Programske paradigme

I - funkcionalno programiranje

1. Šta su funkcije višeg reda? Navesti primere nekih funkcija višeg reda.
2. Kako je LISP dobio ime?
3. Šta je Haskell i kako je dobio ime?
4. Šta je GHC i kako je dobio ime?
5. Navesti primere korišćenja listi i torki.
6. Šta je stanje programa?
7. Poređenje svojstava jezika - zadaci.
8. Određivanje tipa datih funkcija ili konstruisanje funkcija sa zadatom namenom - zadaci.
9. Kog je tipa konstanta u Haskellu?
10. Šta su striktni, a šta nestriktni izrazi? Primeri.
11. Na koji način se u funkcionalnim jezicima vodi računa o memoriji?
12. Navesti induktivnu definiciju građenja lambda termova.
13. Navesti induktivnu definiciju slobodne promenljive.
14. Ispitivanje alfa ekvivalentnosti termova - zadaci.
15. Izvođenje normalnog oblika - zadaci.

II - konkurentno programiranje | logičko programiranje

1. Šta je koncept napredovanja?
2. Koji problemi mogu da se jave ako nemamo sinhronizaciju?
3. Paralelizacija - zadaci.
4. Unifikacija - zadaci.
5. Metod rezolucije - zadaci.
6. Deklarativna i proceduralna interpretacija - zadaci.
7. Stablo izvođenja - zadaci.
8. Operator NOT - zadaci.

III - uvod u paradigme | programiranje ograničenja

1. Šta je TIOBE indeks, kako se računa i koliko često?
2. Koje su prednosti i mane mašinski zavisnih i mašinski nezavisnih programskih jezika?
3. Koja su prva četiri programska jezika i kada su nastali?
4. Koje su osnovne primene programiranja ograničenja?

Uvod u programske jezike i paradigme

Jezik je skup pravila za komunikaciju između subjekata. **Programski jezik** je skup sintaktičkih i semantičkih pravila koja se koriste za opis (definiciju) računarskih programa. Programski jezici se koriste za komunikaciju ljudi i računara, ali i za međusobnu komunikaciju među računarima i međusobnu komunikaciju među ljudima. Postoji veliki broj programskih jezika, broj se kreće u hiljadama, ali nisu svi programski jezici jednako važni i zastupljeni. |**TIOBE Index** predstavlja indikator popularnosti programskih jezika. Indeks se računa jednom mesečno na osnovu velikog broja faktora koji daju celokupnu sliku o popularnosti jezika, kao što su broj korisnika, broj održanih kurseva, zastupljenost na Gitu i slično.|

Reč paradigma označava uzor, obrazac, šablon i obično se koristi da označi vrstu objekata koji imaju zajedničke karakteristike. **Programska paradigma** predstavlja određeni programski stil, šablon, odnosno način programiranja. Broj programskih paradigmi nije veliki kao broj programskih jezika i izučavanjem određene programske paradigme upoznajemo globalna svojstva jezika koji pripadaju toj paradigmi. Samim tim poznavanje paradigme značajno olakšava učenje programskog jezika koji joj pripada. Svakoj programskoj paradigmi pripada više programskih jezika i potrebno je izučiti svojstva najistaknutijih predstavnika programskih paradigmi ("Koliko predstavnika različitih paradigmi znaš, toliko vrediš"). Takođe, jedan programski jezik može podržati više različitih paradigmi. Na primer, C++ podržava proceduralni, objektno-orijentisani, funkcionalni i još neke stilove programiranja.

Prvi elektronski računari nastali su krajem 30-ih godina 20. veka, kao što je ABC koji se koristio za rešavanje sistema linearnih jednačina. Krajem 1940. godine dolazi do konceptualne promene u vidu **fon Nojmanove arhitekture** i do razdvajanja hardvera i softvera. Programske jezike u skladu sa tim možemo podeliti na **mašinski zavisne**, na primer asembleri, i na **mašinski nezavisne**, na primer Python, C++ itd. |Prednosti mašinski zavisnih programskih jezika su veća kontrola nad memorijom i načinom izvršavanja konkretnih operacija, a mane to što su komplikovaniji, teži za razumevanje, zahtevaju mnogo vremena za kodiranje osnovnih operacija i nisu prenosivi, odnosno zavise od konkretne mašine za koju je program pisan. Prednosti mašinski nezavisnih jezika su lakše razumevanje, činjenica da su sličniji prirodnim jezicima i da prilikom kodiranja ne moramo razmišljati o konkretnom načinu izvršavanja osnovnih operacija, prenosivost i brže kodiranje, a mane to što sa druge strane gubimo kontrolu nad memorijom i izvršavanjem konkretnih operacija.|

|Prvi programski jezici nastaju krajem 50-ih godina 20. veka i to su bili **FORTRAN** koji se koristio za matematička izračunavanja, **LISP** koji je bio zasnovan na funkcionalnoj paradigmi i kome je najduže vremena trebalo da se razvije i bude prihvaćen i **COBOL** koji je napravljen za ljude koji nisu bili matematičari i programeri i koristio se u preduzećima, fabrikama i slično. U istom periodu nastaje i **ALGOL** koji se koristio za različita izračunavanja.| Šezdesetih godina nastaju Simula i Basic, 70-tih C, Pascal i Prolog, 80-tih C++ i Erlang, 90-tih Haskell, Python, PHP, Ruby, JavaScript itd. Tokom 2000-tih nastaju C#, F#, Scala, Elixir i mnogi drugi. U početku je svaki jezik pripadao samo jednoj paradigmi, a kasnije su jezici implementirali funkcionalnosti različitih paradigmi. Nastanak i razvoj programskih jezika se može prikazati pomoću razvojnog stabla, gde je prikazano kako i kada se koji jezik razvijao i pod uticajem kojih drugih jezika.

Ne postoji jedinstveno shvatanje programskih paradigmi, a samim tim ni jedinstvena podela. Najopštija podela programskih paradigmi je na **proceduralnu**, gde je osnovni zadatak programera da opiše način kojim se dolazi do rešenja problema, i **deklarativnu paradigmu**, gde je osnovni zadatak programera da precizno opiše problem, dok se mehanizam programskog jezika bavi pronalaženjem rešenja problema. Osnovnim programskim paradigmama smatraju se:

- **imperativna paradigma**
- **objektno-orijentisana paradigma**
- **funkcionalna paradigma**
- **logička paradigma**

Ostale paradigme se često tretiraju kao podparadigme ili kombinacije osnovnih. Neke dodatne programske paradigme su:

- **komponentna paradigma**
- **paradigma upitnih jezika**
- **paradigma programiranja ograničenja**
- **generička paradigma**
- **vizuelna paradigma**
- **konkurentna paradigma**
- **reaktivna paradigma**
- **skript paradigma**

Programiranje ograničenja

Programiranje ograničenja je deklarativna programska paradigma gde se nad upravljačkim promenljivama zadaju određena ograničenja. Dakle, ne zadaje se postupak kojim se rešava problem, već se postavljaju uslovi koje promenljive moraju da ispunjavaju. Ograničenja se razlikuju od ograničenja u imperativnoj paradigmi. Na primer, x < y se u imperativnoj paradigmi evaluira u tačno ili netačno, dok u paradigmi programiranja ograničenja zadaje relaciju između objekata x i y koja mora uvek da važi. Ograničenja mogu da budu različitih vrsta, kao što su **ograničenja iskazne logike** (A ili B je tačno), **linearna ograničenja** ($x \leq 10$), **ograničenja nad konačnim domenima** i slično. Programiranje ograničenja je savremen pristup rešavanju teških kombinatornih problema. U skladu sa tim, ima primene pre svega u operacionim istraživanjima, odnosno u rešavanju kombinatornih i optimizacionih problema. Osnovna ideja je da korisnik opiše problem na odgovarajući način, pomoću ograničenja, a od sistema se očekuje da izračuna rešenje, odnosno vrednosti promenljivih koje zadovoljavaju postavljena ograničenja. Podrška za programiranje ograničenja je ili ugrađena u programski jezik ili je data preko neke biblioteke. Različiti jezici koriste različite biblioteke koje mogu imati različite pristupe u rešavanju problema, ali nije neophodno poznavati konkretne algoritme, već je dovoljno opisati problem. **Programiranje ograničenja nad konačnim domenom** podrazumeva da su sve promenljive definisane na konačnim domenima. Sastoji se od tri dela:

- **Generisanje promenljivih i njihovih domena**
- **Generisanje ograničenja nad promenljivama**
- **Obeležavanje** - podrazumeva označavanje promenljivih čiju vrednost želimo da izračunamo

Programiranje ograničenja nad konačnim domenom je u Pythonu podržano kroz modul **python-constraint**. U njemu se podrazumeva da se vrednost izračunava za sve promenljive koje su generisane, pa je treći korak podrazumevan i potrebno je samo generisati promenljive, njihove domene i ograničenja nad njima. Neka opšta ograničenja:

- **AllDifferentConstraint()** - vrednosti svih promenljivih moraju biti različite
- **AllEqualConstraint()** - vrednosti svih promenljivih moraju biti iste
- **MaxSumConstraint(s, [težine])** - suma vrednosti promenljivih pomnožena sa prosleđenim težinama ne prelazi vrednost s
- **ExactSumConstraint(s, [težine])** - suma vrednosti promenljivih pomnožena sa prosleđenim težinama mora biti jednaka vrednosti s
- **InSetConstraint(skup)** - vrednosti svih promenljivih moraju se nalaziti u prosleđenom skupu
- **NotInSetConstraint(skup)** - vrednosti svih promenljivih ne smeju se nalaziti u prosleđenom skupu
- **SomeInSetConstraint(skup)** - vrednosti nekih promenljivih moraju se nalaziti u prosleđenom skupu
- **SomeNotInSetConstraint(skup)** - vrednosti nekih promenljivih ne smeju se nalaziti u prosleđenom skupu

Primer: **Kriptoaritmetike** su matematičke igre u kojima se rešavaju jednačine kod kojih su cifre brojeva zamenjene određenim slovima. Na primer:

$$\begin{array}{r} SEND \\ +MORE \\ \hline MONEY \end{array}$$

```
import constraint

# Definišemo promenljive i njihove domene (S i M ne smeju biti 0)
problem = constraint.Problem()
problem.addVariables('SM', range(1, 10))
problem.addVariables('ENDORY', range(10))

# Definišemo ograničenje (SEND + MORE = MONEY)
def ogranicenje(s, e, n, d, m, o, r, y):
        if(1000*s+100*e+10*n+d+1000*m+100*o+10*r+e) == (10000*m+1000*o+100*n+10*e+y):
                return True
# Dodajemo definisano ograničenje na promenljive i ograničenje da su sve cifre različite
problem.addConstraint(ogranicenje, "SENDMORY")
problem.addConstraint(constraint.AllDifferentConstraint())

# Rešavamo problem i dobijamo sva moguća rešenja
resenja = problem.getSolutions()
```

