti programiranje u C-u

```
main() {
     int i,n;
     float h, x0, x, y;
     printf ("Unsete h, x0 i n\n");
     scanf("%f%f%d",&h,&x0,&n);
     if (n<0) goto komentar;
     printf("Pocetak \n");
     i=0;
poc: x=x0+i*h;
     y=x*x;
     printf("x=%f, y=%f\n", x, y);
     if(i<n) goto uvecanje;
     return 0;
uvecanje: i=i+1; goto poc;
komentar: printf("Nekorektno zadato n\n");
}
```

Špageti programiranje u Pascal-u

```
program sagoto;
     label poc, kraj;
     var i,n: integer;
         h, x0, x, y: real;
begin
     writeln ('Unsete h, x0 i n');
     readln (h, x0, n);
     writeln ('Pocetak');
     i:=0;
poc: x:=x0+i*h;
     y:=x*x;
     writeln('x = ', x, ' y = ', y);
     if(i>=n) then goto kraj;
     i:=i+1;
     goto poc;
kraj:
    end.
```

Špageti programiranje



Problemi

- Teška čitljivost programa i održavanje programa.
- Kreirani softver je nepogodan za izmene i prilagođavanje novim situacijama.
- Softverska kriza 1970-ih, zbog loše prakse programiranja softver nije mogao da dostigne mogućnosti hardvera.
- Uzrok: nekontrolisana upotreba GOTO-naredbe.
- Rešenje: strukturno programiranje – disciplinovan pristupu programiranju, bez nekontrolisanih skokova i uz korišćenje samo malog broja naredbi za kontrolu toka programa.

9.2 Strukturna paradigma

C. Bohm i G. Jacopini su 1966. godine publikovali naučni rad u kojem su dokazali da se svaki prost program može izraziti pomoću 3 upravljačke strukture:

- sekvenca (naredba za naredbom)
- selekcija (odluka da li da se izvrši neka naredba zavisno od tačnosti ili netačnosti nekog uslova)
- ponavljanje (ponavljanje bloka koda vraćanjem na početak sve dok je ispunjen neki uslov)

Kasnije su usledili radovi: Dijkstre, Knutha i E. Ashcroft i Z. Manna, ... u kojima je pokazano da goto naredba nije neophodna. Dijkstra je napisao 1968. čuveno pismo *"Go To Statement Considered Harmful"*.
Strukturno programiranje nastaje kao nastojanje da zapis programa bude pregledniji. Akcenat je na programskim strukturama u kojima svaka komanda ima jednu ulaznu i jednu izlaznu tačku. Cilj je da se proceduralni jezik više prilagodi čoveku. Minimalan skup upravljačkih struktura čine:

```
komanda ::=
        identifikator := izraz       | (naredba dodele)
        komanda ; komanda            | (sekvenca)
        IF log_izraz THEN-grana
                     ELSE-grana      | (selekcija)
        WHILE log_izraz DO naredba   | (iteracija)
```

Zbog preglednijeg zapisa programa, najčešće se uvode dodatne upravljačke strukture, kao što su:

```
CASE (switch)
FOR
REPEAT-UNTIL (do-while)
```

Kontrolne naredbe su sintaksičke strukture preko kojih se definiše redosled u kojem se vrši dodeljivanje. Sve kontrolne strukture mogu da se svrstaju u 3 kategorije: sekvencijalna kompozicija, alternacija (selekcija) i iteracija. Programi zapisano pomoću ovih upravljačkih struktura su pregledniji, jasnihi i često kraći.

```
#include <stdio.h>
main() {
    float x,y;
    int
    n;
    n=1;
    l20: scanf("%f", &x);
        y=x;
        if(x<0) goto l30;
        y=-x;
    l30: printf("y = %f\n",y);
        n=n+1;
        if (n!=5) goto l20;
}
```

```
program idina;
    label 20, 30;
    var x,y:real;
    n:integer;
begin
    n:=1;
20: readln(x);
    y:=x;
    if x<0 then goto 30;
    y:=-x;
30: writeln(y);
    n:=n+1;
    if n<>5 then goto 20
end.
```

```
#include <stdio.h>
main() {
    float x,y;
    int
    n;
    printf("Unesite 4 realne vrednosti\n");
    for(n=1; n<5; n++) {
        scanf("%f", &x);
        y=x;
        if(x>=0) y=-x;
        printf("y = %f\n",y);
    }
}
```

```
program bezIdina;
    var x,y:real;
    n:integer;
begin
    writeln('Unesite 4 realne vrednosti');
    for n:=1 to 4 do
    begin
        readln(x);
        y:=x;
        if x>=0 then y:=-x;
        writeln('y = ', y);
    end
end.
```

9.3 Proceduralna paradigma

Apstrakcija kontrole toka – oodrutine (funkcije, procedure, metodi, korutine, potprogrami).
Podrutine predstavljaju apstrakciju niza naredbi. Poziv podrutine je poziv na apstrakciju, približava programiranje deklarativnosti. Podrutina izvršava svoje operacije u ime svog pozivaoca. Nastanak potprograma prethodi strukturnom programiranju. Svaka podrutina ima svoje lokalne podatke i algoritam, nezavisna je od ostalih. Razvija se mehanizam prenosa parametara. Nastaju korisnički definisani tipovi. Uvodi se vidljivost i doseg podataka. Podržava se ugnježdavanje podrutina. Podrutine su osnovni blokovi za održavanje modularnog programiranja.
U imperativnim programskim jezicima, podrutine su najčešće procedure i funkcije. Procedure nemaju povratnu vrednost, dok funkcije imaju. Jednim imenom se procedure i funkcije nazivaju potprogrami. Sintaksa potprograma je obično jednostavna:

```
Definicija potprograma:
        ime( parametar-lista ) { telo }
Poziv potprograma:
        ime( argument-lista )
```

Parametri se često nazivaju formalni parametri, a argumenti, aktuelni parametri. Sintaksa parametar-liste, tj. argument-liste je različita u različitim programskim jezicima (obično lako shvatljiva). Mogu se razlikovati:

Vrednosni parametri (in-parametri): služe samo za unos vrednosti u proceduru, mogu se menjati u proceduri, ali ne mogu izneti izmenjene vrednosti. Kao stvarni argumenti vrednosnih parametara obično se pojavljuju izrazi.

Promenljivi parametri (out-parametri): mogu uneti vrednost u proceduru, mogu biti menjani u proceduri i (najvažnije) zadržavaju izmenjene vrednosti po izlasku iz procedure. Kao stvarni argumenti promenljivih parametara obično su promenljive ili pokazivači.

Neki jezici imaju samo jedan način prenosa parametara, dok neki dozvoljavaju više načina. Osnovne vrste prenosa parametara:

Prenos po vrednosti: agrument se evaluira i kopira njegova vrednost u podrutinu (podrazumevan prenos u jezicima nakon Algol 60, Pascal, Delphi, Simula, Modula, Oberon, Ada, C, C++, ...)

Prenos po referenci: prenosi se referenca na argument, obično adresa (moguće odabrati u svim prethodno navedenim jezicima)

Prenos po rezultatu: na izlasku iz podrutine, vrednost parametara se kopira (prenosi) pozivajućoj rutini (Ada OUT parametri)

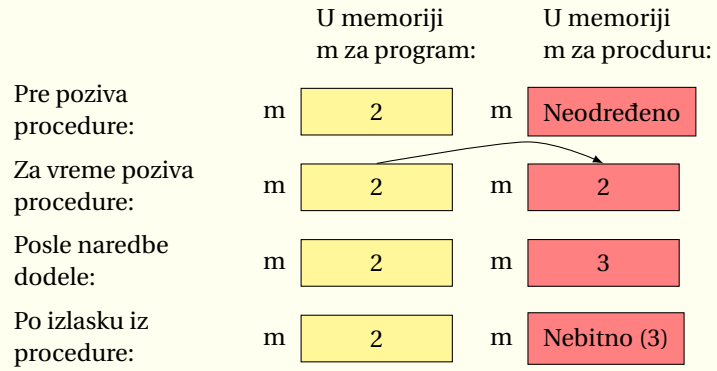
Prenos po vrednosti i rezultatu: vrednost parametra se kopira na ulasku i na izlasku iz podrutine (Algol)

Prenos po imenu: kao u makroima, parametri se zamenjuju sa neevaluiranim izrazima (Algol, Scala)

Prenos po konstantnoj vrednosti: isto kao kod prenosa po vrednosti, osim što se parametar tretira kao konstanta (npr kvalifikator const u C i C++)

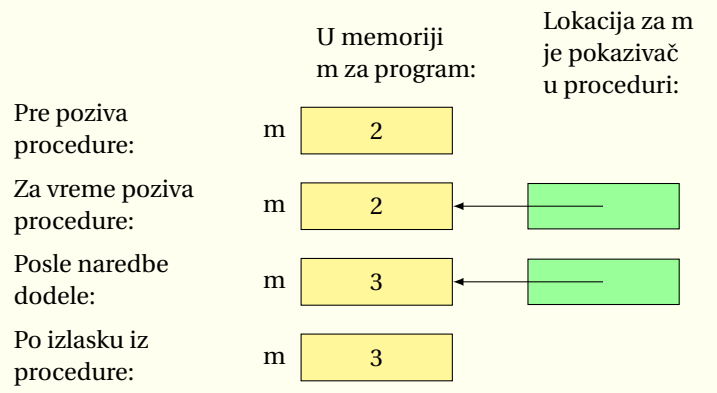
Prenos argumenata po vrednosti:

```
var m: integer                          procedure prva (m: integer);
....                                            ....
    m:=2;                                       m:= m+1;
    prva (m);                                   ....
....
```