Programiranje ograničenja je nastalo u okviru logičkog programiranja i često je raspoloživo u okviru sistema za logičko programiranje. **Programiranje ograničenja u logici** je vrsta programiranja gde je logičko programiranje prošireno tako da sadrži koncepte programiranja ograničenja. **B-Prolog** je implementacija Prologa koja sadrži bogat sistem za programiranje ograničenja. Podržava programiranje ograničenja na **konačnom** i **iskaznom domenu**. Postoje **opšta ograničenja**, na primer all_different ili all_distinct, kao i **numerička ograničenja** koja počinju znakom #. Za razliku od Pythona, u B-Prologu treći korak, obeležavanje, nije podrazumevan. Rešenje prethodnog primera u B-Prologu:

```
% Definišemo promenljive, njihove domene i postavljamo ograničenja nad njima
% Na kraju vršimo obeležavanje promenljivih čije vrednosti želimo dobiti (u ovom slučaju sve)
funkcija(promenljive) :- promenljive = [S, E, N, D, M, O, R, Y],
promenljive :: 0..9,
S #\= 0,
M #\= 0,
all_different(promenljive),
1000*S+100*E+10*N+D+1000*M+100*O+10*R+E #= 10000*M+1000*O+100*N+10*E+Y,
labeling(promenljive).
```

Funkcionalno programiranje

Uvod u funkcionalno programiranje

U imperativnoj paradigmi izvršavanje programa svodi se na izvršavanje naredbi. Izvršavanje programa može se svesti i na evaluaciju izraza, pa u zavisnosti od izraza imamo **logičku paradigmu** gde su izrazi relacije čiji je rezultat tačno ili netačno i **funkcionalnu paradigmu** gde su izrazi funkcije čiji rezultati mogu biti različite vrednosti. Funkcije srećemo i u drugim paradigmama, ali radi se o suštinski različitim pojmovima iako imaju isto ime. Pomenute paradigme temelje se na različitim teorijskim modelima. Formalizam za imperativne jezike su **Tjuringova** i **URM mašina**, formalizam za logičke jezike je **logika prvog reda**, a formalizam za funkcionalne jezike je **lambda račun**. Funkcionalni jezici su mnogo bliži svom teorijskom modelu nego imperativni jezici, pa je poznavanje lambda računa veoma važno. Ekspresivnost funkcionalnih jezika je ekvivalentna ekspresivnosti lambda računa. Ekspresivnost lambda računa je ekvivalentna ekspresivnosti Tjuringovih mašina. Ekspresivnost Tjuringovih mašina je ekvivalentna ekspresivnosti imperativnih jezika. Odavde sledi da svi programi koji se mogu napisati imperativnim stilom, mogu se napisati i funkcionalnim stilom.

Funkcionalna paradigma zasniva se na pojmu matematičkih funkcija, odakle potiče i njen naziv. Njen osnovni cilj je da oponaša matematičke funkcije i zasniva se na izračunavanju izraza kombinovanjem funkcija. Osnovne aktivnosti su:

- **definisanje funkcija**
- **primena funkcija**
- **kompozicija funkcija**

Dakle, program u funkcionalnom programiranju je niz definicija i poziva (primena) funkcija, a izvršavanje programa je evaluacija funkcija. Da bi se uspešno programiralo treba ugraditi osnovne funkcije u sam programski jezik i obezbediti mehanizme za formiranje kompleksnijih funkcija, strukture za prezentovanje podataka i formiranje biblioteka funkcija koje se mogu kasnije koristiti. Ukoliko je jezik dobro definisan, broj osnovnih funkcija i struktura podataka je relativno mali. |Funkcija je u funkcionalnom programiranju ravnopravna sa ostalim tipovima podataka, odnosno može biti povratna vrednost ili parametar druge funkcije. Funkcije koje primaju druge funkcije kao parametre ili vraćaju funkciju kao povratnu vrednost, ili oba, nazivaju se **funkcije višeg reda**. Najvažnije među njima su:

- **map** - primenjuje funkciju na listu parametara, npr. `map (+5) [1, 5, 6, 2, 3]` uvećava svaki element liste za 5. Može se primeniti i **kompozicija**, npr. `map ((>10)) . (^2)) [1, 5, 6, 2, 3]` kvadrira elemente, a zatim ih poredi sa 10 i vraća True za elemente koji su veći od 10.
- **filter** - izdvaja elemente iz liste parametara koji zadovoljavaju uslov funkcije koja vraća true/false, npr. funkcija `filter (>3) [1, 5, 6, 2, 3]` vraća listu samo onih elemenata koji su veći od 3.
- **fold**/**reduce** - "svodi" listu elemenata na jednu vrednost koja se dobija primenom neke funkcije na elemente, npr. `foldl (+) 0 [1, 5, 6, 2, 3]` vraća 0 + 1 + 5 + 6 + 2 + 3 = 17, a `foldr (-) 5 [1, 5, 6, 2, 3]` vraća vrednost 1 - (5 - (6 - (2 - (3 - 5)))) = -2.|

Funkcionalna paradigma nastaje 1959. godine kada nastaje njen najistaknutiji predstavnik - programski jezik **LISP** (**LIS**t **P**rocessing). LISP i funkcionalno programiranje nisu u početku bili opšte prihvaćeni i dugo je trebalo da budu razvijeni, pa sedamdesete godine karakteriše stagnacija u razvoju funkcionalne paradigme. Krajem sedamdesetih **Bakus** dobija Tjuringovu nagradu za razvoj Fortrana i tom prilikom drži predavanje sa poentom da su čisti funkcionalni programski jezici bolji od imperativnih. Rekao je da su funkcionalni jezici lakši za razumevanje, čitljiviji, pouzdaniji, verovatnije ispravni i da je vrednost izraza nezavisna od konteksta u kojem se izraz nalazi. Osnovna karakteristika čistih funkcionalnih jezika je **transparentnost referenci**, odnosno koliko god puta da pozovemo funkciju za iste argumente uvek ćemo dobiti istu povratnu vrednost, što kao posledicu ima nepostojanje propratnih (bočnih) efekata. U okviru svog predavanja Bakus je koristio svoj funkcionalni jezik FP, koji je motivisao dalje istraživanje funkcionalnih jezika, iako sam nije zaživeo. Nakon ovoga, krajem sedamdesetih i osamdesetih nastaju jezici Scheme, ML, Miranda, Erlang, SML, a 1990. godine nastaje **Haskell**. Tokom 2000-tih godina nastaju F#, Scala i Elixir, a danas ogromna većina programskih jezika podržava funkcionalne koncepte, kao npr. C++, Java, Python. Funkcionalni jezici su obični jezici opšte namene i koriste se u raznim domenima kao što su obrade baza podataka, bioinformatika, statička analiza programa, finansijsko modelovanje i tako dalje. Pogodni su i za paralelizaciju, što ih čini posebno popularnim za paralelno i distribuirano programiranje.

Haskell

Haskell je čist funkcionalni programski jezik koji je nastao 1990. godine. Ime je dobio po matematičaru i logičaru Haskelu Kariju koji je doprineo razvoju logike. Haskell implementira **lenju evaluaciju**, tj. izbegava nepotrebna izračunavanja. U njemu postoji i automatsko zaključivanje tipova, pa se tipovi ne moraju uvek navoditi. On je **statički tipiziran** jezik, odnosno tipovi se određuju u fazi kompilacije, ali je i **strogo tipiziran** jezik, odnosno svi tipovi se moraju poklapati i nema implicitnih konverzija. Sadrži podršku za paralelno i distribuirano programiranje. Podržava **parametarski polimorfizam**, tj. funkcije mogu biti definisane tako da rade sa različitim tipovima podataka i primenjuju se isto nezavisno od toga koji je konkretan tip. Takođe podržava i preopterećivanje, kao i kompaktan i ekspresivan način definisanja listi kao osnovnih struktura funkcionalnog programiranja. Podržava i funkcije višeg reda koje omogućavaju visok nivo apstrakcije. Standardizaciju jezika sprovodi međunarodni odbor Haskell Commitee.

GHC (**Glasgow Haskell Compiler**) je interaktivni interpreter i kompajler koji proizvodi optimizovan kod. Haskell se kotira vrlo visoko po brzini izvršavanja, nekada i u nivou sa C-om. Jednostavne primere možemo isprobavati u interpreteru koji se poziva sa `ghci`. Složenije programe pišemo u datoteci sa ekstenzijom hs. Ukoliko želimo da napravimo izvršnu verziju, potrebno je da definišemo main funkciju od koje će početi izračunavanje. Prevođenje vrši kompajler koji se poziva sa `ghc ime.hs`, a izvršni fajl će imati isto ime kao izvorni. Biblioteka **Prelude** definiše sve osnovne funkcije.

Svojstva funkcionalnih jezika

Funkcionalni jezici imaju osnovne tipove podataka kao što su celobrojna vrednost, realne vrednosti, logičke vrednosti i stringovi. Na primer, osnovni tipovi u Haskellu su Int, Integer, Float, Bool, Char, String. Int predstavlja standardnu celobroju vrednost fiksne dužine, a Integer neograničeno velike vrednosti koje mogu biti ograničene samo memorijom računara. Osnovna struktura podataka su **liste** koje predstavljaju kolekcije proizvoljnog broja vrednosti istog tipa i najčešće su implementirane preko povezanih listi. Moguće je generisati i konačne i beskonačne liste, sa različitim koracima. Postoje i razne funkcije za rad sa listama kao što su head, length, sum itd. Na primer, `[0, 2..] !! 50` označava beskonačnu listu parnih brojeva iz koje tražimo pedeseti element. Iako je u teoriji lista beskonačna, u praksi se ona generiše tek kada tražimo taj element i to samo do 50. člana. Funkcionalni jezici podržavaju i **torke** koje predstavljaju kolekcije fiksiranog broja vrednosti potencijalno različitih tipova i najčešće zauzimaju kontinualni prostor u memoriji. Liste i torke mogu biti parametri i povratne vrednosti funkcija. Izbor između liste i torke zavisi od konkretnog problema. Na primer, ako želimo da opišemo studenta preko vrednosti za ime, prezime, indeks, godište i slično, moramo koristiti torke jer su tipovi različiti. Ako želimo da opišemo polinome proizvoljnog stepena moramo koristiti liste, jer su torke konačne. Ako želimo da opišemo polinome trećeg stepena možemo koristiti i liste i torke.