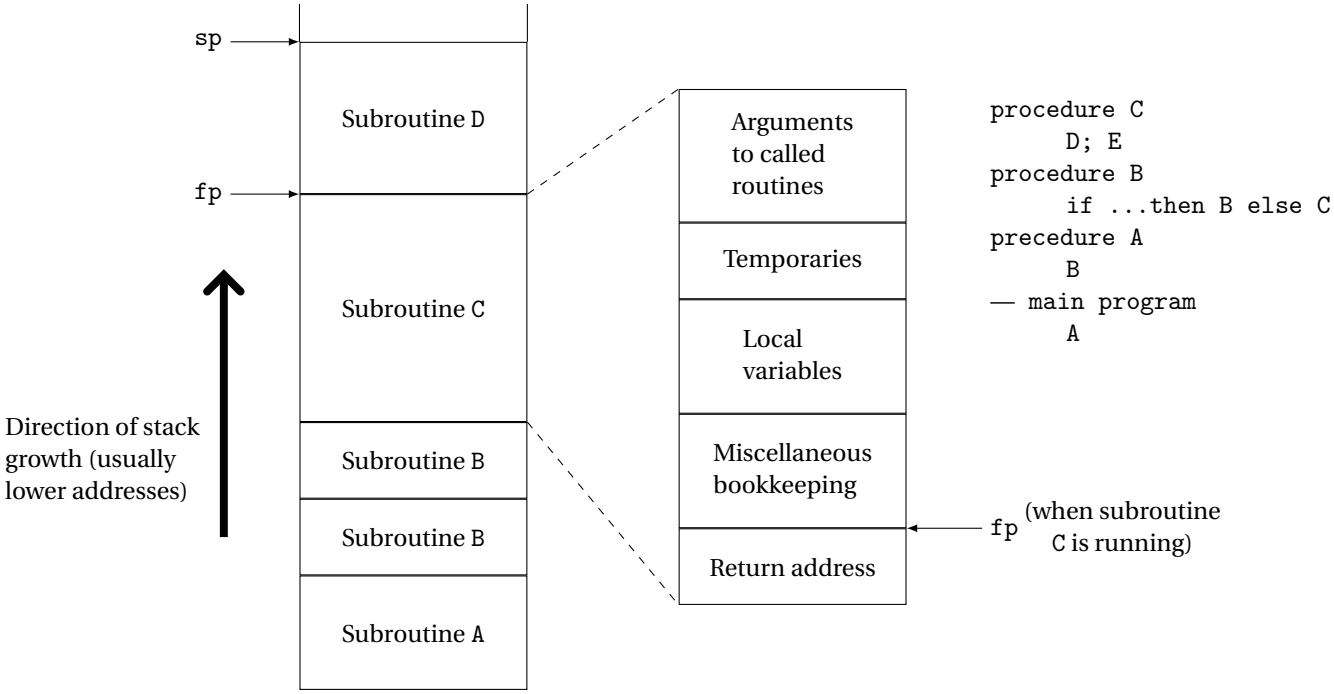


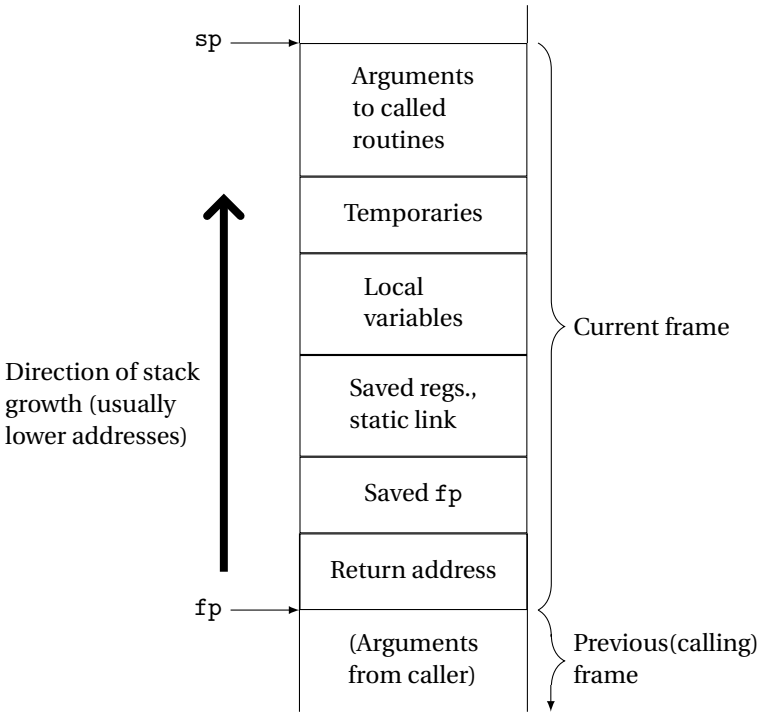
Prenos argumenata po vrednosti:

```
var m: integer                          procedure prva (var m: integer);
....                                            ....
    m:=2;                                       m:= m+1;
    prva (m);                                   ....
....
```



U većini jezika, prostor potreban za smeštanje podataka potrebnih za izvršavanje potprograma (promenljive, argumenti i slično) se rezerviše na **steku**. Uvođenje steka je imalo za ideju uštedu memorije – u memoriji su podaci samo za trenutno aktivne potprograme. Alternativa bi bila da se za svaki potprogram unapred rezerviše potrebna memorija. Ovo onemogućava korišćenje rekurzija i zauzima više prostora nego što je potrebno. S druge strane, kreiranje stek okvira povećava cenu poziva potprograma.





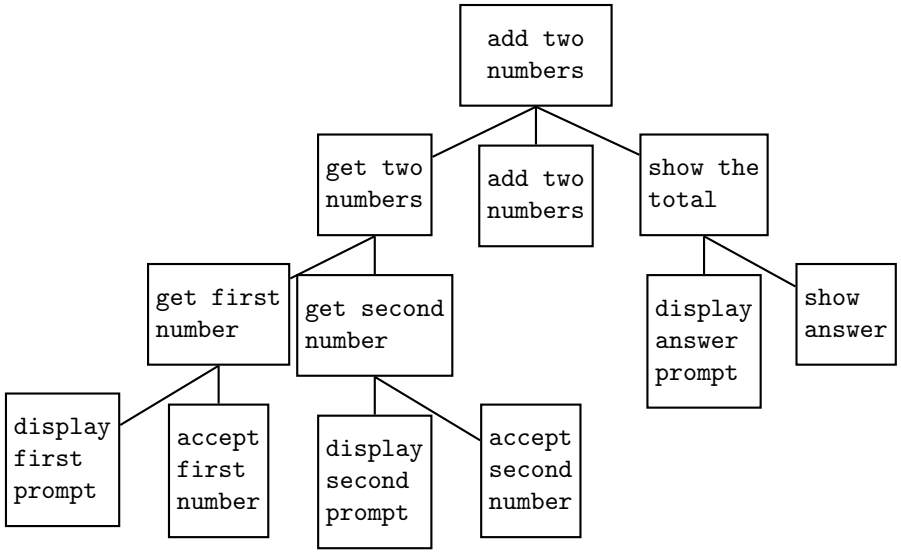
Završetak rada potprograma vraća nas u prethodni stek okvir (stek okvir pozivaoca potprograma). Pored procedura i funkcija, u imperativnim jezicima mogu se javiti i druge vrste potprograma (upravljačkih struktura), kao sto su **korutine**. One omogućavaju "preskakanje" stek okvira i povratak u određeni stek okvir koji nije nužno stek okvir pozivaoca potprograma. Upotreba može da bude za brz i efikasan izlaz iz duboke rekurzije. Slični mehanizmi koriste se u npr objektno orijentisanim programskim jezicima za rad sa izuzecima. U C-u korutine nisu podržane direktno u jeziku, već se mogu koristiti funkcije `setjmp` i `longjmp` (iz zaglavlja `setjmp.h`). Ispravna upotreba korutina zahteva precizno poznavanje njihovog mehanizma funkcionisanja.

9.4 Modularna paradigma - rane 1970.

Modularnost podrazumeva razbijanje većeg problema na nezavisne celine. Celine sadrže definicije srodnih podataka i funkcija. Često se celine (moduli) smeštaju u posebne datoteke, čime se postiže lakše održavanje kompleksnih sistema. Moduli mogu međusobno da komuniciraju, kroz svoje interfejse. Modularnost je sakrivanje podataka, razdvajanje poslova. Moduli omogućavaju višestruku upotrebu jednom napisanog koda. Oni se zasebno prevode i kasnije povezuju u jedinstven program. U imperativnim jezicima često se nazivaju **biblioteke**. Modularnost je sada prisutna u većini programskih jezika.

9.5 Način rešavanja problema

Kreiranje programa proceduralne paradigme zasniva se na principu **"Od opšteg ka posebnom"** (odozgo-nadole, od vrha ka dnu, *Top-down concept*). Polazi se od postavljenog zadatka kao opšteg. Zatim se uočavaju jednostavnije celine i zadatak dekomponuje na jednostavnije delove (koji se izražavaju procedurama i funkcijama). Ukoliko su ti delovi i dalje kompleksni, razbijaju se na još jednostavnije delove (takođe pomoću procedura i funkcija) dok se ne dodje do nivoa naredbi. Funkcionalna dekompozicija problema:



Napisati program koji izračunava prosečno rastojanje između zadatih tačaka u ravni.

- Opšte rešenje: učitati tačke iz datoteke, izračunati prosečno rastojanje, odštampati prosečno rastojanje.
- Sledeći korak: profiniti svaki od prethodnih koraka, posebno izračunavanje prosečnog rastojanja.
- Pronalaženje prosečnog rastojanja: sabrati rastojanja između svake dve tačke, podeliti rezultat sa ukupnim brojem parova tačaka.
- Obezbediti funkciju `rastojanje`
- ...

9.6 Specifičnosti imperativne paradigme

Prilikom izračunavanja vrednosti izraza, kod imperativnih programa česti su **propratni(bočni) efekti**. Propratni efekti se odnose na situacije kada se prilikom izračunavanja nekog izraza istovremeno menja i stanje memorije (na primer, vrednost izraza se upisuje u neku memorijsku lokaciju). Oznake promenljivih su istovremeno i oznake memorijskih lokacija, u zavisnosti od operatora koji se nad promenljivom primeni.
U naredbi `x = x + 1;` sa desne strane, koristi se vrednost sa lokacije `x`, dok se sa leve strane koristi sama lokacija na koju se upisuje sračunata vrednost. Propratni (bočni) efekti mogu da budu prisutni u mnogim naredbama (npr. u C-u naredbe: `++, -, +=, -=, *=`,...). Oni mogu značajno da otežaju razumevanje imperativnih programa.
Propratni efekti odnose se i na izmenu globalnog stanja memorije nakon izvršenja neke funkcije (u tom slučaju se kaže da funkcija ima propratne efekte). Ova vrsta propratnih efekata je posebno nezgodna za razumevanje rada programa.
Ukoliko funkcija *sqrt* nema propratnih efekata `z=f(sqrt(2), sqrt(2));`
onda taj kod možemo transformisati na sledeći način: `s=sqrt(2); z=f(s,s);`
Međutim, za sledeći poziv funkcije f ne možemo da ponovimo isti postupak! `z=f(getchar(), getchar());`
Ukoliko funkcija ima propratne efekte, onda to može da dovede do raznih nelogičnosti. Na primer, izrazi: `2*fun()` i `fun()+fun()` algebarski imaju iste vrednosti, ali u programskom jeziku ne moraju.

```
#include <stdio.h>

int a=3;
int fun(int);
int main() {
        int f, g;
        f= a+fun(1)+a;
        g= a+fun(1)+a;
        printf(''%d %d\n'', f, g);
}

int fun(int b){
        a+=5;
        return a+b;
}
```

```
Program p;
var z:integer;
Function f(Var x:integer):boolean;
begin
        x:= x+1;
        f := x > 0
end;
begin
        z := -1;
        writeln(f(z), '         ', f(z))
end.
```

10 Objektno-orijentisano programiranje

10.1 Funkcionalna dekompozicija problema i izmene

Kreiranje programa u objektno orijentisane programiranju zasniva se na principu **"Od opšteg ka posebnom"**. Postupak se sastoji u tome da se veliki problem podeli u manje korake koji su potrebni da bi se problem rešio. Za baš velike probleme, podela je u manje potprobleme, pa se onda dekomponuju manji problemi u odgovarajuće funkcionalne korake. Cilj je da se problem deli dok ne dođe do onog nivoa koji je jednostavan i može da se reši u nekoliko koraka, tada se ti koraci poređaju u odgovarajućem redosledu koji rešava identifikovane potprobleme i na taj način je i veliki problem rešen. Ovo je prirodan način razmišljanja, ali postoje dva osnovna problema ovog pristupa:

1. Na ovaj način se kreira program koji je dizajniran na osnovu osobina glavnog programa – ovaj program kontroliše i zna sve detalje o tome šta će biti izvršeno i koje će se strukture podataka za to koristiti.
2. Ovakav dizaj ne odgovara dobro na izmene zahteva – ovi programi nisu dobro podeljeni na celine tako da svaki zahtev za promenom obično zahteva promenu glavnog programa: mala promena u strukturama podataka, na primer, može da ima uticaja kroz ceo program.

Rešavanje ovog problema orijentisanjem na procese koji se dešavaju da bi se problem rešio ne vodi do programskih struktura koje mogu da lako reaguju na izmene: izmene u razvoju softvera obično uključuju varijacije na postojeće teme. Na primer, razmatrali smo funkcionalnu dekompoziciju problema izračunavanja prosečnog rastojanja između tačaka u ravni:

- Razmatrati prosečno rastojanje tačaka u prostoru.
- Razmatrati prosečno rastojanje između nekih drugih objekata.
- Razmatrati i druge odnose objekata, ne samo rastojanje.

U glavnom programu, ovi tipovi izmena povećavaju kompleksnost i zahtevaju puno dodatnih fajlova da se ponovo kompajliraju.
Zašto je uticaj izmena važan?

- Mnoge greške nastaju prilikom izmena koda.
- Stvari se menjaju, uvek se menjaju i ništa ne može da spreči nastajanje izmena i potrebu za prepravljanjem izmena!
- Moramo da obezbedimo da postoji dizajn koji će odoleti zahtevima za izmenama, dizajn koji je otporan (fleksibilan) na izmene.
- Treba nam dizajn koji je takav da može da se prilagodi promenala na pravi način.

10.2 Kohezija, kopčanje i efekat talasanja

Kohezija i **kopčanje** su uvedeni 70-ih godina 20. veka kao metrike koje omogućavaju kvantitativno praćenje kvaliteta koda, sa ciljem da se smanje troškovi održavanja i modifikacije koda.
Kohezija je pojam koji se odnosi na to koliko blisko su povezane operacije koje se vrše u rutini ili modulu. Pojednostavljeno, ono što želimo je da svaka rutina radi samo jednu stvar, ili da se svaki modul odnosi samo na jedan zadatak. Na

primer, nije dobro da funkcija radi računanje minimuma, maksimuma i prosečne vrednosti niza istovremeno, bolje je da to rade tri razdvojene funkcije.
Kopčanje se odnosi na jačinu povezanosti dve rutine ili modula. Na primer, nije dobro da jedan modul modifikuje i oslanja se na internu strukturu i rad drugog modula. To je jako kopčanje. Sa jakim kopčanjem, jedna promena u nekoj funkciji ili strukturi podataka uzrokuje potrebu za dodatnim promenama u svim drugim delovima sistema. Kopčanje je pojam komplementaran koheziji, slaba kohezija povlači jako kopčanje i jaka kohezija povlači slabo kopčanje. Želimo jaku kohezivnost i slabo kopčanje.

Potrebe za izmenama u okviru celog koda otežavaju debagovanje i razumevanje zadataka koje sistem obavlja. Napravimo izmenu, a neočekivano nešto drugo u sistemu ne funkcioniše, to je neželjeni propratni **efekat talasanja**. Ukoliko imamo kod sa jakim kopčanjem, otkrivamo da mnogi delovi sistema zavise od koda koji je izmenjen – treba vremena da se otkriju i razumeju ti odnosi.

10.3 Nastanak OOP

Razvoj softvera se suočavao sa velikim brojem problema. Simptomi softverske krize:

- kasne isporuke
- probijanje rokova i budžeta
- loš kvalitet
- nezadovoljivanje potreba
- slaba pouzdanost
- nepostojanje metodologije u razvoju softvera

Cena softvera se značajno povećala, održavanje i razvoj softvera je nadmašilo troškove hardvera. OOP je nastala kao jedna od posledica softverske krize. Kompleksnost softvera zahtevala je promene u stilu programiranja. Cilj je bio da se:

- proizvodi pouzdan softver
- smanji cena proizvodnje softvera
- razvijaju ponovo upotrebljivi moduli
- smanje troškovi održavanja
- smanji vreme razvoja softvera

OOP uvodi novi način razmišljanja za pronalaženje rešenja problema.

Studenti nakon predavanja idu na naredno predavanje. Funkcionalna dekompozicija problema:
Nastavnik je glavni program koji rešava problem:

- za svakog studenta nastavnik pronalazi kojoj grupi pripada
- u rasporedu časova nastavnik pronalazi koji čas data grupa treba da ima
- nastavnik svakom studentu kaže gde treba da ide i šta treba da sluša

Da li se tako rešava problem u stvarnom životu? OO pristup:

- nastavnik podrazumeva da svaki student zna raspored časova i kojoj grupi pripada, tako da sam zna gde treba da ide
- u najgorem slučaju, nastavnik može svima da saopšti gde mogu da nađu raspored časova i podelu po grupama i da kaže: koristite ove informacije da odredite lokaciju sledećeg predavanja.

Razlika između pristupa:

Prvi pristup

- Nastavnik zna sve i za sve je odgovoran, ukoliko dođe do nekih promena, na njemu je odgovornost da te promene izvede.
- Nastavnik mora da daje detaljna uputstva svakom studentu (entitetu u sistemu)

Drugi pristup

- Nastavnik očekuje da su studenti (entiteti u sistemu) sposobni da sami reše problem (da su samodovoljni).

- Nastavnik daje samo opšte instrukcije i očekuje od studenata da znaju da ih primene u svojim specifičnim situacijama.

Osnovna prednost drugog pristupa je u **podeli odgovornosti** – svaki deo sistema ima svoju odgovornost i odgovornost se prebacuje sa glavnog programa na entitete u sistemu. Pretpostavimo da u okviru grupe postoje i studenti volonteri koji između predavanja treba da urade nešto specijalno. U prvom sučaju nastavnik bi morao da zna da oni postoje i šta je to što oni treba da urade, dok u drugom nastavnik samo završava čas i kaže svima da idu na naredno predavanje, a svaki student zna šta treba, studenti volonteri sami znaju koje su njihove obaveze. Dodavanje novih vrsta obaveza i entiteta u sistemu ne remeti nastavnika (glavni program).

Prethodni primer ilustruje osnovne benefite OOP:

- Samodovoljni entiteti - objekti
- Davanje opštih instrukcija - kodiranje interfejsa
- Očekivanje da entiteti umeju da prime opšte instrukcije na njihove specifične situacije - polimorfizam i podklase
- Dodavanje novih entiteta u sistem bez uticaja na vođu sesije - kodiranje interfejsa, polimorfizam, podklase
- Prebacivanje odgovornosti - funkcionalnost je podeljena kroz mrežu objekata u sistemu

10.4 Poređenje strukturne i OO paradigme

Strukturno programiranje	Objektno-orijentisano programiranje
Primenjuje se top-down pristup	Primenjuje se bottom-up pristup
Akcenat je algoritmima i kontroli toka	Akcenat je na objektnom modelu
Program je podeljen na nekoliko potprograma, ili funkcija, ili procedura	Program je organizovan kroz nekoliko klasa i objekata
Funkcije su nezavisne jedne od drugih	Sve klase su u hijerarhijskom odnosu
Nema određenog primaoca u pozivu funkcije	Postoji određen primalac za svaku prosleđenu poruku
Podaci i funkcije se posmatraju kao dva odvojena entiteta	Podaci i funkcije su jedan jedini entitet
Skupo odrzavanje	Održavanje je relativno jeftinije
Softver se ne može koristiti za nešto drugo	Pomaže pri korišćenju softvera za druge probleme
Koriste se pozivi funkcija	Koristi se prosleđivanje poruka
Koriste se apstrakcije funkcija	Koristi se apstrakcija podataka
Prednost se daje algoritmima	Prednost se daje podacima
Rešenje je usko povezano sa domenom rešenja	Resenje je povezano sa domenom problema koji se rešava
Nema enkapsulacije. Podaci i funkcije su razdvojeni	Ekapsulacija paketa koda i podataka sveukupno. Podaci i funkcionalnosti se stavljaju u jedan entitet
Odnos između programera i programa je naglašena	Odnos između programera i korisnika je naglašena
Koristi se tehnika koja zavisi od podataka.	Podstaknuta je raspodela obaveza

Osnovne prednosti OOP u odnosu na strukturnu paradigmu je lakše održavanje, lakša ponovna upotrebljivost koda i veća skalabilnost.

10.5 Apstrakcija, interfejs, implementacija, enkapsulacija

Apstrakcija je skup osnovnih koncepata koje neki entitet obezbeđuje sa ciljem omogućavanja rešavanja nekog problema. Ona uključuje atribute koji oslikavaju osobine entiteta kao i operacije koje oslikavaju ponašanje entiteta. Daje neophodne i dovoljne opise entiteta, a ne i njihove implementacione detalje. Apstrakcija rezultuje u odvajanju interfejsa i implementacije.
Veoma je važno znati razliku između interfejsa i implementacije. **Interfejs** je korisnički pogled na to šta neki entitet može da uradi, dok **implementacija** vodi računa o internim operacijama interfejsa koji ne moraju da budu poznati korisniku. Interfejs govori ŠTA entitet može da uradi, dok implementacija govori KAKO entitet interno radi.

Intefejs	Implementacija
Korisnički pogled. (**Šta**)	Pogled dobavljača. (**Kako**)
Koristi se za interakciju sa spoljašnjim svetom.	Opisuje kako se poverena odgovornost izrvršava.
Korisniku je odobren samo pristup interfejsu.	Funkcije i metodi imaju dozvolu da pristupaju podacima. Prema tome, dobavljač je u stanju da pristupi podacima i interfejsu.
Enkapsulira znanje o objektu.	Pruža ograničenje korisniku da pristupa podacima.

Enkapsulacija (učaurivanje) je skup mehanizama koje obezbeđuje jezik (ili skup tehnika za dizajn) za skrivanje implementacionih detalja (klase, modula ili podsistema od ostalih klasa, modula i podsistema). Enkapsulacijom su podaci zaštićeni od neželjenih spoljnih uticaja. Na primer, u većini OO programskih jezika, označavanje privatne promenljive u okviru klase obezbeđuje da druge klase ne mogu da pristupe toj vrednosti direktno, već isključivo putem metoda za pristup i izmenu, ukoliko ih klasa obezbedi. Ovo se naziva **skrivanje podataka**. Enkapsulacija je širi pojam od skrivanja podataka.
Sa korisničkog stanovišta, entitet nudi izvestan broj usluga preko interfejsa i skriva implementacione detalje – termin enkapsulacija se koristi za skrivanje implementacionih detalja. Prednosti enkapsulacije su skrivanje informacija i implementaciona nezavisnost (promena implementacije može da se uradi bez promene interfejsa). Primer: klasa kompleksan broj.
Problemi adaptacije na izmene postoje kod funkcionalne dekompozicije problema jer rezultujući softver ima slabo korišćenje apstrakcija i slabu enkapsulaciju. Kada postoji slaba apstrakcija, a želimo da dodamo novu, onda nije jasno kako to treba da uradimo. Kada je loša enkapsulacija izmene imaju tendenciju da se prostiru kroz ceo kod jer ništa ne sprečava formiranje zavisnosti između delova koda.

10.6 Objekti i klase

Objekat

Filozofski: Entitet koji se može prepoznati.

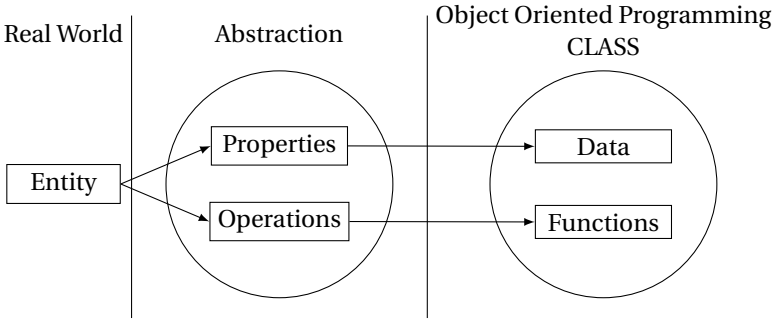
Konceptualno: Skup odgovornosti.

U termimima objektne terminologije: Apstrakcija entiteta iz stvarnog sveta.

Specifikacijski: Skup metoda.

Implementaciono: Podaci sa odgovarajućim funkcijama

Objekti imaju svoje *ponašanje* (operacije) i *osobine* (atribute). Kako se određuje koje ponašanje i koje osobine su relevantne za neki objekat? Pre svega, kako se određuju potrebni objekti? (Objekti imaju svoje odgovornosti!)
Zadatak OO analize i dizajna: u okviru analize treba da se sagleda šta je problem, koje su odgovornosti koje sistem treba da ostvari i u tome se pronalaze kandidati za potrebne objekte. Domen problema takođe sugeriše koji su objekti u sistemu potrebni.
Objekti mogu da sadrže druge objekte (**agregacija objekata**). Objekti međusobno komuniciraju slanjem poruka. Poruke su komande koje se šalju objektu sa ciljem da izvrši neku akciju. Poruke se sastoje od objekta koji prima poruku, metoda koji treba da se izvrši i argumenata (opciono). Npr: Panel1.Add(Button1) – primalac je Panel1, metod je Add, a argument je Button1.
U svetu postoji previše objekata i oni se klasifikuju u kategorije, ili u **klase**. Klase su šabloni za grupe objekata. Daju specifikaciju za sve podatke i ponašanja objekata. Za objekat kažemo da je **instanca** klase. Postoje različiti odnosi između klasa, osnovni su **nasleđivanje** i **agregacija**.