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. Odnosno, oni mogu da se čuvaju u promenljivama, da se prosleđuju funkcijama, kreiraju u okviru funkcija i da se vrate kao povratna vrednost funkcija. U funkcionalnom programiranju funkcije su građani prvog reda. Primena funkcija određena je uparivanjem imena funkcija sa određenim elementom iz domena i predstavlja izračunavanje vrednosti funkcije za taj konkretan argument. **Funkcije višeg reda** su funkcije koje imaju jednu ili više funkcija kao parametre ili imaju funkciju kao rezultat, ili oba. Najvažnije funkcije višeg reda su map, filter i reduce.

Matematička funkcija je preslikavanje elemenata jednog skupa koji nazivamo **domen** u elemente drugog skupa koji nazivamo **kodomen**. Definicija funkcija uključuje zadavanje domena, kodomena i preslikavanja, a funkcije se onda primenjuju na pojedinačne elemente iz domena koji se zadaju kao argumenti funkcija. Domen može da bude i Dekartov proizvod različitih skupova, tj. funkcija može da ima više od jednog parametra. Na primer, $kub(x) \equiv x \cdot x \cdot x$ je funkcija definisana na skupu realnih brojeva, a kodomen je takođe skup realnih brojeva.

Parametar x može da bude bilo koji član domena, ali se fiksira kada god se evaluira u okviru izraza funkcije i njegova vrednost se ne može promeniti. Osnovna karakteristika matematičkih funkcija je da se izračunavanje preslikavanja kontroliše **rekurzijom** i **kondicionim izrazima**, a ne sekvencom i iteracijom kao što je to slučaj kod imperativnih jezika. Na primer, funkcija faktorijel ima domen i kodomen prirodnih brojeva, a preslikavanje se zadaje izrazom $n! \equiv \begin{cases} 1, & n = 0 \\ n(n-1)!, & n > 0 \end{cases}$.

Vrednosti preslikavanja elemenata iz domena su uvek isti elementi iz kodomena jer ne postoje bočni efekti i funkcije ne zavise od drugih spoljašnjih promenljivih. Kod imperativnih jezika vrednost funkcije može da zavisi od tekućih vrednosti različitih globalnih ili nelokalnih promenljivih. |**Stanje programa** čine sve vrednosti u memoriji kojima program u toku izvršavanja ima pristup.| Imperativni jezici imaju implicitno stanje i izvršavanje programa se svodi na postepeno menjanje tog stanja izvođenjem pojedinačnih naredbi i to stanje se predstavlja promenljivama. Promena stanja se najčešće vrši naredbama dodele. Zbog svega toga, potrebno je razumeti korišćenje promenljivih i njihove promene tokom izvršavanja da bi se razumeo program, što je veoma zahtevno za velike programe. Funkcionalni jezici nemaju impliticno stanje, čime je razumevanje efekta rada funkcije značajno olakšano. Nepostojanje naredbe dodele i promenljivih u imperativnom smislu ima za posledicu da iterativne konstrukcije nisu moguće pa se ponavljanje ostvaruje kroz **rekurziju**. U funkcionalnim jezicima rekurzija je prirodnija nego u imperativnim, ali je i često skrivena kroz upotrebu osnovnih funkcija višeg reda.

Funkcionalne programske jezike karakteriše **transparentnost referenci**, odnosno vrednost izraza je svuda jedinstveno određena. Ako se na dva mesta referencira na isti izraz, onda je vrednost ta dva izraza ista. Ovo svojstvo govori da redosled naredbi nije bitan, tj. funkcije možemo kombinovati na razne načine. Programi sa transparentnim referencama su formalno koncizni, prikladni za formalnu verifikaciju, manje podložni greškama i lakše ih je transformisati, optimizovati i paralelizovati. Neki algoritmi se suštinski temelje na promeni stanja, na primer algoritam generisanja slučajnih brojeva, a neke funkcije postoje samo zbog svojih bočnih efekata, na primer funkcije za obradu ulaza i izlaza. Iz ovih razloga, većina funkcionalnih programskih jezika dopušta **kontrolisane propratne efekte**, tj. imperativnost je prisutna na razne načine u većoj ili manjoj meri, što se naziva **monadičko programiranje**. Na primer, većina funkcionalnih jezika uključuje promenljive i konstrukte koji se ponašaju kao naredbe dodele. U zavisnosti od prisutnosti imperativnih osobina, postoje **čisti funkcionalni jezici** koji nemaju propratne efekte i jezici koji nisu čisti. Postoji mali broj čistih funkcionalnih jezika, kao što su Haskell, Clean, Miranda. Neki imperativni jezici podržavaju neke funkcionalne koncepte, na primer C, C++, C#, Java.

|

Funkcionalni jezici mogu imati statičko (Haskell) ili dinamičko zaključivanje tipova (Elixir). Najčešće su jako tipizirani jezici, odnosno svi tipovi moraju da se poklapaju i nema implicitnih konverzija. S druge strane, nije neophodno navoditi sve tipove, kompajler je u stanju sam da zaključi tipove.|

U Haskellu možemo definisati funkciju bez navođenja tipa, na primer `let uvecaj x = x + 1` definiše funkciju kojoj onda možemo proslediti vrednosti tipova Int, Integer, Float, Double, jer će kompajler sam zaključiti da se radi o nekom numeričkom tipu. Sa druge strane može se i navesti tip, na primer `Bool -> Bool` ili `(Int, Int) -> Int`. Mogu se koristiti i **tipske promenljive** a, b, c i slično. Na primer, `[a] -> Int` označava da se lista bilo kog tipa slika u Int, a `[a] -> a` označava da se lista određenog tipa slika u jednu vrednost istog tog tipa. Dodatne uslove možemo zadati preko **tipskih razreda** koji definišu koje funkcije neki tip mora da implementira da bi pripadao tom razredu, tj. klasi:

- **Eq** - tipovi sa jednakošću == i /=
- **Ord** (nasleđuje Eq) - tipovi sa uređenjem <, >, $\leq$, $\geq$, ...
- **Num** - tipovi sa numeričkim operacijama +, -, *, ...
- **Integral** (nasleđuje Num) - celobrojni tipovi sa operacijama div i mod, nasleđuju ga **Int** i **Integer**
- **Fractional** (nasleđuje Num) - razlomački tipovi sa operacijama / i recip, nasleđuju ga **Float** i **Double**
- **Show** - tipovi koji se mogu konvertovati u string
- **Read** - tipovi koji se mogu konstruisati na osnovu stringa
- **Bounded** - tipovi koji imaju gornje i donje ograničenje

|Primeri tipova funkcija:

- `sum :: Num a => [a] -> a` - sumira listu elemenata numeričkog tipa i vraća isti taj tip
- `elem :: Eq a => [a] -> Bool` - proverava da li se vrednost tipa `a` nalazi u listi elemenata tipa `a` i vraća True/False vrednost
- `max :: Ord a => a -> a -> a` - poredi dva objekta istog tipa i vraća veći koji je istog tog tipa
- `map :: (a -> b) -> [a] -> [b]` - prvi argument je funkcija koja slika objekte tipa `a` u objekte tipa `b`, drugi argument je lista parametara tipa `a`, a rezultat je lista vrednosti koje se dobijaju primenom funkcije i one su tipa `b`
- `head :: [a] -> a` - uzima prvi element iz liste
- `fst :: (a, b) -> a` - uzima prvi element iz para, tj. torke od dva elementa
- `inc :: Num a => a -> a` - uvećava objekat numeričkog tipa za jedan i vraća objekat istog tog tipa
- `(==) :: Eq a => a -> a -> Bool` - poredi dva objekta istog tipa i vraća Bool vrednost, može da se stavi i klasa Ord jer nasleđuje klasu Eq, ali onda zahtevamo da taj tip implementira operacije >, < i slično što nije neophodno za poređenje po jednakosti
- `(>) :: Ord a => a -> a -> Bool` - poredi dva objekta istog tipa i vraća Bool vrednost
- `f :: (Num a, Show b) => (a -> b) -> (a -> b)` - slika funkciju u funkciju istog tipa, a ta funkcija slika numerički tip `a` u tip koji može da se konvertuje u string `b`
- `20 :: Num a => a` - tip svake konstante je Num||

Prvi funkcionalni jezik, Lisp, koristi sintaksu koja je drugačija od sintakse koja se koristi u imperativnim jezicima. Zbog velikog broja zagrada dobija nadimak "Lots of Irritating Silly Parentheses". Kao multiparadigmatski jezik, dozvoljava i upotrebu konstrukcija koje odgovaraju petljama. Noviji funkcionalni jezici koriste sintaksu koja je slična sintaksi imperativnih jezika. Izrazi mogu da se uparuju sa obrascima, a rezultat je izraz koji odgovara prvom uparenom obrascu, što se naziva **uklapanje obrazaca** (**pattern matching**). Sa uparenim obrascem vrši se vezivanje, a ako vezivanje nije potrebno koristi se **wildcard _**. Na primer, funkcija za računanje NZD-a dva broja može se definisati na sledeći način:

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

-- ili

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

Obrasci su posebno korisni u radu sa listama:

- `[]` - prazna lista
- `[x]` - lista sa tačno jednim elementom
- `h:t` - lista sa bar jednim elementom (neprazna lista)
- `[x, y]` - lista sa tačno dva elementa
- `x:y:t` - lista sa bar dva elementa

Skraćenice (**comprehensions**) predstavljaju sintaksni dodatak koji omogućava skraćen način zapisa nekih čestih konstrukata. Uglavnom su vezane za definisanje listi na način blizak matematičkim definicijama i imaju naredni oblik: `[izraz | generator, filter]`. Na primer, skraćenica `[(x, y) | x <- [1..10], y <- [1..10], x + y == 10]` predstavlja sve parove brojeva od 1 do 10 čiji je zbir jednak 10.

Semantika programskog jezika opisuje proces izvršavanja programa na tom jeziku i ona može da se opiše formalno i neformalno. Uloge semantike su razumevanje karakteristika programskog jezika, dokazivanje njegovih svojstava, kao i pomoć programeru da razume kako se program izvršava pre pokretanja i šta mora da obezbedi prilikom kreiranja kompilatora. Jedno od semantičkih svojstava je **striktnost semantike**. Izraz je **striktan** ako nema vrednost kad bar jedan od njegovih operanada nema vrednost, a **nestriktan** kad može da ima vrednost čak i ako neki od njegovih operanada nema vrednost. Na primer, neki striktni izrazi su a + b, a - b, a / b jer je potrebno znati oba operanda da bismo izračunali konačnu vrednost izraza. Izrazi a & b i a || b su nestriktni jer se vrednost može zaključiti ako je a = 0 u slučaju konjukcije i ako je a = 1 u slučaju disjunkcije. Izraz a · b je takođe nestriktan jer ako je a = 0 ceo izraz će biti 0. Semantika je **striktna** ako je svaki izraz tog jezika striktan, a **nestriktna** ako su dozvoljeni i nestriktni izrazi. Kod striktne semantike, prvo se izračunavaju vrednosti svih operanada, pa se onda izračunava vrednost izraza. Ova semantika podržava **prenos parametara po vrednosti** - **call by value**. Kod nestriktne semantike izračunavanje operanada se odlaže sve dok te vrednosti ne budu neophodne što je poznato pod imenom **zadržano** ili **lenjo izračunavanje** (**lazy evaluation**). Samim tim, nestriktna semantika omogućava kreiranje beskonačnih struktura i izraza. Ova semantika omogućava odlaganje izračunavanja argumenata sve dok ne budu neophodne, tj. **prenos parametara po potrebi** - **call by need**. Većina funkcionalnih jezika ima striktnu semantiku, npr. Lisp, Scala i OCaml, dok Miranda i Haskell imaju nestriktnu semantiku.

Funkcionalnim programiranjem definišu se izrazi čije se vrednosti evaluiraju, vrši se oponašanje matematičkih funkcija koje je na visokom nivou apstrakcije i to je veoma udaljeno od konkretnog hardvera računara na kojem se program izvršava. Bez obzira odakle krećemo, moramo stići do asemblera i izvršnog koda. Izvršni program uvek odgovara hardveru računara, a asembler je po prirodi imperativni jezik, pa je proces kompilacije funkcionalnih jezika zbog toga veoma složen. Zbog toga su funkcionalni jezici dugo bili interpretirani jezici. Iako u funkcionalnim jezicima ne postoje promenljive, interno podaci moraju da se čuvaju u dinamičkim promenljivama koje se alociraju na hipu. O dealokaciji te memorije brine se **sakupljač otpadaka** (**garbage collector**).

Naredni razlozi se često navode kao prednosti funkcionalnog programiranja, ali se neki smatraju i manama:

- **Stanje i propratni efekti** - za velike programe teško je pratiti stanje i razumeti propratne efekte i lakše je kada imamo uvek isto ponašanje funkcija, ali svet koji nas okružuje pun je 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 poledica transparentnosti referenci, ali rad sa podacima koji se ne menjaju dovodi do mogućeg rada sa podacima koji su u međuvremenu izmenjeni, dok rad sa podacima koji se menjaju jeste komplikovaniji i zahteva kodiranje kompleksne logike.
- **Stil programiranja** - programi su često kraći, lakši za čitanje i omogućavaju razbijanje koda u manje delove koji imaju jaku koheziju i izraženu modularnost, ali treba ih znati pročitati i razumeti.
- **Produktivnost programera** - produktivnost je veća, ali većina programera ne gradi nove sisteme već radi na održavanju starih koji su pisani u drugim jezicima, koji su možda i imperativni.
- **Efikasnost** - dugo se navodila kao mana, ali danas nije problematična. Merenja pokazuju da je Haskell u nekim slučajevima jednako efikasan kao C.
- **Težina** - često se navodi da je funkcionalno programiranje lako ili teško za učenje, što zavisi od programera do programera.
- **Matematički dokaz ispravnosti** - lakše se konstruiše nego kod imperativnih jezika.
- **Testiranje i debagovanje** - testiranje je jednostavnije jer je svaka funkcija potencijalni kandidat za unit testove. Funkcije ne zavise od stanja sistema što olakšava sintezu test primera i proveru da li je izlaz odgovarajući. Debagovanje je jednostavnije, osim kada imamo nestriktnu semantiku, 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.
- **Biblioteke** - mogu se jednostavno graditi biblioteke funkcija.

Lambda račun

Lambda račun (λ**-calculus**) je formalni model izračunljivosti funkcija koji je 1930. osmislio Alonzo Čerč. Takođe predstavlja i formalni model definisanja algoritama. Zasniva se na apstrakciji i primeni funkcija korišćenjem vezivanja i supstitucije. Funkcije tretira kao izraze koji se postepeno transformišu do rešenja. Iako to nije bila primarna ideja, danas se lambda račun smatra prvim funkcionalnim jezikom. Ekspresivnost lambda računa ekvivalentna je ekspresivnosti Tjuringovih mašina. Lambda račun naglašava pravila za transformaciju izraza i ne zamara se arhitekturom mašine koja to može da ostvari, dok su moderni funkcionalni jezici samo njegove sintaksno ulepšane varijante. Prvo je nastao **netipizirani lambda račun** gde domen funkcije nije bio ugrađen. Kasnije nastaje **tipizirani lambda račun** gde se na funkcije postavlja ograničenje u vidu domena, tj. funkcije mogu da se primenjuju samo na odgovarajući tip podataka.

Definisanje funkcije može se razdvojiti od imenovanja funkcije. Na primer, $sum(x, y) = x + y$ može da se definiše sa $(x,\ y) \to x + y$, a $id(x) = x$ sa $x \to x$. Lambda izraz definiše parametre i preslikavanje funkcije, ali ne i ime, pa se takođe naziva **anonimna** (**bezimena**) **funkcija**. Sintaksa lambda izraza je u obliku $\lambda promenljiva.\, telo$ sa značenjem $promenljiva \to telo$. Na primer:

- identiteta: $\lambda x.\, x$
- inkrementiranje: $\lambda x.\, x + 1$
- kvadriranje: $\lambda x.\, x \cdot x$

Lambda izraz se može primeniti na druge izraze. Koristi se sintaksa u obliku $(\lambda promenljiva.\, telo) izraz$. Na primer, u izrazu $(\lambda x.\, x \cdot x + 3)((\lambda x.\, x + 1)5)$ prvo se na inkrementaciju primeljuje izraz 5 i dobija se 6, a zatim se 6 primenjuje na kvadriranje i uvećavanje za 3 pa se dobija 39.

Ispravni lambda izrazi nazivaju se **lambda termovi**. Oni se sastoje od promenljivih, simbola apstrakcije λ, tačke i zagrada. |Induktivna definicija za građenje lambda termova:

1. **promenljive**: promenljiva je validni lambda term.
2. λ**-apstrakcija**: ako je t lambda term, a x promenljiva, onda je $\lambda x.\, t$ lambda term.
3. λ**-primena**: ako su t i s lambda termovi, onda je $(t\ s)$ lambda term.|

Prirodni brojevi se mogu definisati korišćenjem osnovne definicije lambda računa. Ukoliko lambda račun ne uključuje konstante u definiciji, onda se naziva **čist**. Radi jednostavnosti, numerali se često podrazumevaju i koriste već u okviru same definicije lambda termova i takav lambda račun naziva se **primenjen**. Slično, korišćenjem osnovnog lambda računa mogu se definisati i aritmetičke funkcije. Radi jednostavnosti, u okviru lambda termova koristimo aritmetičke funkcije imenovane na standardni način. Zagrade su takođe veoma bitne i mogu se koristiti da promene značenje termova. Na primer, termovi $\lambda x.\, ((\lambda x.\, x + 1)x)$ i $(\lambda x.\, (\lambda x.\, x + 1))x$ su različiti. Da bi se smanjila upotreba zagrada postoje pravila asocijativnosti za primenu i apstrakciju. **Primena** funkcije je **levo asocijativna**, tj. umesto $(e_1 e_2) e_3$ pišemo $e_1 e_2 e_3$. **Apstrakcija** je **desno asocijativna**, tj. umesto $\lambda x.\, (e_1 e_2)$ pišemo $\lambda x.\, e_1 e_2$. Moguće je skratiti i sekvencu apstrakcija, na primer umesto $\lambda x.\, \lambda y.\, \lambda z.\, e$ pišemo $\lambda xyz.\, e$.

U okviru lambda računa ne postoji koncept deklaracije promenljive. Promenljiva može biti **vezana** i **slobodna**. Na primer, u termu $\lambda x.\, x + y - 1$ promenljiva x je vezana, a promenljiva y slobodna, odnosno slobodne promenljive su one promenljive koje nisu vezane lambda apstrakcijom. |Induktivna definicija slobodne promenljive:

1. **promenljive**: slobodna promenljiva terma x je samo x.
2. **apstrakcija**: skup slobodnih promenljivih terma $\lambda x.\, t$ je skup slobodnih promenljivih terma t bez promenljive x.
3. **primena**: skup slobodnih promenljivih terma $(t\ s)$ je unija skupova slobodnih promenljivih terma t i terma s.|

α **ekvivalentnost** definiše se za lambda termove, sa ciljem da se uhvati intuicija da izbor imena vezane promenljive u lambda računu nije važan. |Da li su sledeći termovi alfa ekvivalentni:

- x i y - **ne** jer nisu vezani u okviru lambda apstrakcije
- $\lambda x.\, x$ i $\lambda y.\, y$ - **da**
- $\lambda x.\, x$ i $\lambda x.\, y$ - **ne** jer je prvi izraz funkcija identiteta, a drugi konstantna funkcija
- $\lambda x. 5 + x/2$ i $\lambda y. 5 + y/2$ - **da**
- $\lambda x. 5 + x/2$ i $\lambda y. 5 + y/3$ - **ne** jer se u prvom deli sa 2, a u drugom sa 3
- $\lambda a.\, a \cdot y - 1$ i $\lambda b.\, b \cdot z - 1$ - **ne** jer y i z nisu iste promenljive
- $\lambda z.\, z \cdot y - 1$ i $\lambda x.\, x \cdot z - 1$ - **ne** jer y i z nisu iste promenljive
- $\lambda ij.\, i - j \cdot 3$ i $\lambda mn.\, m - n \cdot 3$ - **da**
- $\lambda ij.\, i - j \cdot 3$ i $\lambda nm.\, m - n \cdot 3$ - **ne** jer su m i n zamenili mesta u drugom termu|

Za transformacije izraza koriste se **redukcije** i **supstitucije**. Postoje razne vrste redukcija i one se nazivaju slovima grčkog alfabeta. One daju uputstva kako transformisati izraze iz početnog stanja u neko finalno stanje.

1. δ **redukcija** se odnosi na transformacije funkcija koje kao argumente sadrže konstante. Na primer, $3 + 5 \to_\delta 8$. Ako je jasno o kojoj redukciji je reč, onda se piše samo $\to$, tj. $3 + 5 \to 8$.
2. α **redukcija** (**preimenovanje**) dozvoljava da se promene imena vezanim promenljivama. Na primer, α redukcija izraza $\lambda x.\, x$ može biti $\lambda y.\, y$. Termovi koji se razlikuju samo po α konverziji su α ekvivalentni. Alfa preimenovanje je nekada neophodno da bi se izvršila beta redukcija. Treba voditi računa da se pri α redukciji ne promeni značenje terma. Na primer, izraz $\lambda x.\, \lambda x.\, x$ može da se svede na $\lambda y.\, \lambda x.\, x$ jer u oba slučaja dobijamo preslikavanje kojim se neka promenljiva preslikava u funkciju identiteta. Međutim, on ne može da se svede na $\lambda y.\, \lambda x.\, y$ jer time dobijamo preslikavanje kojim se neka promenljiva preslikava u konstantnu funkciju.