Klasa	Objekat
Klasa je tip podataka.	Objekat je instanca klasnog tipa podataka.
Ona generiše objekat.	Daje život klasi.
Ona je prototip objekta.	Predstavlja skladište za svoje karakteristike.
Ne zauzima memorijsku lokaciju.	Zauzima lokaciju u memoriji.
Ne može se manipulisati njom jer nije dostupna u memoriji.	Može se manipulisati njime.

Klase definišu kreiranje i uništavanje objekata – metodi konstruktori i dekonstruktori. Konstruktori obezbeđuju da se objekti ispravno inicijalizuju. Dekonstruktori obezbeđuju da objekat oslobodi sve resurse koje je zauzimao dok je bio aktivan.
Apstrakcija vođena podacima je osnovni koncept objektnih pristupa. U zavisnosti od prisutnosti ostalih koncepata, jezik može da bude **objektno-zasnovan** ili **objektno-orijentisan**. Objektno-zasnovan jezik podržava enkapsulaciju i identitet objekta (jedinstvene osobine koje ga razdvajaju od ostalih objekata) i nema podršku za polimorfizam, nasleđivanje, komunikaciju porukama iako takve stvari mogu da se emuliraju. Objektno-orijentisani jezici imaju sve osobine objektno-zasnovanih jezika, zajedno sa nasleđivanjem i polimorfizmom.

10.7 Nasleđivanje, polimorfizam

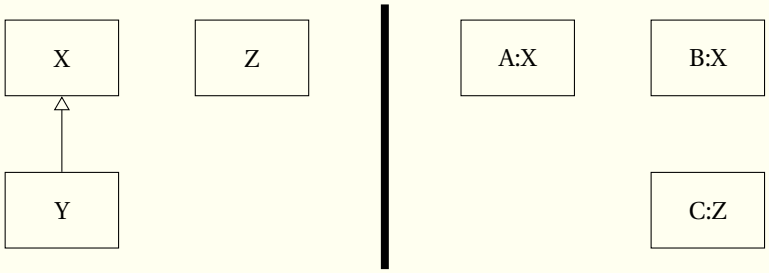
Nasleđivanje je kreiranje novih klasa (izvedenih klasa) od postojećih klasa (osnovnih ili baznih klasa). Kreiranje novih klasa omogućava postojanje hijerarhije klasa koje simuliraju koncept klasa i potklasa iz stvarnog sveta. Primer: životinja, sisar, mačka, pas, riba, štuka, som, ... Primer: vozilo, kopneno, vazdušno, vodeno, automobil, kamion, brod, čamac, avion, jedrilica, ... Nasleđivanje omogućava proširivanje i ponovno korišćenje postojećeg koda, bez ponavljanja i ponovnog pisanja koda. Bazne klase se mogu koristiti puno puta.
Nasleđivanje je korisno za **proširivanje** i **specijalizaciju**. Proširivanje koristi nasleđivanje da se razviju klase od postojećih dodavanjem novih osobina. Na primer, u okviru stambene zdrade može postojati deo za poslovni prostor, pa je klasu `StambenaZgrada` potrebno proširiti tako da može da prati i poslovni prostor. Specijalizacija koristi nasleđivanje da se preciznije definiše ponašanje opšte (apstraktne) klase. Podklase obezbeđuju specijalizovano ponašanje na osnovu zajedničkih elemenata koje obezbeđuje bazna klasa. Na primer, bazna klasa može da bude student, a podklase koje obezbeđuju specijalizovano ponašanje su klase `StudentOsnovnihStudija`, `StudentMasterStudija` i `StudentDoktorskihStudija`.
Prilikom nasleđivanja, podklasa može da doda nova ponašanja i nove podatke koji su za nju specifični, ali takođe može i da promeni ponašanje koje je nasledila od bazne klase kako bi njene specifičnosti bile uzete u obzir (polimorfizam). Poželjno je da svaka osobina koja važi za baznu klasu važi i za njenu podklasu, ali obrnuto ne mora da važi (npr, svaki pas ima sve osobine životinje, ali svaka životinja ne mora da ima sve osobine psa). Potklase se mogu tretirati kao bazne klase, jer sadrže sve atribute i metode kao i bazne klase tako da kod koji je razvijen za rad sa baznim klasama, može da se primeni i na podklase.

> Problem: imamo različite klase studenata (osnovne, master i doktorske studije) i treba da napravimo leksikografski sortiran spisak studenata.
> Rešenje: ukoliko su sve tri klase studenata podklase bazne klase Student, onda se svi studenti mogu staviti u istu kolekciju (npr u niz ili listu) i mogu se tretirati na isti način, bez obzira na njihove raznorodne specifičnosti. Ne samo što možemo sve studente da stavimo u istu kolekciju, već ih možemo tretirati na isti način, pri čemu će se svaki student ponašati u skladu sa svojim specifičnostima (polimorfizam).

Klase definišu vidljivost osobina svojim objektima: atribut ili metod može da bude javan, zaštićen ili privatan. Javna vidljivost omogućava svima da pristupe atributu ili metodu. Zaštićena vidljivost omogućava pristup samo podklasama date klase, dok privatna vidljivost omogućava pristup samo u okviru same klase. U jeziku JAVA zaštićena vidljivost je moguća samo između klasa koje nisu u istom paketu.

Ovo su, naravno, opšte definicije, različiti jezici implementiraju vidljivost na različite načine.
Klasa može da se izvede kroz nasleđivanje iz više od jedne osnovne klase – **višestruko nasleđivanje**. Instance klase koja koristi višestruko nasleđivanje imaju osobi koje nasleđuju od svake bazne klase. Višestruko nasleđivanje ima očigledne prednosti, ali ima i mane (kompleksnost, uvođenje virtuelnog nasleđivanja, teže debagovanje). Podržano je u C++-u. Java ne podržava višestruko nasleđivanje, ali podržava implementiranje interfejsa, što do neke mere liči na višestruko nasleđivanje.
Polimorfizam – puno formi – različita ponašanja. U okviru objektne terminologije, najčešće se odnosi na tretiranje objekata kao da su objekti bazne klase ali od njih dobijati ponašanje koje odgovara njihovim specifičnim potklasama. U ovom slučaju, polimorfizam se odnosi na kasno vezivanje poziva za jednu od više različitih implementacija metoda u hijerarhiji nasleđivanja.
Treba razlikovati pojmove **preopterećivanja** i **predefinisanja**.
Preopterećivanje (eng. *overloading*) – isto ime ili operator može da bude pridružen različitim operacijama, u zavisnosti od tipa podataka koji mu se proslede. Na primer `int f(){ ldots }` i `int f(int x) { ...}`.
Predefinisanje (eng. *overriding*) – mogućnost različitih objekata da odgovore na iste poruke na različiti način. Ukoliko u

Objekat A je instance klase X.
Objekat B je instanca klase Y koja je potklasa klase X.
Objekat C je instanca klase Z koja je nezavisna od klasa X i Y.



Javna vidljivost karakteristika klase X znači da A, B i C mogu da im pristupe.
Zaštićena vidljivost karakteristika klase X znači da A i B mogu da im pristupe, ali da C ne može.
Privatna vidljivost karakteristika klase X znači da samo A može da im pristupi.

baznoj i izvedenoj klasi imamo metod koji ima isti potpis, onda kažemo da je izvedena klasa zapravo predefinisala metod iz bazne klase.
Statičko vezivanje je "odlučivanje " koja će metoda biti pozvana u vreme prevođenja. Pretpostavimo da postoje dve metode sa istim imenom u istoj klasi koje se razlikuju po broju i/ili tipu argumenata – odluka o tome koja će od ove dve metode biti pozvana može se doneti u fazi prevođenja i to tako što se izvrši poređenje tipova argumenata, a ukoliko postoji dvosmislenost onda prevodilac javlja grešku. Ako ne postoji metod sa datim imenom u izvedenoj klasi onda se on traži u baznoj klasi. Preopterećivanje se odlučuje statički.
Dinamičko vezivanje je "odlučivanje" koja će metoda biti pozvana u vreme izvršavanja programa.

Ako imamo baznu klasu `Zivotinja` i ako iz nje izvedemo klasu `Macka`, klasu `Pas` i klasu `Konj`, tada na primer možemo formirati niz pokazivača na klasu `Zivotinja` kojima u zavisnosti od situacije možemo dodeliti da pokazuju na različite `Macke`, `Pse`, `Konje`. Ako su izvedene klase predefinisale neku metodu `f` klase `Zivotinja`, želimo da pozivom te metode uz pomoć pokazivača na `Zivotinju` bude pozvana odgovarajuća predefinisana metoda i to iz klase `Macka` ukoliko pokazivač pokazuje na `Macku`, iz klase `Pas` ukoliko pokazivač pokazuje na `Psa` i analogno za klasu `Konj`.

Da bi se koristilo dinamičko vezivanje, potrebno je koristiti odgovarajuće mehanizme jezika (virtuelne metode). Cena dinamičkog vezivanja je kreiranje tabela **virtuelnih funkcija**. Za svaki objekat kompajler implicitno dodaje pokazivač na tabelu virtuelnih funkcija. Na osnovu te tabele određuje se u fazi izvršavanja koja će tačno metoda biti pozvana.

Apstraktne klase definišu ponašanja koja su relevantna za sve podklase. One definisu potpise metoda koje podklase treba da implementiraju, definišu metode ponašanja koja su zajednička i definišu podatke koji su zajednički i korisni za sve podklase. One ne mogu da se instanciraju – instanciraju se konkretne klase, dok se pristupa preko interfejsa definisanog od strane apstraktne klase.

Šabloni omogućavaju prevazilaženje ograničenja strogo tipiziranih jezika (*templates* – C++). Stroga tipiziranost ima kao posledicu da se i jednostavne funkcije moraju definisati više puta da bi se mogle upotrebljavati na raznim tipovima. Na primer, funkcija koja računa minimum dva prirodna broja, dva realna, dva razlomka, dva kompleksna, ... – za ovo se koriste šabloni funkcija. Mehanizam šablona omogućava i automatsko generisanje klasnih tipova. Primer: niz, lista, skup, sortiranje, ... Definicija šablona klase određuje kako se konstruišu pojedine klase kada je dat skup od jednog ili više stvarnih tipova ili vrednosti. **Generičko programiranje** omogućava višestruko korišćenje softvera.
Implicitni polimorfizam – uopštenje parametarskog, prisutno na primer u funkcionalnim programskim jezicima.

10.8 OO jezici i njihove osobine

Većina OO jezika je veoma slično imperativnoj paradigmi, ali je promenjen način razmišljanja o problemu. OOP pomera fokus sa algoritama na podatke. Zasniva se na modelovanju objekata iz stvarnog sveta. Osnovna jedinica **apstrakcije** je **objekat** koji **enkapsulira** podatke i ponašanje.

Vezivanje se odnosi na određivanje tipova promenljivih – Late je dinamičko (u fazi izvršavanja), Early je statičko (u fazi prevođenja).

Jezik	Pronalazač i godina	Organizacija
Simula	Kristen Nygaard, Ole-Johan Dahl, 1960.	Norwegian Defens Research Establishment, Norway
Ada	Jean Ichbiah, 1970.	Honeywell-CII-Bull, France
Smalltalk	Alan Kay, 1970.	Xerox PARC, USA
C++	Bjarne Stroustrup, 1980.	AT&T Bell Labs, USA
Objective C	Brad Cox, 1980.	Stepstone, USA
Object Pascal	Larry Tesler, 1985.	Apple Computer, USA
Eiffel	Bertrand Meyer, 1992.	Eiffel Software, USA
Java	James Gosling, 1996.	Sun Microsystems, USA
C#	Andres Hejlsberg, 2000.	Microsoft, USA

Slika 4: OO jezici i njihove osobine

Feature	Java	C++	Smalltalk	Objective C	Simula	Ada	Eiffel	C#
Encapsulation	✓	✓	Poor	✓	✓	✓	✓	✓
Single inheritance	✓	✓	✓	✓	✓	✗	✓	✓
Multiple inheritance	✗	✓	✗	✓	✗	✗	✓	✗
Polymorphism	✓	✓	✓	✓	✓	✓	✓	✓
Binding (Late, Early)	Late	both	Late	both	both	Early	Early	Late
Concurrency	✓	Poor	Poor	Poor	✓	Difficult	✓	✓
Garbage collection	✓	✗	✓	✓	✓	✗	✓	✓
Persistent objects	✗	✗	promised	✗	✗	like 3GL	limited	✗
Genericity	✓	✓	✗	✗	✗	✓	✓	✓
Class libraries	✓	✓	✓	✓	✓	limited	✓	✓

Slika 5: OO jezici i njihove osobine

11 Osnovna svojstva programskih jezika

11.1 Uvod

- Šta je ono što dva programska jezika čini sličnim?
- Na osnovu čega neki jezik pripada nekoj paradigmi?
- Šta je ono što dva programska jezika čini različitim?
- Na osnovu čega neki jezik ne pripada nekoj paradigmi?
- Šta je zajedničko za sve programske jezike?
- Šta je ono što je specifično za svaki programski jezik?
- Šta je ono što je specifično za neki programski jezik?

Postoje različita svojstva programskih jezika koja ćemo izučiti: sintaksa, semantika, imena, doseg, povezanost, kontrola toka, podrutine, tipovi, kompajlirani/interpretirani jezici, izvršavanje.

11.2 Leksika, sintaksa, semantika

Programski jezici moraju da budu precizni. Leksika je podoblast sintakse koja se bavi opisivanjem osnovnih gradivnih elemenata jezika. U okviru leksike, definišu se reči i njihove kategorije. U programskom jeziku, reči se nazivaju *lekseme*, a kategorije *tokeni*. U prirodnom jeziku, kategorije su imenice, glagoli, pridevi,... U programskom jeziku, tokeni mogu da budu identifikatori, ključne reči, operatori, ...

`a=2*b+1` – Lekseme su `a`, `=`, `2`, `*`, `+`, `1` i `b`, a njima odgovarajući tokeni su identifikator (`a`, `b`), operator (`=`, `*`, `+`), celobrojni literali (`1`, `2`).

Neki tokeni sadrže samo jednu reč, a neki mogu sadržati puno različitih reči. Programski jezik C sadrži više od 100 različitih tokena: 44 različite ključne reči, identifikatore, celobrojne konstante, realne vrednosti, karakterske konstante, stringovske literale, dve vrste komentara, 54 operatora, ... Drugi moderni programski jezici (npr Ada, Java) imaju sličan nivo kompleksnosti tokena.

Razlikujemo ključne reči i identifikatore: identifikator ne može biti neka od ključnih reči, npr ne možemo da napravimo promenljivu koja bi se zvala if ili while. Postoje ključne reči koje zavise od konteksta, eng. *contextual keywords* koje su ključne reči na određenim specifičnim mestima programa, ali mogu biti identifikatori na drugim mestima. Na primer, u C#-u reč `yield` može da se pojavi ispred break ili return, na mestima na kojim identifikator ne može da se pojavi. Na tim mestima, ona se interpretira kao ključna reč, ali može da se koristi na drugim mestima i kao identifikator. C# 4.0 ima 26 takvih konteksno zavisnih ključnih reči, C++ 11 ih takođe ima dosta. Većina je uvedena revizijom postojećeg jezika sa ciljem da se definiše novi standard: sa velikim brojem korisnika i napisanog koda, vrlo je verovatno da se neka reč već koristi kao identifikator u nekom postojećem programu. Uvođenjem kontekstualne ključne reči, umesto obične ključne reči, smanjuje se rizik da se postojeći program neće kompilirati sa novom verzijom standarda. Reči se obično definišu regularnim izrazima. Na primer, `[a-zA-Z_][a-zA-Z_0-9]*`. Regularnim izrazima se definišu reči, dok se konačnim automatima prepoznavaju ispravne reči. Generisanje je bitno programerima, a prepoznavanje kompajlerima. Leksikom programa obično se bavi deo prevodioca koji se naziva leksički analizator: on dodeljuje ulaznim rečima odgovarajuće kategorije, što je bitno za dalji proces prevođenja.

Sintaksa programskog jezika definiše strukturu izraza, odnosno načine kombinovanja osnovnih elemenata jezika u ispravne jezičke konstrukcije. Formalno, sintakse se opisuju kontekstno slobodnim gramatikama, a prepoznavaju parserima (potisni automat).

Sledeća jednostavna gramatika definiše jednostavne aritmetičke izraze:

```
<exp> ::= <exp>''+'' <exp>
<exp> ::= <exp> ''*'' <exp>
<exp> ::= ''('' <exp> ''<exp>''
<exp> ::= ''a''
<exp> ::= ''b''
<exp> ::= ''c''
```

Program u ovom jeziku je proizvod ili suma od 'a', 'b', 'c', na primer ispravan izraz je `a*(b+c)`.

Neformalno govoreći, semantika pridružuje značenje ispravnim konstrukcijama na nekom programskom jeziku. Semantika programskog jezika određuje značenje jezika. Obično je značajno teže definisati nego sintaksu. Uloga semantike je da programer može da razume kako se program izvršava pre njegovog pokretanja. Semantika može da se opiše formalno i neformalno, često se zadaje samo neformalno. Na primer, semantika naredbe `if(a<b) a++;` neformalno se opisuje sa "ukoliko je vrednost promenljive `a` manja od vrednosti promenljive `b`, onda uvećaj vrednost promenljive `a` za jedan".

Formalna semantika omogućava formalno rezonovanje o svojstvima programa. Na primer, formalna semantika nekog jezika, zajedno sa modelom programa (tranzicionim sistemom) omogućava ispitivanje različitih uslova ispravnosti/korektnosti tog programa. Različitim programskim jezicima prirodno odgovaraju različite semantike. Formalni opis značenja jezičkih konstrukcija:

- Operaciona semantika
- Aksiomska semantika
- Denotaciona semantika

Operaciona semantika

Opisuje kako se izračunavanje izvršava. Ponašanje se formalno definiše korišćenjem apstraktnih mašina, formalnih automata, tranzicionih sistema... U okviru ove semantike postoje strukturna operaciona semantika (*small-step semantics*, opisuje individualne korake izračunavanja) i prirodna operaciona semantika (*big-step semantics*, opisuje ukupne rezultate izračunavanja). Najčešće se koristi za opis i rezonovanje o imperativnim jezicima jer individualni koraci izračunavanja opisuju na koji način se menja stanje programa.

Denotaciona semantika

Definiše značenje prevođenjem u drugi jezik, za koji se pretpostavlja da je poznata semantika. Najčešće je taj drugi jezik nekakav matematički formalizam. Povezivanje svakog dela programskog jezika sa nekim matematičkim objektom kao što je to broj ili funkcija: svaka sintaksna definicija se tretira kao objekat na koji se može primeniti funkcija koja taj objekat preslikava u matematički objekat koji definiše značenje. Dodeljivanjem značenja delovima programa dodeljuje se značenje celokupnom programu, tj. semantika jedne programske celine definisana je preko semantike njenih poddelova. Ova osobina denotacione semantike naziva se **kompozitivnost**.

Primer operacione sematike - Prirodna semantika:

$$\langle S, s\rangle \to s'$$

Intuitivno ovo znači da će se izvršavanje programa S sa ulaznim stanjem s završiti i rezultujuće stanje će biti s'.
Pravilo generalno ima formu

$$\frac{\langle S_1,\ s_1\rangle \to s_1', \ldots, \langle S_n,\ s_n\rangle \to s_n'}{\langle S, s\rangle \to s'}$$

gde $S_1, \ldots, S_n$ nazivamo neposrednim konstituentima (eng. *immediate constituents*) od S. Pravilo se sastoji iz određenog broja *premisa* (nalaze se iznad linije) i jednog *zaključka* (nalazi se ispod linije). Pravilo takođe može imati određen broj *uslova* (nalaze se sa desne strane linije) koji moraju biti ispunjeni kako bi se pravilo primenilo. Pravilo sa praznim skupom premisa naziva se *aksioma*.
Primer prirodne semantike:
$\mathscr{B}-$semantika bulovskog izraza, $\mathscr{A}-$semantika aritmetičkog izraza

$$(z := x, x := y); y := z$$

Neka s_0 bude stanje koje mapira sve promenljive osim x i y u 0 i ima $x = 5$ i $y = 7$. Tada dobijamo sledeće stablo izvođenja:

$$\frac{\dfrac{\langle z{:=}x,\ s_0\rangle \to s_1 \qquad \langle x{:=}y,\ s_1\rangle \to s_2}{\langle z{:=}x,\ x{:=}y,\ s_0\rangle \to s_2} \qquad \langle y{:=}z,\ s_2\rangle \to s_3}{\langle (z := x;\ x := y);\ y := z,\ s_0\rangle \to s_3}$$

$$[ass_{ns}] \qquad \langle x := a,\ s\rangle \to s[s \mapsto \mathscr{A}[[a]]s]$$

$$[skip_{ns}] \qquad \langle skip,\ s\rangle \to s$$

$$[comp_{ns}] \qquad \frac{\langle S_1,\ s\rangle \to s', \langle S_2,\ s'\rangle \to s''}{\langle S_1;\ S_2,\ s'\rangle \to s''}$$

$$[if_{ns}^{tt}] \qquad \frac{\langle S_1,\ s\rangle \to s'}{\langle if\ \ b\ \ then\ S_1\ \ else\ S_2,\ \ s\rangle \to s'}\ if \mathscr{B}[[b]]s = tt$$

$$[if_{ns}^{ff}] \qquad \frac{\langle S_2,\ s\rangle \to s'}{\langle if\ \ b\ \ then\ S_1\ \ else\ S_2, s\rangle \to s'}\ if\ \mathscr{B}[[b]]s = ff$$

$$[while_{ns}^{tt}] \qquad \frac{\langle S,\ s\rangle \to s', \langle while\ \ b\ \ do\ S,\ s'\rangle \to s''}{\langle while\ \ b\ \ do\ S,\ \ s\rangle \to s''}\ if \mathscr{B}[[b]]s = tt$$

$$[while_{ns}^{ff}] \qquad \langle while\ \ b\ \ do\ \ S, s\rangle \to s\ \ if\ \mathscr{B}[[b]]s = ff$$

Osobine semantike
Kada imamo definisanu semantiku, to nam omogućava da rezonujemo o osobinama te semantike. Na primer, može nas interesovati da li su dve naredbe semantički ekvivalentne, odnosno da li važi da za svaka dva stanja s i s' važi sledeće:

$$\langle S_1,\ s\rangle \to s' akko \langle S_2,\ s\rangle \to s'$$

Možemo da formulišemo sledeću lemu: Naredba `while b do S` je semantički ekvivalentna sa `if b then (S; while b do S) else skip`.
Potrebno je dokazati u dva smera, tj.

$$\langle while\ \ b\ \ do\ \ S,\ s\rangle \to s''\ \ akko\ \langle if\ \ b\ \ then\ (S;\ while\ \ b\ \ do\ \ S)\ else\ \ skip,\ s\rangle \to s''$$

Dokaz se izvodi konstruisanjem stabla izvođenja na osnovu pravila izvođenja koja su data semantikom. Na primer, ako pretpostavimo da važi

$$\langle while\ \ b\ \ do\ \ S,\ s\rangle \to s''$$

onda postoji stablo izvođenja kojima se dolazi do stanja s''. Ukoliko pogledamo pravila izvođenja, do takvog stabla možemo doći samo primenom pravila $[while_{ns}^{tt}]$ ili pravila $[while_{ns}^{ff}]$. Potrebno je razmotriti jedan i drugi slučaj.
(prvi slučaj): ukoliko se primenilo pravilo $[while_{ns}^{tt}]$ to se onda stablo izvođenja svodi na primenu ovog pravila:

$$\frac{\langle S,\ s\rangle \to s', \langle while\ \ b\ \ do\ \ S,\ s'\rangle \to s''}{\langle while\ \ b\ \ do\ \ S,\ \ s\rangle \to s''}$$

pri čemu važi $\mathscr{B}[[b]]s = tt$ odnosno stablo izvođenja izgleda ovako:

$$\frac{T_1 \quad T_2}{\langle while\ \ b\ \ do\ \ S,\ s\rangle \to s''}$$

pri čemu je T_1 nekakvo izvođenje za $\langle S, s\rangle \to s'$, a T_2 nekakvo izvođenje za $\langle while\ b\ do\ S, s'\rangle \to s''$. Koristeći T_1 i T_2 možemo da primenimo i pravilo kompozicije $[skip_{ns}]$ i da izvedemo sledeći zaključak

$$\frac{T_1 \quad T_2}{\langle S;\ while\ b\ do\ S,\ s\rangle \to s''}$$

Pošto znamo da je $\mathscr{B}[[b]]s = tt$, možemo da primenimo $[if^{tt}_{ns}]$ i da nam stablo izvođenja sada izgleda ovako, čime smo pokazali da željeno svojstvo važi:

$$\frac{\dfrac{T_1 \quad T_2}{\langle S;\ while\ b\ do\ S,\ s\rangle \to s''}}{\langle if\ b\ then\ (S;\ while\ b\ do\ S)\ else\ skip,\ s\rangle \to s''}$$

Dalje je potrebno razmotriti $[while^{ff}_{ns}]$.