3. β **redukcija** u telu lambda izraza formalni argument zamenjuje aktuelnim argumentom i vraća telo funkcije, odnosno svako pojavljivanje promenljive u telu se zamenjuje sa datim izrazom. Kada želimo da izračunamo vrednost funkcije, to odgovara primeni β redukcije sve dok je to moguće. Koristi se sintaksa u obliku $(\lambda promenljiva.\, telo) izraz \rightarrow_\beta [izraz/promenljiva] telo$. Na primer:
   - $(\lambda x.\, x + 1)5 \rightarrow_\beta [5/x](x + 1) \rightarrow 5 + 1 \rightarrow_\delta 6$
   - $(\lambda x.\, x \cdot x + 3)((\lambda x.\, x + 1)5) \rightarrow_\beta [6/x](x \cdot x + 3) \rightarrow 6 \cdot 6 + 3 \rightarrow_\delta 39$
   - $(\lambda x.\, x + 3)((\lambda x.\, x + 5)4) \rightarrow (\lambda x.\, x + 3)(4 + 5) \rightarrow (\lambda x.\, x + 3)9 \rightarrow 12$
   - $(\lambda x.\, x)(\lambda y.\, y) \rightarrow \lambda y.\, y$
   - $(\lambda x.\, x\ x)(\lambda y.\, y) \rightarrow (\lambda y.\, y)(\lambda y.\, y) \rightarrow (\lambda x.\, x)(\lambda y.\, y) \rightarrow \lambda y.\, y$
   - $(\lambda x.\, x(\lambda x.\, x))y \rightarrow (\lambda z.\, z(\lambda x.\, x))y \rightarrow y(\lambda x.\, x)$
   - $(\lambda x.\, x(\lambda x.\, x))(\lambda x.\, x\ x) \rightarrow (\lambda x.\, x(\lambda y.\, y))(\lambda z.\, z\ z) \rightarrow (\lambda z.\, z\ z)(\lambda y.\, y) \rightarrow (\lambda y.\, y)(\lambda y.\, y) \rightarrow \lambda y.\, y$
   - $(\lambda x.\, \lambda y.\, y\ x)(\lambda z.\, u) \rightarrow (\lambda x.\, (\lambda y.\, y\ x))(\lambda z.\, u) \rightarrow \lambda y.\, y(\lambda z.\, u)$
   - $(\lambda x.\, x\ x)(\lambda z.\, u) \rightarrow (\lambda z.\, u)(\lambda z.\, u) \rightarrow u$
   - $(\lambda x.\, x\ x)(\lambda x.\, x\ x) \rightarrow (\lambda y.\, y\ y)(\lambda x.\, x\ x) \rightarrow (\lambda x.\, x\ x)(\lambda x.\, x\ x) \rightarrow\ \ldots$
4. η **redukcija** predstavlja funkcijsko proširenje. Ideja je da se uhvati intuicija po kojoj su dve funkcije jednake ako imaju identično spoljašnje ponašanje, odnosno ako se za sve vrednosti evaluiraju u iste rezultate. Na primer, ako se x ne javlja kao slobodna promenljiva u f, onda važi $\lambda x.\, f\ x \rightarrow_\eta f$ jer oba izraza predstavljaju primenu funkcije f na neku promenljivu.

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

1. **promenljive**:
   - $[N/x]x = N$
   - $[N/x]y = y$, pri čemu je $x \neq y$
2. **apstrakcija**:
   - $[N/x](\lambda x.\, M) = \lambda x.\, M$
   - $[N/x](\lambda y.\, M) = \lambda y.\, ([N/x]M)$, pri čemu je $x \neq y$ i $y \notin N$
3. **primena**:
   - $[N/x](M_1 M_2) = ([N/x])(M_1)([N/x])(M_2)$

Funkcije višeg reda su funkcije koje kao argument ili kao povratnu vrednost imaju funkciju. Funkcija koja očekuje funkciju tu će funkciju primeniti negde u okviru svog tela. Na primer:

- $\lambda x.\, (x\ 2) + 1$ - x primenjujemo na dvojku, a onda dodamo broj 1
- $(\lambda x.\, (x\ 2) + 1)(\lambda x.\, x) \rightarrow (\lambda x.\, x)2 + 1 \rightarrow 3$
- $(\lambda x.\, (x\ 2) + 1)(\lambda x.\, x + 3) \rightarrow (\lambda x.\, x + 3)2 + 1 \rightarrow 2 + 3 + 1 \rightarrow 6$

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

- $\lambda x.\, (\lambda y.2 \cdot y + x)$
- $(\lambda x.\, (\lambda y.2 \cdot y + x))5 \rightarrow \lambda y.2 \cdot y + 5$ - primenimo 5 i dobijamo funkciju kao povratnu vrednost. Zbog česte upotrebe uvedena je skraćenica pa možemo pisati $\lambda xy.2 \cdot y + x$.

Lambda izrazi su ograničeni samo na jedan argument pa je pitanje kako definisati funkcije sa više argumenata. Dokazano je da bilo koja funkcija sa više argumenata može da se definiše pomoću funkcije sa samo jednim argumentom. Taj postupak naziva se **Karijev postupak**, po matematičaru Haskelu Kariju. Ideja je da funkcija koja treba da uzme dva argumenta prvo uzme jedan argument, od koga se pravi funkcija koja će onda uzeti drugi argument. Na primer:

- $f(x,\ y) = x + y$ definišemo kao $\lambda xy.\, x + y$
- $((\lambda xy.\, x + y)2)6 \rightarrow \lambda y.\, (2 + y)6 \rightarrow 2 + 6 \rightarrow 8$

Dakle, **Karijeve funkcije** argumente uzimaju jedan po jedan. One se mogu delimično evaluirati tako da se definišu nove funkcije kojima su neki argumenti početne funkcije fiksirani. Delimičnom evaluacijom fiksiraju se levi argumenti funkcije. Na primer, u Haskellu se izrazom $(max\ 3)\ 10$ prvo dobija funkcija kojoj je levi argument fiksiran na 3, a potom se takva funkcija primenjuje na 10.

Višestrukom beta redukcijom izražunavamo vrednost izraza i zaustavljamo se tek onda kada dalja beta redukcija nije moguća. Tako dobijen lambda izraz naziva se **normalni oblik** i on intuitivno odgovara vrednosti polaznog izraza. Izvesti normalni oblik primenom odgovarajućih redukcija na termove:

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

- $(\lambda kmn.\, k - m + n)10)5 \to (\lambda mn.10 - m + n)5 \to \lambda n.\, 10 - 5 + n \to \lambda n.\, n + 5$
- $(\lambda k.\, k \cdot k + 1)((\lambda m.\, m + 1)2) \to (\lambda k.\, k \cdot k + 1)3 \to 3 \cdot 3 + 1 \to 10$
- $(\lambda k.\, k\, 4)(\lambda y.\, y - 2) \to (\lambda y.\, y - 2)4 \to 4 - 2 \to 2$
- $((\lambda kmn.\, k \cdot m + n)2)3 \to (\lambda mn.\, 2 \cdot m + n)3 \to \lambda n.\, 2 \cdot 3 + n \to \lambda n.\, n + 6$
- dodatni primeri i rešenja

Videli smo da neki izrazi, kao što je $(\lambda x.\, x\, x)(\lambda x.\, x\, x)$, nemaju svoj normalni oblik. Za neke izraze mogu postojati i različite mogućnosti primene beta redukcije. Postavlja se pitanje da li je svejedno kojim ćemo putem krenuti. **Church-Rosser teorema** tvrdi da ako se lambda izraz može svesti na dva različita lambda izraza M i N, onda postoji treći izraz Z do kojeg se može doći i iz M i iz N. Kao posledica teoreme, jasno je da svaki lambda izraz ima najviše jedan normalni oblik, odnosno on ili ne postoji ili je jedinstven. Postoje dva poretka izvođenja redukcija. **Aplikativni poredak** podrazumeva da prvo izračunavamo vrednost argumenta i tek onda ga šaljemo u funkciju, što odgovara **pozivu po vrednosti** (**call-by-value**). **Normalni poredak** podrazumeva da beta redukcijom uvek redukujemo najlevlji izraz, što odgovara **evaluaciji po imenu** (**call-by-name**) ili **evaluaciji po potrebi** (**call-by-need**). Na primer:

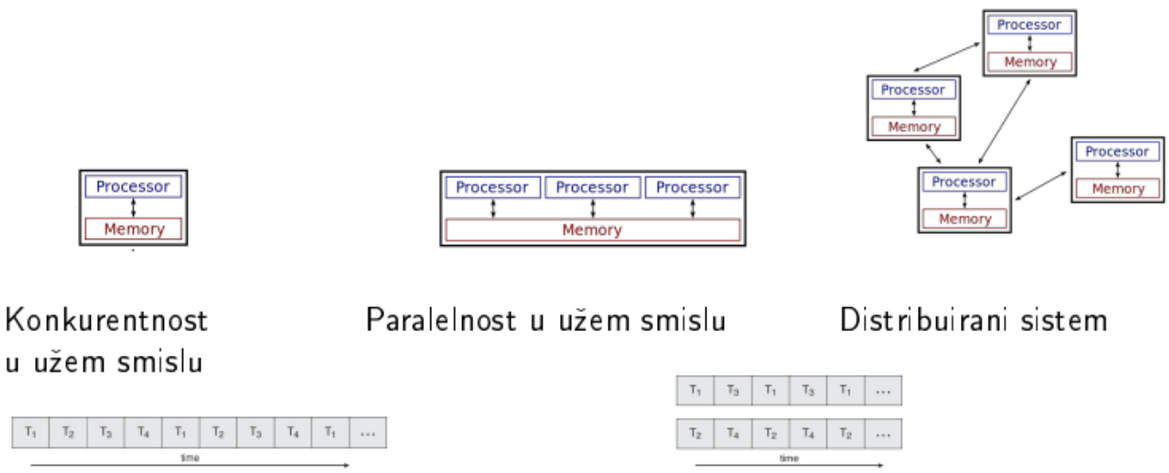
- aplikativni: $(\lambda x.5 \cdot x)(2 + 1) \to (\lambda x.\, 5 \cdot x)3 \to 5 \cdot 3 \to 15$
- normalni: $(\lambda x.5 \cdot x)(2 + 1) \to 5 \cdot (2 + 1) \to 5 \cdot 3 \to 15$

Teorema standardizacije tvrdi da ako je Z normalni oblik izraza E, onda postoji niz redukcija u normalnom poretku koji vodi od E do Z. Dakle, normalni poredak, tj. lenja evaluacija, garantuje završetak izračunavanja uvek kada je to moguće. Na primer, izraz $(\lambda x.1)((\lambda x.\, x\, x)(\lambda x.\, x\, x))$ se normalnim poretkom evaluira u 1, a aplikativnim poretkom se ne završava. Normalni poredak odgovara nestriktnoj semantici i omogućava korišćenje beskonačnih struktura. Kod normalnog poretka se lenjom evaluacijom izbegavaju nepotrebna izračunavanja, jer se svi izrazi evaluiraju tek kada su potrebni.