Strukturna operaciona semantika
Tranzicionu relaciju zapisujemo ovako:

$$\langle S,\ s\rangle \Longrightarrow \gamma$$

ovo treba razmatrati kao *prvi korak* izvršavanja programa S u stanju s koji vodi do stanja γ:

1. $\gamma =< S', s >$: izvršavanje programa S sa ulaznim stanjem s *nije završeno,* i ostatak izračunavanja će biti izraženo srednjom konfiguracijom $< S', s' >$.
2. $\gamma = s'$: izračunavanje programa S sa ulaznim stanjem s završilo se sa završnim stanjem s'.

$[ass_{sos}]$	$\langle x := a,\ s\rangle \Rightarrow s[s \mapsto \mathscr{A}[[a]]s]$
$[skip_{sos}]$	$\langle skip,\ s\rangle \Rightarrow s$
$[comp^1_{sos}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S'_1,\ s'\rangle}{\langle S_1;\ S_2,\ s\rangle \Rightarrow \langle S'_1;\ S_2,\ s'\rangle}$
$[comp^2_{sos}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$
$[if^{tt}_{sos}]$	$\langle if\ b\ then\ S_1\ else\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle\ if\ \mathscr{B}[[b]]s = tt$
$[if^{ff}_{sos}]$	$\langle if\ b\ then\ S_1\ else\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle\ if\ \mathscr{B}[[b]]s = ff$
$[while_{sos}]$	$\langle while\ b\ do\ S, s\rangle \Rightarrow \langle\ if\ b\ then\ (S;\ while\ b\ do\ S)\ else\ skip,\ s\rangle$

SOS omogućava praćenje "sitnijih detalja" izračunavanja.

Aksiomatska semantika - primer

$[ass_p]$	$\{P[x \to [[A]]]\}x := a\ \{P\}$
$[skip_p]$	$\{P\}skip\{P\}$
$[comp_p]$	$\dfrac{\{P\}\ S_1\ \{Q\},\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ S_2\{R\}}$
$[if_p]$	$\dfrac{\{B[[b]] \wedge P\}\ S_1\ \{Q\},\ \{\neg B[[b]] \wedge P\}\ S_1\ \{Q\}}{\{P\}\ if\ b\ then\ S_1\ else\ S_2\ \{Q\}}$
$[while_p]$	$\dfrac{\{B[[b]] \wedge P\}\ S\ \{P\}}{\{P\}\ while\ b\ do\ S\ \{\neg B[[b]] \wedge P\}}$
$[cons_p p]$	$\dfrac{\{P'\}\ S\ \{Q'\}}{\{P\}\ S\ \{Q\}}\ if\ P \Rightarrow P'\ and\ Q \Rightarrow Q'$

Sintaksni domeni i pravila:
$B : Broj$ B je nenegativan broj
$C : Cifra$ C je cifra 0,1,...,9
$I : Izraz$
$Broj ::== Cifra | BrojCifra$
$Cifra ::== 0|1|2|3|4|5|6|7|8|9$
$Izraz ::== Broj | Izraz + Izraz$

Sledeći korak jeste definisanje matematičkih objekata koji će predstavljati semantičke vrednosti. Ti matematički objekti nazivaju se **semantički domeni**. Njihova kompleksnost zavisi od toga koliko je kompleksan programski jzik kojem dajemo značenje, u ovom jednostavnom primeru, semantička vrednost može biti i samo prirodan broj ($N = 0, 1, 2, 3, \ldots$).

Funkcije značenja daju značenje uvedenim sintaksnim definicijama. (videti primer)
Denotaciona sematika apstrahuje izvršavanje funkcionalnih programskih jezika – funkcionalno programiranje zasniva se na pojmu matematičkih funkcija i izvršavanje programa svodi se na evaluaciju funkcija. Analiziranje programa se svodi na analiziranje matematičkih objekata, što olakšava formalno dokazivanje semantičkih svojstava programa.

Funkcije značenja:
$povezibn : B \to N$ unarna funkcija - povezuje broj sa N
$povezicn : C \to N$ unarna funkcija - povezuje cifru sa N
$semantika : I \to N$ unarna funkcija - povezuje izraz sa N
$plus : N \times N \to N$ binarna funkcija plus - isto što i +
$pom : N \times N \to N$ binarna funkcija pom - isto što i *
$povezicn[[0]] = 0, ..., povezicn[[9]] = 9$
$povezibn[[C]] = povezicn[[C]]$
$povezibn[[BC]] = plus(pom(10, povezibn[[B]]), povezibn[[C]])$
$semantika[[B]] = povezibn[[B]]$
$semantika[[I1 + I2]] = plus(semantika[[I1]], semantika[[I2]])$

Zagrade $[[,]]$ imaju ulogu da razdvoje semantički deo od sintaksnog dela. U okviru zagrada nalazi se sintaksni deo definicija.
Pronaći značenje izraza $2 + 32 + 61$.

$$\begin{aligned} semantika[[2+32+61]] &= plus(semantika[[2+32]], semantika[[61]]) \\ &= plus(plus(semantika[[2]], semantika[[32]]), povezibn[[61]]) \\ &= plus(plus(2, 32), 61) = plus(2+32, 61) \\ &= plus(34+61) = 34+61 = 95 \end{aligned}$$

jer je:

$$\begin{aligned} semantika[[2]] &= povezibn[[2]] = povezicn[[2]] = 2 \\ semantika[[32]] &= povezibn[[32]] \\ &= plus(pom(10, povezibn[[3]]), povezibn[[2]]) \\ &= plus(pom(10, povezicn[[3]]), povezicn[[2]]) \\ &= plus(pom(10,3), 2) = plus(10*3, 2) \\ &= plus(30, 2) = 30+2 = 32 \\ semantika[[61]] &= povezibn[[61]] \\ &= plus(pom(10, povezibn[[6]]), povezibn[[1]]) \\ &= plus(pom(10, povezicn[[6]]), povezicn[[1]]) \\ &= plus(pom(10,6), 1) = plus(10*6, 1) \\ &= plus(6center0, 1) = 60+1 = 61 \end{aligned}$$

Aksiomatska semantika

Aksiomska semantika zasniva se na matematičkoj logici, na primer na Horovoj logici. Horova trojka `{P} C {Q};` opisuje kako izvršavanje dela koda menja stanje izračunavanja: ako je ispunjen preduslov `{P}`, izvršavanje komande `C` vodi do postuslova `{Q}`. Horova logika obezbeđuje aksiome i pravila izvođenja za sve konstrukte jednostavnog imperativnog programskog jezika.

```
                          {P}S{Q}, {Q}T{R}
--------------            -----------------
  {P}skip{P}                 {P}S;T{R}
```

Semantika

Kompajler prevodi kod na mašinski jezik u skladu sa zadatom semantikom jezika. Tokom kompilacije, vrši se proveravanje da li postoji neka semantička greška, tj. situacija koja je sintaksno ispravna, ali za kontretne vrednosti nema pridruženo značenje zadatom semantikom. Neki aspekti semantičke korektnosti programa se mogu proveriti tokom prevođenja programa – na primer, da su sve promenljive koje se koriste u izrazima definisane i da su odgovarajućeg tipa. Neki aspekti semantičke korektnosti se mogu proveriti tek u fazi izvršavanja programa – na primer, deljenje nulom, pristup elementima niza van granica,...

Semantičke provere se mogu podeliti na statičke (provere prilikom kompilacije) i dinamičke (provere prilikom izvršavanja programa). Statički se ne može pouzdano utvrditi ispunjenost semantičkih uslova, tako da je moguće da se neke greške ne utvrde, iako prisutne, kao i da se neke greške pogrešno utvrde iako nisu prisutne i da rezultiraju nepotrebnim proverama prilikom izvršavanja programa. Različitim programskim jezicima odgovaraju izvršni programi koji sadrže različite nivoe dinamičkih provera ispravnosti.

11.3 Imena, povezanost, doseg

Ime – string koji se koristi za predstavljanje nečega (promenljive, konstante, operatora, tipova, ...).

Prvi programski jezici imali su imena dužine jednog karaktera (po uzoru na matematičke promenljive). Fortran I je to promenio dozvoljavajući 6 karaktera u imenu. Fortran 95 i kasnije verzije Fortrana dozvoljavaju 31 karakter u imenu, C99 nema ograničenja za interna imena, ali samo prvih 63 je značajno. Eksterna imena u C99 (ona koja su definisana van mogula i o kojima mora da brine linker) imaju ograničenje od 31 karaktera. Imena u Javi, C# i Adi nemaju ograničenja dužine i svi karakteri su značajni. C++ ne zadaje limit dužine imena, ali implementacije obično zadaju.

Imena u većini programskih jezika imaju istu formu – slova, cifre i podvlaka. Podvlaka se sve češće zamenjuje kamiljom notacijom. U nekim jezicima moraju da počnu specijalnim znacima (npr PHP ime mora da počne sa $). Imena su najčešće case senstive, što može da stvara probleme.

Specijalne reši se koriste da učine program čitljivijim – imenuju akcije koje treba da se sprovedu ili sintaksno odvajaju delovi naredbi i programa. U većini jezika specijalne reči su rezervisane reči koje ne mogu da budu predefinisane od strane programa. Ključne reči su najčešće rezervisane reči, ali ne moraju to da budu, kao na primer kontekstno zavisne ključne reči. Nije dobro ako jezik ima prevelik broj rezervisanih reči. Na primer, COBOL ima oko 300 rezervisanih reči, u koje spadaju i `count`, `length`, `bottom`, `destination`, ...

Promenljiva – apstrakcija memorijskih jedinica (ćelija), ime promenljive je imenovanje memorijske lokacije.

Karakteristike promenljivih: ime, adresa, vrednost, tip, životni vek, doseg.

Imena promenljivih su najčešća imena u programu, ali nemaju sve promenljive imena (tj nemaju sve memorijske lokacije imena, na primer, memorijske lokacije eksplicitno definisane na hipu kojima se pristupa preko pokazivača).

Adresa promenljive je adresa fizičke memorije koja je pridružena promenljivoj za skladištenje podataka. Promenljiva može imati različite fizičke lokacije prilikom izvršavanja programa (npr različiti pozivi iste funkcije rezultuju različitim adresama na steku). Adresa se često naziva *l-vrednost*. Moguće je da postoje više promenljivih koje imaju istu adresu – **aliasi**. Aliasi pogoršavaju čitljivost programa i čine verifikaciju programa težom.

Aliasi: unije u C/C++-u, dva pointera koji pokazuju na istu memorijsku lokaciju, dve reference, pointer i promenljiva,... Aliasi se u mnogim jezicima kreiraju kroz parametre poziva potprograma.