Konkurentno programiranje

Uvod u konkurentno programiranje

Konkurentnu paradigmu karakteriše više procesa koji se izvršavaju u istom vremenskom periodu, a koji imaju zajednički cilj. Veliki deo teorijskih osnova datira iz 60tih godina, a već Algol 68 sadrži podršku za konkurentno programiranje. Uzroci za široko rasprostranjen interes za konkurentno programiranje poslednjih godina su podrška logičkoj strukturi problema, dobijanje na brzini i rad sa fizički nezavisnim uređajima. Konkurentni mehanizmi su originalno izmišljeni za rešavanje problema u okviru operativnih sistema, ali se sada koriste u raznim aplikacijama jer mnogi programi moraju da vode računa o više nego jednom zadatku koji su u velikoj meri nezavisni. Primer aplikacije čiji se dizajn oslanja na konkurentnost je veb pregledač. **Konkurentnost u užem smislu** podrazumeva konkurentnost koja obuhvata jedan procesor. Ukoliko imamo više procesora, to odgovara **paralelnom programiranju**. Može se odnositi na više procesora na različitim mašinama, ali se najčešće misli na višeprocesorku mašinu, tj. procesi međusobno komuniciraju preko zajedničke memorije. Aplikacije koje rade distribuirano korišćenjem različitih mašina, bilo da su u pitanju lokalno povezane mašine ili Internet, predstavljaju **distribuirano programiranje**. Ovi sistemi pokreću nezavisne programe na svakom od umreženih procesora i koriste slanje poruka za komunikaciju. Komunikacija upravo predstavlja i najskuplji deo ove vrste programiranja, pa efikasni distribuirani algoritmi teže da smanje potrebu za komunikacijom. Primer je klijent-server arhitektura aplikacije, kao i www, online igre sa velikim brojem igrača, lokalizovani sistemi različitih komponenti kao što su automobil i avion, telekomunikacione mreže i peer-to-peer aplikacije koje se pokreću na mreži računara ravnomernom podelom zadataka na delove.



Postoje dve **vrste konkurentnosti** - **fizička** koja podrazumeva postojanje više procesora i **logička**. Sa stanovišta programera ove dve konkurentnosti su iste u kontekstu konkurentnog i paralelnog programiranja, dok su kod distribuiranog programiranja stvari malo složenije i podrazumevaju da je programer svestan da će se program izvršavati distribuirano. Postoje četiri osnovna **nivoa konkurentnosti**:

- **nivo instrukcije**
- **nivo naredbe**
- **nivo jedinica**
- **nivo programa**

Prvi i četvrti nivo ne utiču na dizajn programskog jezika jer se prvi odnosi na karakteristike hardvera i mogućnost kompajlera da te karakteristike iskoristi, a četvrti na mogućnost operativnog sistema da pokrene više nezavisnih programa istovremeno. **Skalabilnost** se odnosi na ubrzavanje izvršavanja porastom broja procesora. U razmatranju skalabilnosti mora se uzeti u obzir i priroda problema i njegova prirodna ograničenja. Idealna bi bila linearna skalabilnost, ali je ona retka. **Faktor ubrzanja paralelizacijom** je broj koji pokazuje koliko puta se program izvršavan na više procesora brže izvršava u odnosu na to kada se on izvršava na jednom procesoru. Dobija se kao količnik vremena sekvencijalnog izvršavanja i vremena paralelnog izvršavanja:

$$S = \frac{T_s}{T_p}$$

Amdalov zakon: ako je α procenat koda koji se ne može paralelizovati, onda je maksimalno ubrzanje paralelizacijom $\frac{1}{\alpha}$. Odnosno, mali delovi programa koji se ne mogu paralelizovati će ograničiti mogućnost ukupnog ubrzavanja paralelizacijom. Ako sa α označimo deo progama koji se mora izvršavati sekvencijalno, onda se ostatak $(1 - \alpha)$ može izvršiti paralelno korišćenjem N procesorskih jedinica, odnosno dobijamo:

$$S = \frac{T_s}{T_p} = \frac{T_s}{T_s \cdot (\alpha + \frac{1-\alpha}{N})} \rightarrow \frac{1}{\alpha},\ kada\ N \rightarrow \infty$$

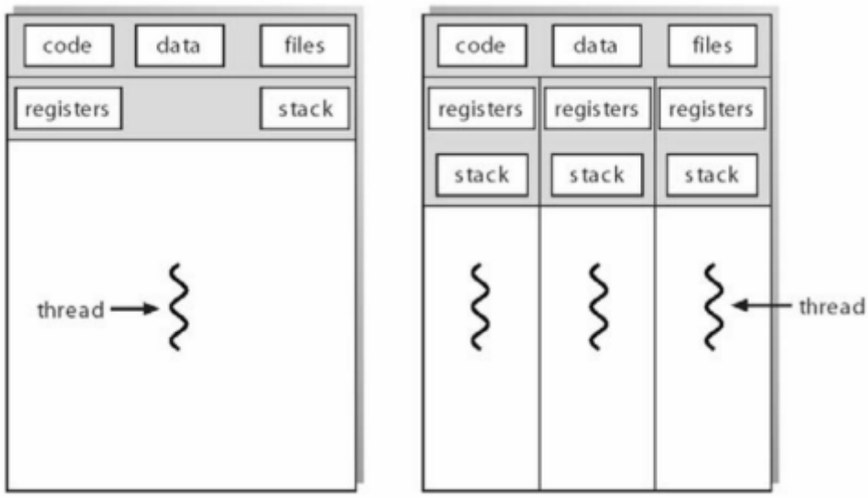
Dodatno treba uzeti u obzir i usporavanje koje nameće komunikacija između različitih niti izvršavanja. **Portabilnost** se odnosi na nezavisnost od konkretne arhitekture. Međutim, poznavanje ciljnog hardvera omogućava implementaciju efikasnijeg rešenja i često se uzima u obzir. Najveću razliku čine distribuirani sistemi u odnosu na sisteme sa deljenom memorijom. U okviru sistema sa deljenom memorijom to mogu biti simetrični multiprocesori i višejezgarni procesori, ali i grafički kartice. U okviru distribuiranih sistema to mogu biti mreža radnih stanica, mreže u opštem smislu, lokalne mreže i slično.

Osnovni koncepti

Zadatak ili **posao** (**task**) je jedinica programa, slična potprogramu, koja može da se izvrši konkurentno sa drugim jedinicama istog programa. Razlike u odnosu na potprogram:

- zadaci mogu da počinju implicitno, tj. ne moraju eksplicitno biti pozvani.
- kada program pokrene zadatak, ne mora da čeka na njegovo izvršavanje, već može da nastavi sa radom.
- kada se izvršavanje zadatka završi, kontrola ne mora da se vrati na mesto odakle je počela izvršavanje.

Zadaci mogu biti **teški** (heavyweight) i **laki** (lightweight). Teški zadaci imaju svoj sopstveni adresni prostor, dok laki dele isti. Teškim zadacima upravlja operativni sistem koji obezbeđuje deljenje procesorskog vremena, pristup datotekama i adresni prostor. Stanje teškog zadatka obuhvata podatke o izvršavanju, vrednosti registara, brojač instrukcija i informacije o upravljanju resursima kao što su informacije o memoriji, datotekama, ulazno-izlaznim zahtevima i ostalo. **Promena konteksta** je promena stanja procesora koja je neophodna kada se sa izvršavanja jednog teškog zadatka prelazi na izvršavanje drugog teškog zadatka, odnosno kada se u memoriji pamti stanje zadatka koji se prekida i na osnovu informacija u memoriji rekonstruiše drugi zadatak koji treba da nastavi izvršavanje. Promena konteksta je skupa i vremenski i memorijski, a dešava se veoma često - par stotina ili hiljada puta u sekundi. Koncept lakih zadataka uvodi se kako bi se omogućio efikasniji prelaz sa jednog na drugi zadatak. Kreiranje lakih zadataka je efikasnije od kreiranja teških. Oni ne zahtevaju posebne računarske resurse, već postoje unutar jednog teškog zadatka i samim tim podaci koji se vode na nivou teškog zadatka zajednički su za sve lake zadatke koje on obuhvata. Podaci o lakom zadatku obuhvataju samo njegovo stanje izvršavanja, brojač instrukcija i vrednosti radnih registara, pa se prelaz sa jednog lakog zadatka na drugi vrši brzo.



Terminologija je nekonzistentna kroz različite programske jezike. U C terminologiji teški zadaci nazivaju se **procesi**, a laki **niti**. Upravljanje nitima najčešće se ostvaruje preko korisničkih biblioteka i one su danas podržane u svim operativnim sistemima.

Paralelizacija zadataka je strategija koja podrazumeva da se koriste različite niti za svaki od glavnih programerskih zadataka ili funkcija. Ona dobro radi na malim mašinama, ali ne skalira dobro ukoliko imamo veliki broj procesora. Za dobro skaliranje na velikom broju procesora, potrebna je **paralelizacija podataka** koja podrazumeva da se iste operacije primenjuju konkurentno na elemente nekog velikog skupa podataka. Najčešće se zasniva na paralelizaciji petlji, odnosno predstavlja paralelizam na nivou naredbe. Pozivi funkcija koje se konkurentno izvršavaju moraju biti nezavisni. Paralelizacija podataka je prirodna za neke vrste problema, ali ne za sve. Podatke uglavnom delimo na n jednakih delova, ali nemamo uvek pretpostavku o uniformnosti podataka. Tada se vrši podela tako da svaka nit uzima jedan po jedan element kada završi sa prethodnim. U ovom slučaju je potrebna i komunikacija između niti, pa se nekada uzima 5 po 5 ili 10 po 10 elemenata kako bi se smanjila komunikacija. Primeri:

- množenje matrica i vektora
- množenje matrica
- proizvod po komponentama vektora
- manipulacija slikama
- pronalaženje brojeva koji imaju određenu osobinu i pripadaju intervalu [n, m]
- dodatni primeri

Komunikacija se odnosi na svaki mehanizam koji omogućava jednom zadatku da dobije informacije od drugog. Može se ostvariti preko **zajedničke memorije**, ako zadaci dele memoriju, i **razmenom poruka**. Razmena poruka može se upotrebljavati uvek i u tom slučaju jedan zadatak mora eksplicitno da pošaljem podatak drugom. Može se ostvariti na razne načine kao što su mehanizmi međuprocesne komunikacije na istoj mašini (tokovi, signali, cevi, soketi, kanali) i mreže u distribuiranim sistemima. Ukoliko je slanje poruka u okviru iste mašine, onda se ono smatra pouzdanim, dok kod distribuiranih sistema podaci mogu da se zagube dok putuju kroz mrežu. U distribuiranim sistemima se za slanje poruka koriste različiti protokoli. Ukoliko imamo zajedničku memoriju, promenljivama se može pristupiti iz različitih (lakih) zadataka. Potrebno je da jedan zadatak upiše vrednost promenljive, a drugi da je pročita. U ovom slučaju je važan redosled čitanja i pisanja promenljivih.

Sinhronizacija se odnosi na mehanizam koji omogućava programeru da kontroliše redosled u kojem se operacije dešavaju u okviru različitih zadataka. Ona je implicitna u modelu slanja poruka jer poruka prvo mora da se pošalje da bi mogla da se primi. Međutim, u modelu deljene memorije se može desiti da zadatak pročita staru vrednost promenljive, pre nego je ona zamenjena novom od strane drugog zadatka. Sinhronizacija može da se implementira na dva načina:

- **zauzeto čekanje -** zadatak u petlji stalno proverava da li je neki uslov ispunjen. Ovaj pristup 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.

Postoje dve vrste potrebe za sinhronizacijom kada zadaci dele memoriju:

- **saradnja -** kada zadatak A mora da čeka da zadatak B završi neku aktivnost pre njega, kako bi nastavio ili započeo svoje izvršavnje. Primer je odnos proizvođač-potrošač.

- **takmičenje** - oba zadatka zahtevaju isti resurs. Primer je korišćenje štampača. Na mašinskom nivou imamo tri koraka pri korišćenju resursa - uzimanje vrednosti, izmena i upisivanje nove vrednosti. Ako imamo sinhronizaciju mogući ishodi su da se prvo završi zadatak A, a zatim B ili obrnuto. Ako je nema, moguće je da oba zadatka istovremeno uzmu vrednost, ali jedan završi pre drugog i samim tim drugi prepiše vrednost koju je upisao drugi. Zadaci se takmiče da koriste deljene resurse i ponašanje programa zavisi od toga ko pobedi. Ova situacija naziva se **uslov takmičenja** ili **nadmetanje za resurse**.

Za kontrolisanje pristupa deljenim resursima koristi se **zaključavanje** ili **uzajamno isključivanje**. Za uzajamno isključivanje mogu se koristiti **semafori** ili **monitori**. Semafori imaju dve moguće operacije - P (wait) i V (release). Nit koju poziva P atomično umanjuje brojač i čeka da n postane nenegativan. Nit koju poziva V atomično uvećava brojač i budi čekajuću nit ukoliko postoji. Semafori se smatraju kontrolom niskog nivoa i lako dovode do grešaka. Bili su prisutni već u jeziku Algol 68. Monitori enkapsuliraju deljene strukture podataka sa njihovim operacijama, tj. čine deljene podatke apstraktnim tipovima podataka sa specijalnim ograničenjima. Takođe se smatraju kontrolom niskog nivoa, a prisutni su u jezicima kao što su Java i C#. **Katanci** su mehanizmi koji zaključavaju resurse i omogućavaju bezbedno deljenje podataka u konkurentnom okruženju. Postoje različiti pristupi - na primer više niti može da čita istovremeno resurs, ali samo jedan može da piše i tada niko ne sme da čita. **Mutex** (mutual exclusion) je katanac koji podrazumeva da samo jedna nit u jednom trenutku može da koristi resurs. Zaključavanje je veoma skupa operacija. Ako postoji veliki broj katanaca i ako se često vrši zaključavanje i otključavanje to može da uspori rad aplikacije. Sa druge strane, ako postoji mali broj katanaca i ako se zaključava velika količina podataka istovremeno to može dovesti do dužeg čekanja i ponovo sporijeg rada aplikacije.

|Kod sekvencionalnog izvršavanja programa, program ima karakteristiku **napredovanja** (liveness) ukoliko nastavlja sa izvršavanjem, dovodeći do završetka rada u nekom trenutku, ako se program zaustavlja. Odnosno, ako neki događaj treba da se desi, on će se i desiti u nekom trenutku, tj. stalno se pravi nekakav progres.| U konkurentnoj sredini sa deljenim objektima, može se desiti da program nikada ne završi svoj rad. |Pored pogrešnog izračunavanja, ako nemamo dobru sinhronizaciju mogući su i sledeći problemi:

- **uzajamno** (**smrtonosno**) **blokiranje** (**deadlock**) - zadaci A i B zahtevaju resurse X i Y da bi mogli završiti posao. Zadatak A uzme resurs X, a zadatak B uzme resurs Y i onda oba zadatka čekaju na drugi resurs, pa program ne može normalno da završi sa radom.
- **živo blokiranje** (**livelock**) - svi zadaci nešto rade, ali opet nema progresa. Zadaci A i B vraćaju resurse X i Y i uzimaju onaj drugi, odnosno zamenjuju ih i tako u nedogled.
- **individualno izgladnjivanje** (**lockout**) - jedan ili više zadataka sprečavaju izvršavanje nekog zadatka. Pretpostavimo da zadatak A koristi samo resurs X, zadatak B samo resurs Y, a zadatak C oba. Može se desiti da resursi A i B non-stop uzimaju resurse i koriste ih, dok zadatak C nikad ne može da dobije oba resursa kako bi završio svoj rad.|

Potprogrami i klase imaju različite oznake koje opisuju njihovu bezbednost za korišćenje u konkurentnom okruženju:

- **jedinstveno pozivanje** - ne sme se istovremeno upotrebljavati u različitim nitima.
- **ulazne** (**reentrant**) - može se upotrebljavati na različitim nitima uz dodatne pretpostavke.
- **bezbedne po niti** (**thread-safe**) - sme se bezbedno upotrebljavati bez ograničenja.

Primer jednog modela razmene poruka je **Actor**. Podrazumeva **SN** (Share Nothing) **arhitekturu** gde se podaci između čvorova sistema međusobno ne dele. Svaka komponenta ima svoje stanje koje može da menja, ali ga nikada ne deli sa drugima. Komponente predstavljaju objekti koji se nazivaju **aktori**. Oni formiraju hijerarhijsku strukturu gde svaki ima svoje **poštansko sanduče** i njegov zadatak je da obradi svaku poruku koju dobije. Poruke koje stižu čuvaju se u redu i obrađuju redom koje su stigle. Kao odgovor na poruku, aktor može da promeni svoje stanje, napravi još novih aktora (koji su njegova deca), šalje poruke drugim aktorima i zaustavi svoj rad ili rad svoje dece. Kada pošalje poruku, aktor dalje nastavlja sa radom. Pogodan je u aplikacijama gde je moguće rasporediti posao na veliki broj manjih, nezavisnih poslova i tada svaki posao predstavlja jedan aktor. Roditelji prikupljaju rezultate izvršavanja svoje dece. Može se koristiti i na distribuiranim sistemima, ali nije ogodan ukoliko je potrebno intenzivno deljenje podataka kao u slučaju rada sa bazama podataka i transakcijama. Model Actor enkapsulira rad sa nitima čime se nalazi na višem nivou apstrakcije koja omogućava udobniji rad i manje grešaka, ali to sa druge strane povlači slabije performanse.

Distribuirani sistemi

Distribuirani sistem je sistem koji se sastoji od više komponenti koje su locirane na različitim mašinama i koje komuniciraju i koordiniraju sa ciljem da izgledaju kao jedan koherentni sistem krajnjem korisniku. Te komponente mogu biti sve mašine koje imaju mogućnost da se zakače na mrežu, imaju svoju lokalnu memoriju i mogu da komuniciraju putem slanja poruka. Osnovne karakteristike ovih sistema su to što sve komponente rade konkurentno, ne postoji globalni časovnik, tj. sinhronizacija mora softverski da se obezbedi i sve komponente mogu da prestanu sa radom neočekivano i nezavisno jedna od druge. Osnovni pristupi funkcionisanja distribuiranih sistema:

- mašine rade zajedno oko zajedničkog cilja i korisnik vidi rezultat kao jedinstven.
- svaka mašina ima svog korisnika i distribuirani sistem omogućava deljenje resursa ili komunikaciju mašine i korisnika.

Kada sistem raste i kada se zahtevi sistemu povećavaju, potrebno je unaprediti hardver. Postoje dva tipa unapređivanja:

- **vertikalno skaliranje** - poboljšavanje karakteristika hardvera jedne mašine. Dobro je dok je moguće, ali nakon određene tačke i najbolji hardver nije dovoljno dobar za srednju i veliku količinu komunikacije i izračunavanja.
- **horizontalno skaliranje** - dodavanje više računara. Ovde ne postoji gornji limit skaliranja, jer uvek možemo dodati nove računare.

Prednosti distribuiranog programiranja:

- horizontalna skalabilnost - u opštem slučaju dodavanje novih računara nije skupo.
- pouzdanost - ne postoje posledice ako jedna mašina otkaže, jer su sistemi razvijeni tako da budu tolerantni na padove.
- performanse - posao se deli na veći broj mašina i nekad se rešavaju zadaci koje nije moguće rešiti na jednoj mašini.

Izazovi distribuiranog programiranja:

- komunikacija - brzina kojom paket putuje kroz mrežu je ograničena brzinom svetlosti, pa velika količina komunikacije može predstavljati izazov.
- kompleksan dizajn, konstrukcija i debagovanje aplikacije
- raspoređivanje poslova
- latentnost (kašnjenje) - što je sistem šire distribuiran, to je kašnjenje zbog komunikacije veće.
- posmatranje - padovi su neizbežni i potrebno je ispratiti ih kako bi se razumela njihova priroda i kako bi se napravile odgovarajuće reakcije i prevencija.

Distribuirani sistemi mogu imati **monolitnu** i **mikroservisnu arhitekturu**. Mikroservisna arhitektura uvodi kompleksnost u razne delove razvoja, ali ima veliki broj benefita kao što su lakša podela poslova, veća skalabilnost, pouzdanost i otpornost na padove. Monolitne aplikacije mogu imati sledeće strukture:

- **klijent-server** - klijenti kontaktiraju server za podatke i onda dobijene podatke formatiraju i prikazuju krajnjem korisniku.
- **troslojna arhitektura** (three-tier) - informacije o klijentu su sačuvane u srednjem sloju umesto na klijentu, kako bi se pojednostavio razvoj i stavljanje u rad aplikacije. Često se koristi kod veb aplikacija.
- **višeslojna arhitektura** (n-tier) - koristi se kada server ima potrebu da prosledi zahtev dodatnim servisima na mreži.
- **ravnopravni učesnici** (peer-to-peer) - odgovornost je uniformno distribuirana između mašina sistema i ne postoje dodatne mašine koje se koriste.