Tip promenljive određuje opseg vrednosti koje promenljiva može da ima kao i operacije koje se mogu izvršiti za vrednosti tog tipa. Osnovni tipovi, niske, korisnički definisani prebrojivi tipovi (enum, su- brange), nizovi, asocijativni nizovi, strukture, torke, liste, unije, pokazivači i reference.

Vrednost promenljive je sadržaj odgovarajuće memorijske ćelije, često se naziva *r-vrednost*. Da bi se pristupilo r-vrednosti mora najpre da se odredi l-vrednost, što ne mora da bude jednostavno, jer zavisi od pravila dosega.

Povezivanje uspostavlja odnos izmedu imena sa onim što to ime predstavlja. Vreme povezivanja je vreme kada se ova veza uspostavlja. Rano vreme povezivanja – veća efikasnost, kasno vreme povezivanja – veća fleksibilnost. Vreme povezivanja može biti (vreme - primer):

- vreme dizajna programskog jezika (osnovni konstrukti, primitivni tipovi podataka...)
- vreme implementacije (veličina osnovnih tipova podataka...)
- vreme programiranja (algoritmi, strukture, imena promenljivih)
- vreme kompiliranja (preslikavanje konstrukata višeg nivoa u mašinski kod)
- vreme povezivanja (kada se ime iz jednog modula odnosi na objekat definisan u drugom modulu)
- vreme učitavanja (kod starijih operativnih sistema, povezivanje objekata sa fizičkim adresama u memoriji)
- vreme izvršavanja (povezivanje promenljivih i njihovih vrednosti)

Doseg određuje deo programa u kojem je vidljivo neko ime. Pravila dosega mogu da budu veoma jednsotavna (kao u skript jezicima) ili veoma kompleksna. Doseg ne određuje nužno životni vek objekta.

Životni vek obično odgovara jednom od tri osnovna mehanizma skladištenja podataka. Razlikuju se:

Statički objekti koji imaju životni vek koji se prostire tokom celog rada programa (npr globalne promenljive)

Objekti na steku živoni vek koji se alocira i dealocira LIFO principom, obično zajedno sa pozivom i završetkom rada podprograma.

Objekti na hipu Objekti na hipu — koji se aociraju i dealociraju proizvoljno od strane programera, implicitno ili eksplicitno, i koji zahtevaju opštiji i skuplji sistem za upravljanje skladištenjem (memorijom)

11.4 Kontrola toka i tipovi

Kontrola toka definiše redosled izračunavanja koje računar sprovodi da bi se ostvario neki cilj. Postoje različiti mehanizmi odredivanja kontrole toka:

1. **Sekvenca** – određen redosled izvršavanja
2. **Selekcija** – u zavisnosti od uslova, pravi se izbor (if, switch)
3. **Iteracija** – fragment koda se izvršava više puta
4. **Podrutine** – apstrakcija kontrole toka: one dozvoljavaju da programer sakrije proizvoljno komplikovan kod iza jednostavnog interfejsa (procedure, funkcije, rutine, metodi, potprogrami)
5. **Rekurzija** – definisanje izraza u terminima samog izraza (direktno ili indirektno)
6. **Konkurentnost** – dva ili više fragmenta programa mogu da se izvršavaju u istom vremenskom intervalu, bilo paralelno na različitim procesorima, bilo isprepletano na istom procesoru
7. **Podrška za rad sa izuzecima** – fragment koda se izvršava očekujući da su neki uslovi ispunjeni, ukoliko se desi suportno, izvršavanje se nastavlja u okviru drugog fragmenta za obradu izuzetka
8. **Nedeterminizam** – redosled izvršavanja se namerno ostavlja nedefinisan, što povlači da izvršavanje proizvoljnim redosledom dovodi do korektnog rezultata

Programski jezici moraju da organizuju podatke na neki način. Tipovi pomažu u dizajniranju programa, proveri ispravnosti programa i u utvrđivanju potrebne memorije za skladištenje podataka. Mehanizmi potrebni za upravljanjem podacima nazivaju se **sistem tipova**. Sistem tipova obično uključuje:

- skup predefinisanih osnovnih tipova (npr int, string...)
- mehanizam građenja novih tipova (npr struct, union)
- mehanizam kontrolisanja tipova
  - pravila za utvrđivanje ekvivalentnosti: kada su dva tipa ista?
  - pravila za utvrđivanje kompatibilnosti: kada se jedan tip može zameniti drugim?
  - pravila izvođenja: kako se dodeljuje tip kompleksnim izrazima?
- pravila za proveru tipova (statička i dinamička provera)

Jezik je tipiziran ako precizira za svaku operaciju nad kojim tipovima podataka može da se izvrši. Jezici kao što je asembler i mašinski jezici nisu tipizirani jer se svaka operacija izvršava nad bitovima fiksne širine. Postoji slabo i jako tipiziranje. Kod jako tipiziranih jezika izvođenje operacije nad podacima pogrešnog tipa će izazvati grešku. Slabo tipizirani jezici izvršavaju implicitne konverzije ukoliko nema poklapanja tipova. Neki jezici dozvoljavaju eksplicitno kastovanje tipova.
Važan je trenutak kada se radi provera tipova:

- Za vreme prevođenja programa – statičko tipiziranje
- Za vreme izvršavanja programa – dinamičko tipiziranje

Statičko tipiziranje je manje sklono greškama ali može da bude previše restriktivno, dok je dinamičko tipiziranje sklonije greškama i teško za debagovanje ali fleksibilnije.

11.5 Prevođenje i izvršavanje

Kompilirani jezici se prevode u mašinski kod koji se izvršava direktno na procesoru računara – faze prevodenja i izvršavanja programa su razdvojene. U toku prevođenja, vrše se razne optimizacije izvršnog koda što ga čini efikasnijim. Jednom preveden kod se može puno puta izvršavati, ali svaka izmena izvornog koda zahteva novo prevođenje.
žindent **Interpretirani jezici** se prevode naredbu po naredbu i neposredno zatim se naredba izvršava – faze prevođenja i izvršavanja nisu razdvojene već su međusobno isprepletene. Rezultat prevođenja se ne smešta u izvršnu datoteku, već je za svako naredno pokretanje potrebna ponovna analiza i prevođenje. Sporiji je, ali prilikom malih izmena koda nije potrebno vršiti analizu celokupnog koda.
Postoje i **hibridni jezici** koji su kombinacija prethodna dva.
Teorijski, svi programski jezici mogu da budu i kompilirani i interpretirani. Moderni programski jezici najčešće pružaju obe mogućnosti, ali u praksi je za neke jezike prirodnije koristiti odgovarajući pristup. Nekada se u fazi razvoja i testiranja koristi interpretirani pristup, a potom se generiše izvršni kod kompilacijom. Prema tome, razlika se zasniva pre na praktičnoj upotrebi nego na samim karakteristikama jezika.

Svaka netrivijalna implementacija jezika višeg nivoa intenzivno koristi rantajm biblioteke. Rantajm sistem se odnosi na skup biblioteka od kojih implementacija jezika zavisi kako bi program mogao ispravno da se izvršava. Neke bibliotečke funkcije rade jednostavne stvari (npr podrška aritmetičkim funkcijama koje nisu implementirane u okviru hardvera, kopiranje sadržaja memorije...) dok neke rade komplikovanije stvari (npr upravljanje hipom, rad sa baferovanim I/O, rad sa grafičkim I/O...).
Neki jezici imaju veoma male rantajm sisteme (npr C), dok ih drugi intenzivno koriste. Virtuelne mašine u potpunosti skrivaju hardver arhitekture nad kojom se program izvršava. C# koristi rantajm sistem CLI, JAVA koristi JVM.

12 Pitanja

12.1 Uvod

1. Značenje reči paradigma i programska paradigma.
2. Uloga programskih paradigmi.
3. Definicija programskog jezika.
4. Povezanost paradigmi i jezika.
5. Razvoj programskih jezika.

12.2 Osnovne paradigme

1. Šta karakteriše proceduralnu paradigmu?
2. Šta karakteriše deklarativnu paradigmu?
3. Koje su osnovne četiri programske paradigme?
4. Nabroj bar četiri dodatne programske paradigme.
5. Koje su osnovne karakteristike imperativne paradigme?
6. Nabroj tri jezika koji pripadaju imperativnoj paradigmi.
7. Koje su osnovne karakteristike ... paradigme?
8. Nabroj tri jezika koji pripadaju ... paradigmi.
9. Šta je programska paradigma?
10. Koje su osnovne programske paradigme?
11. Šta je programski jezik?
12. Koji je odnos programskih jezika i programskih paradigmi?
13. Zašto su nastajale i nastaju nove programske paradigme?

12.3 Dodatne paradigme

1. Koje su osnovne karakteristike komponentne paradigme?
2. Nabroj tri jezika koji pripadaju (podržavaju) komponentnu paradigmu.
3. Koje su osnovne karakteristike ... paradigme?
4. Nabroj tri jezika koji pripadaju ... paradigmi.

12.4 Funkcionalna paradigma

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 referentne prozirnosti i na koji način ovo svojstvo utiče na redosled naredbi u funkciji?
6. Koje su osobine programa u kojima se poštuje pravilo referentne prozirnosti?
7. Da li je moguće u potpunosti zadržati svojstvo referentne prozirnosti?
8. Koji je odnos referentne prozirnosti sa bočnim efektima?
9. Da li je moguće obezbediti promenu stanja programa i istovremeno zadržati svojstvo referentne prozirnosti?
10. Šta su funkcionalni jezici? Šta su čisto 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 proramiranje?
15. Šta je striktna/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?

12.5 Lambda racun i Haskell

1. Koji je formalni okvir funkcionalnog programiranja?
2. Koji se jezik smatra prvim funkcionalnim jezikom?
3. Koja je ekspresivnost lambda računa?
4. Koji su sve sinonimi za lambda izraz?
5. Navesti definiciju lambda terma.
6. Da li čist lambda račun uključuje konstante u definiciji?
7. Navesti primer jednog lambda izraza, objasniti njegovo značenje i primeniti dati izraz na neku konkretnu vrednost.
8. Koja je asocijativnost primene a koja apstrakcije?
9. Navesti ekvivalentan izraz sa zagradama za izraz ...

10. Koje su slobodne a koje vezane promenljive u izrazu ...
11. Navesti definiciju slobodne promenljive? Koje promenljive su vezane?
12. Koja je uloga alfa ekvivalentnosti?
13. Šta su redukcije?
14. Šta je delta redukcija? Navesti primer.
15. Šta je alfa redukcija? Navesti primer.
16. Kada se koristi alfa redukcija?
17. Šta je beta redukcija? Navesti primer.
18. Definisati supstituciju.
19. Navesti primer lambda izraza koji definiše funkciju višeg reda koja prima funkciju kao argument.
20. Navesti primer lambda izraza koji definiše funkciju višeg reda koja ima funkciju kao povratnu vrednost.
21. Čemu služi Karije postupak?
22. Kako se definišu funkcije sa više argumenata?
23. Šta je normalni oblik funkcije?
24. Da li svi izrazi imaju svoj normalni oblik?
25. Navesti svojstvo konfluentnosti.
26. Da li izraz može imati više normalnih obilika?
27. Koja je razlika izmedu aplikativnog i normalnog poretka?
28. Šta govori teorema standardizacije?
29. Šta se dobija lenjom evaluacijom?
30. Koje su osnovne karakteristike Haskela?
31. Šta izračunava naredni Haskel program ...

12.6 Konkurentno programiranje

1. Šta je konkurentna paradigma?
2. Da li su ideje o konkurentnosti nove? Zbog čega je konkurentnost važna?
3. Koji su osnovni nivoi konkurentnosti?
4. Koje su vrste konkurentnosti u odnosu na hardver? Kako se to odnosi na programere i dizajn programskog jezika?
5. Šta je osnovni cilj koji želimo da ostvarimo razvijanjem konkurentnih algoritama?
6. Koji su osnovni razlozi za korišćenje konkurentnog programiranja?
7. Navesti primer upotrebe konkurentnog programiranja za podršku logičkoj strukturi programa.
8. Da li je dobijanje na brzini moguće ostvariti i na jednoprocesorskoj mašini?
9. Koji je hijerarhijski odnos u okviru konkurentne paradigme?
10. Šta je zadatak? Na koji način se zadatak razlikuje od potprograma?
11. Koje su osnovne kategorije zadataka i koje su karakteristike ovih kategorija?
12. Šta je paralelizacija zadataka?
13. Šta je paralelizacija podataka?
14. Navesti primere paralelizacije zadataka i paralelizacije podataka.
15. Koji je odnos ovih paralelizacija?
16. Šta je komunikacija?
17. Koji su osnovni mehanizmi komunikacije?
18. Šta karakteriše slanje poruka u okviru iste mašine, a šta ukoliko je slanje poruka preko mreže?
19. Šta je sinhronizacija?
20. Kakva je sinhronizacija u okviru modela slanja poruka?
21. Kakva je sinhronizacija u okviru modela deljene memorije?
22. Koje su dve osnovne vrste sinhronizacije u okviru modela deljene memorije?
23. Objasniti sinhronizaciju saradnje.
24. Šta je uslov takmičenja?
25. Koji su načini implementiranja sinhronizacije?
26. Šta je koncept napredovanja?
27. Navesti primer uzajamnog blokiranja.
28. Navesti primer živog blokiranja.
29. Navesti primer individualnog izgladnjivanja.
30. Koje vrste uzajamnog isključivanja postoje?
31. Koji je odnos konkurentnosti i potprograma/klasa.
32. Opisati problem filozofa za večerom.
33. Semantika muteksa i katanaca.

12.7 Logičko programiranje i Prolog

1. Šta čini teorijske osnove logičkog programiranja?
2. Na koji način se rešavaju problemi u okviru logičke paradigme?
3. Koji su osnovni predstavnici logičke paradigme?
4. Za koju vrstu problema je pogodno koristiti logičko programiranje?
5. Za koju vrstu problema nije pogodno koristiti logičko programiranje?
6. Definisati logičke i nelogičke simbole logike prvog reda.
7. Definisati term logike prvog reda.
8. Definisati atomičku formulu logike prvog reda.
9. Šta je literal? Šta je klauza?
10. Definisati supstituciju za termove.
11. Definisati supstituciju za atomičke formule.
12. Ukoliko je zadata supstitucija $\sigma = \ldots$ i term $t = \ldots$ izračunati $t\sigma$
13. Šta je problem unifikacije?
14. Kada kažemo da su izrazi unifikabilni?
15. Da li za dva izraza uvek postoji unifikator?
16. Ukoliko za dva izraza postoji unifikator, da li on mora da bude jedinstven?
17. Ukoliko su dati termovi $t?1 = \ldots$ i $t_2 = \ldots$ izračunati jedan unifikator ovih termova.
18. Šta je metod rezolucije?
19. Šta čini teorijske osnove logičkog programiranja?
20. Na koji način se rešavaju problemi u okviru logičke paradigme?
21. Koji su osnovni predstavnici logičke paradigme?
22. Za koju vrstu problema je pogodno koristiti logičko programiranje?
23. Za koju vrstu problema nije pogodno koristiti logičko programiranje?
24. Šta je Hornova klauza i čemu ona odgovara?
25. Šta je supstitucija?
26. Kada se dva terma mogu unifikovati?
27. Od čega se sastoji programiranje u Prologu?
28. Šta su činjenice, šta se pomoću njih opisuje?
29. Šta su pravila i šta se pomoću njih zadaje?
30. Šta odreduju činjenice i pravila?
31. Šta govori pretpostavka o zatvorenosti?
32. Šta su upiti i čemu oni služe?
33. Šta su termovi?
34. Kako se vrši unifikacija nad termima u Prologu?
35. Šta je lista?
36. Napisati pun i skraćen zapis liste od tri elementa a, b i c.
37. Šta omogućava metaprogramiranje?
38. Napisati deklarativno tumačenje naredne Prolog konstrukcije ...
39. Napisati proceduralno tumačenje naredne Prolog konstrukcije ...
40. Šta je stablo izvodenja i čemu ono odgovara u smislu deklarativne/proceduralne semantike?
41. Koji su osnovni elementi stabla izvodenja?
42. Nacrtati stablo izvodenja za naredni Prolog program ...
43. Na koji način redosled tvrdenja u bazi znanja utiče na pronalaženje rešenja u Prologu?
44. Koja je uloga operatora sečenja?
45. Nacrtati stablo izvodenja za naredni program bez operatora sečenja, i sličan program sa operatorom sečenja ...
46. Šta je crveni a šta zeleni operator sečenja?
47. Koja je uloga opreatora sečenja u narednom primeru ...
48. Da li se Hornovim klauzama mogu opisati sva tvrdenja logike prvog reda?
49. Šta Prolog ne može da dokaže?
50. Koje su osobine NOT operatora?
51. Da li Prolog može da obezbedi generisanje efikasnih algoritama?
52. Kakva je kompatibilnost izmedu različitih Prolog kompajlera?
53. Da li je Prolog Tjuring kompletan jezik?
54. Kakav je sistem tipova u prologu?
55. Šta je programiranje ograničenja?
56. Koji su predstavnici paradigme programiranja ograničenja?
57. Po čemu se razlikuje izraz $x < y$ u imperativnoj paradigmi i paradigmi ograničenja?
58. Od čega se sastoji programiranje ograničenja nad konačnim domenom?
59. Napisati program u B-Prologu koji pronalazi sve vrednosti promenljivih X, Y i Z za koje važi da je $X \leq Y$ i $X + Y \leq Z$ pri čemu promenljive pripadaju narednim domenima $X \in \{1,2,...,50\}$, $Y \in \{5,10,..100\}$ i $Z \in \{1,3,5,...99\}$

12.8 Imperativna paradigma

1. Pod kakvim uticajem je nastala imperativna paradigma?
2. Šta je stanje programa?
3. Koje su faze razvoja imperativne paradigme?
4. Koje su karakteristike operacione paradigme?
5. Koji je minimalni skup naredbi operaciona pardigme?
6. Koje su karakteristike strukturne paradigme?
7. Koji je minimalni skup naredbi strukturne pardigme?
8. Koje su karakteristike proceduralne paradigme?
9. Koje vrste prenosa parametara postoje?
10. Kako je u memoriji orgranizovano izvršavanje potprograma?
11. Šta su korutine?
12. Koje su karakteristike modularne paradigme?
13. Šta omogućava modularna paradigma?
14. Kako se rešavaju problemi u okviru proceduralne paradigme?
15. Šta su bočni efekti?
16. Do čega dovode bočni efekti?

12.9 Objektno-orijentisano programiranje

1. Koji su principi funkcionalne dekompozicije i koji su osnovni problemi ovoga pristupa?
2. Šta je osnovni uzrok problema kod rešavanja funkcionalnom dekompozicijom?
3. Zašto je uticaj izmena zahteva važan?
4. Šta je kohezija, a šta kopčanje i kako su povezani?
5. Šta je efekat talasanja i da li je on poželjan?
6. Koji su bili simptomi prve softverske krize?
7. Šta je apstrakcija?
8. Šta je interfejs?
9. Šta implementacija?
10. Objasniti odnos interfejsa i implementacije.
11. Šta je enkapsulacija?
12. Koji je odnos apstrakcije i enkapsulacije?
13. Šta je objekat? (filozofski? konceptulano? u objektnoj terminologiji? specifikacijski? implementaciono?)
14. Kako komuniciraju objekti?
15. Šta je klasa?
16. Koji je odnos klase i objekta?
17. Koji je prvi objektni jezik i kada je nastao?
18. Šta su objektno zasnovani, a šta objektno orijenitsani jezici?
19. Koji su najpopularniji objektno orijentisani jezici?
20. Šta je nasleđivanje?
21. Na koje načine se koristi nasledivanje? Šta je proširivanje, a šta specijalizacija?
22. Šta omogućava nasledivanje?
23. Šta je višestruko nasledivanje?
24. Koji jezici omogućavaju višestruko nasleđivanje, a koji ne?
25. Koje su osnovne vidljivosti koje klase definišu?
26. Šta je polimorfizam?
27. Koja je razlika izmedu preopterećivanja i predefinisanja?
28. Kada se koristi statičko a kada dinamičko vezivanje?
29. Šta definišu apstraktne klase?
30. Koje su mogućnosti generičkog programiranja?
31. Obrazložiti sličnosti i razlike strukturnog i OO programiranja?
32. Koje su osnovne prednosti OO programiranja u odnosu na strukturno programiranje?

12.10 Osnovna svojstva programskih jezika

1. Koja su osnovna svojstva programskih jezika?
2. Koji formalizam se koristi za opisivanje sintakse programskog jezika?
3. Šta definiše semantkika programskog jezika?
4. Koji su formalni okviri za definisanje semantike programskih jezika?
5. Šta je ime?
6. Šta je povezivanje?
7. Koja su moguća vremena povezivanja?
8. Šta je doseg?
9. Šta je kontrola toka?
10. Koji su mehanizmi odredivanja kontrole toka?
11. Šta je sistem tipova i šta on uključuje?
12. Šta je tipiziranje i kakvo tipiziranje postoji?
13. Kada se radi provera tipova?
14. Koja je razlika između kompiliranja i interpretiranja?
15. Šta je rantajm sistem?

13 Literatura

Uvod

- Peter Van Roy, Seif Haridi – Concepts, Techniques, and Models of Computer Programming, MIT Press, 2003.
- Deo materijala je preuzet od prof Dušana Tošica, iz istoimenog kursa

Skript programiranje i programiranje ograničenja

- Programming language pragmatics – Michael L. Scott, 4th edition, Elsevier
- Concepts of programming languages – Robert W. Sebesta
- Slajdovi prof Dušana Tošica sa istoimenog kursa
- Labix.org – Module constraint (API)
- python-constraint 1.3.1
- Constraint programming blog

Haskell

- https://www.haskell.org/
- https://www.haskell.org/documentation
- Real world Haskell – http://book.realworldhaskell.org/
- http://learnxinyminutes.com/docs/haskell/

Konkurentno programiranje

- Programming Language Pragmatics, Third Edition, 2009 by Michael L. Scott
- Concepts of Pprogramming Languages, Tenth Edition, 2012 Robert W. Sebasta

Logičko programiranje

- Programming Language Pragmatics, Third Edition, 2009 by Michael L. Scott
- Concepts of Pprogramming Languages, Tenth Edition, 2012 Robert W. Sebasta
- Skripta Veštačka inteligencija, Predrag Janičić, Mladen Nikolić
- Slajdovi prof Dušana Tošića sa istoimenog kursa
- B-Prolog http://www.picat-lang.org/bprolog/download/manual.pdf
- http://www.hakank.org/bprolog/

Dodatna Absys: the first logic programming language - A retrospective and a com- mentary
http://www.sciencedirect.com/science/article/pii/0743106690900309

Dodatna The early years of logic programming - Robert Kowalski
http://www.doc.ic.ac.uk/ rak/papers/the%20early%20years.pdf
http://www.doc.ic.ac.uk/ rak/papers/History.pdf

Dodatna D. Tošić, R. Protić: PROLOG kroz primere, Tehnička knjiga, Beograd,1991.

Dodatna B. Bajković, M.Durišić, S.Matković: PROLOG i logika, Krug, Beograd,1998.

Imperativna paradigma

- Programming Language Pragmatics, Third Edition, 2009 by Michael L. Scott
- Deo materijala je preuzet od prof Dušana Tošica, iz istoimenog kursa

Objektno-orijentisano programiranje

- The Object-Oriented Paradigm, Kenneth M. Anderson, 2012
- C++ Izvornik — Lippmna Lajoie

- Object Oriented Programming with Java: Essentials and Applications - Rajkumar Buyya, S. Thamarai Selvi, Xingchen Chu
http://www.buyya.com/java/Chapter1.pdf

Osnovna svojstva programskih jezika

- Programming Language Pragmatics, Third Edition, 2009 by Michael L. Scott
- Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages (Pragmatic Programmers), 2010 by Bruce A. Tate
- Deo materijala je preuzet od prof Dušana Tošića, iz istoimenog kursa.