Distribuirana skladišta podataka podrazumevaju da su podaci podeljeni i skladišteni na više mašina ili da su duplirani u celosti na nekoliko mašina. **CAP teorema** iz 2002. godine tvrdi da u okvori distribuiranog sistema nije moguća istovremena prisutnost naredne tri osobine:

- **konzistentnost** (**C**onsistency) - ono što se upiše i pročita redom je ono što se i očekuje.
- **dostupnost** (**A**vailability) - ceo sistem nikada ne umire, tj. svaki čvor sistema koji nije otkazao vraća odgovor.
- **tolerantnost na razdvajanje** (**P**artition tolerant) - sistem nastavlja da radi i zadržava zadate garancije konzistentnost/dostupnosti uprkos mogućim razdvajanjima odnosno prekidima u komunikaciji u okviru mreže.

Razdvojenost dva čvora je nešto što se ne može sprečiti i u tom slučaju treba izabrati između jake konzistentnosti i dostupnosti. Ako se prekina veza između dva čvora, ne postoji način da jedan čvor zna šta drugi radi. U tom slučaju on može da prekine sa radom i postane nedostupan ili da nastavim sa radom, ali u tom slučaju ne garantuje konzistentnost. Praksa pokazuje da većina aplikacija više vrednuje dostupnost i da stroga konzistentnost nije ni neophodna. Između ostalog, razlog za to je što stroga konzistentnost zahteva opštu sinhronizaciju mašina što utiče na performanse.

Nedistribuirane baze podataka zahtevaju **ACID osobine**:

- **A**tomicity (atomičnost) - garantuje da će svaka transakcija biti tretirana kao jedinstvena jedinica koja ili uspeva kompletno ili kompletno ne uspeva.
- **C**onsistency (konzistentnost) - obezbeđuje da svaka transakcija može da prevede bazu podataka iz jednog ispravnog u drugo ispravno stanje.

- **I**solation (izolovanost) - obezbeđuje da će svako konkurentno izvršavanje transakcija ostaviti bazu podataka u istom stanju kao što bi to bila da su se transakcije izvršavale sekvencijalno.
- **D**urability (trajnost) - garantuje da će jednom urađena transakcija biti trajna, tj. da će njeni rezultati biti pristuni čak i ukoliko se desi pad sistema.

Distribuirane baze podataka zahtevaju **BASE osobine**:

- **B**asically **A**vailable - sistem uvek vraća odgovor.
- **S**oft state - sistem može da se menja vremenom, čak i kada nema novih ulaza zbog postizanja konzistentnosti.
- **E**ventual consistency - u odsustvu novih ulaza, podaci će se raširiti po svim čvorovima pre ili kasnije i distribuirani sistem će postati konzistentan.

Distribuirano izračunavanje je tehnika podele velikog zadatka, koji ni jedan pojedinačni računar ne može praktično da izvrši, u puno manjih zadataka, tako da svaki manji može da se izvrši na pojedinačnoj mašini. Ovakav pristup omogućava horizontalnu skalabilnost, odnosno za veći zadatak potrebno je uključiti veći broj čvorova u izračunavanje. Google je razvio paradigmu **MapReduce** koja se bavi ovim izračunavanjem. Distribuirano izračunavanje može da koristi i podelu podataka i podelu zadataka. Na primer, platforma za elektronsko poslovanje može da koristi podelu zadataka.

Distribuirano slanje/primanje poruka zasniva se na konceptu **pouzdanog reda poruka**. Poruke se smeštaju u red asinhrono između aplikacija klijenata i sistema za poruke. Sloj koji je odgovoran za slanje i primanje poruka naziva se **Message Oriented Middleware** (**MOM**). Distribuirani sistem za poruke obezbeđuje pouzdanost, skalabilnost i trajnost. Postoje dve vrste sistema:

- **Point To Point (PTP) MOM** - paradigma gde se komunikacija vrši **jedan-na-jedan**, odnosno pošiljaoc proizvodi poruku za tačno jednog primaoca.
- **Publisher Subscriber (Pub/Sub) MOM** - paradigma gde pošiljaoc šalje poruku svima koji su se za to pretplatili, tj. paradigma **jedan-prema-puno**. Glavno obeležje je **tema** poruke na osnovu koje se primaoci mogu pretplatiti. Jednom kada pošiljaoc pošalje poruku, pretplatioci mogu da prime odabrane poruke uz pomoć opcije **filtriranja**. Postoji **filtriranje zasnovano na temi** i **filtriranje zasnovano na sadržaju**. Najpoznatiji alat ovog tipa je Apache Kafka.

Postoje različiti **obrasci za zahtev/odgovor** i zavise od konkretne arhitekture. Na primer, kod klijent-server arhitekture postoje obrasci:

- **pošalji i zaboravi** (Fire and Forget) - klijent šalje poruku serveru i ne očekuje da server pošalje odgovor nazad. Klijent šalje poruku kada njemu to odgovara.
- **zahtev za odgovorom** (Request Response) - klijent šalje poruku i očekuje da mu server odgovori sa tačno jednim odgovorom. Klijent ne šalje poruku dok nije spreman da prihvati odgovor.
- **zahtev za tokom** (Request Stream) - klijent šalje poruku i očekuje da mu server odgovori u obliku N poruka. Klijent ne šalje poruku dok nije spreman da prihvati odgovore, a server ne šalje odgovore pre nego što klijent potvrdi da očekuje N odgovora.
- **zahtev za kanalom** (Request Channel) - klijent i server se pripreme da emituju i primaju N poruka jedan od drugog. Ni klijent ni server ne počinju slanje poruka dok ne dobiju signal od ovog drugog da su spremni da prime i obrade N poruka.

Logičko programiranje

Uvod u logičko programiranje

Proces izračunavanja kod **logičke paradigme** zasniva se na različitim logičkim metodama. Formalizmi za imperativne i funkcionalne jezike su redom Tjuringova mašina i lambda račun i ti jezici imaju svu izražajnost svojih formalizama. Formalizam logičke paradigme je **logika prvog reda**, koja nije formalizam izračunljivosti pa se ne može vršiti poređenje ove vrste. Ipak, većina logičkih jezika je Tjuring-kompletno. Logika se koristi kao **deklarativni jezik** za opisivanje problema, a **dokazivač teorema** kao mehanizam za njihovo rešavanje. Najznačajniji predstavnik logičke paradigme je **Prolog**, koji se koristi kao sinonim za ovu paradigmu. Sintaksa svih logičkih jezika je dosta slična, dok se dokazivač teorema može razlikovati. Prolog za rešavanje problema koristi **metod rezolucije**. Prvo nastaje kao **ABSYS** 1967. godine i koristi se za rad sa tvrđenjima preko kojih se dedutkivnim putem generiše odgovor na postavljena pitanja. Kasnije je preimenovan u Prolog, a kreiran je i prvi kompajler. U početku se radilo na intenzivnom razvoju jezika, ali sada je fokus na integraciji sa drugim programskim jezicima i danas postoje razne implementacije Prologa. Drugi značajni predstavnici su **ASP** i **Datalog**. ASP koristi rešavače za iskaznu logiku i koristi se za pretragu

kod NP teških problema kao što su bojenje grafova, hamiltonovi ciklusi i slično. Datalog je sličan Prologu i koristi se za rad sa bazama podataka. Koristi razne efikasne algoritme za rešavanje upita, pa nije Tjuring-kompletan. Logičko programiranje je pogodno za rešavanje problema matematičke logike, veštačke inteligencije, obradu prirodnih jezika, automatizaciju projektovanja, simboličko rešavanje jednačina i slično. Sa druge strane, nije pogodno za I/O algoritme, numeričke algoritme, grafiku i ostalo.

Logika prvog reda

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** sastoji se od najviše prebrojivih skupova Σ i Π koje redom nazivamo skupom **funkcijskih simbola** i skupom **predikatskih simbola**, kao i od funkcije 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 Σ i Π čine nelogički deo jezika prvog reda, a sve njihove elemente zovemo **nelogičkim simbolima**. Skup **termova** nad signaturom L = (Σ, Π, ar) i skupom promenljivih V je skup za koji važi:

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

Termovi se mogu dobiti samo konačnom primenom prethodnih pravila. Skup **atomičkih formula** nad signaturom L = (Σ, Π, 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, ..., t_n su termovi, onda je i $p(t_1, t_2, \ldots, t_n)$ atomička formula.

Skup **dobro zasnovanih formula** nad signaturom L = (Σ, Π, 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 formule.
- ako je A dobro zasnovana formula i x je promenljiva, onda su $(\forall x)A$ i $(\exists x)A$ dobro zasnovane formule.

Dobro zasnovane formule mogu se dobiti samo konačnom primenom prethodnih pravila. **Literal** je atomička formula ili negacija atomičke formule. **Klauza** je disjunkcija literala. **Izrazi** su termovi i dobro zasnovane formule.

Term dobijen 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 promenljiva:
  - ako je t = x, onda je $t[x \mapsto t_x]$ = t_x
  - ako je t = y, $y \neq x$, onda je $t[x \mapsto t_x]$ = t
- ako je t = $f(t_1, t_2, \ldots, t_n)$, onda je $t[x \mapsto t_x]$ = $f(t_1[x \mapsto t_x], t_2[x \mapsto t_x], \ldots, t_n[x \mapsto t_x])$

Formulu dobijenu supstitucijom promenljive x termom t_x u formuli A označavamo sa $A[x \mapsto t_x]$ i definišemo na sledeći način:

- ako je A atomička formula:
  - $\top[x \mapsto t_x]$ = $\top$
  - $\bot[x \mapsto t_x]$ = $\bot$
  - ako je A = $p(t_1, t_2, \ldots, t_n)$, onda je $A[x \mapsto t_x]$ = $p(t_1[x \mapsto t_x], t_2[x \mapsto t_x], \ldots, t_n[x \mapsto t_x])$
- $(\neg A)[x \mapsto t_x]$ = $\neg(A[x \mapsto t_x])$
- $(A \wedge B)[x \mapsto t_x]$ = $(A[x \mapsto t_x] \wedge B[x \mapsto t_x])$
- $(A \vee B)[x \mapsto t_x]$ = $(A[x \mapsto t_x] \vee B[x \mapsto t_x])$
- $(A \Rightarrow B)[x \mapsto t_x]$ = $(A[x \mapsto t_x] \Rightarrow B[x \mapsto t_x])$
- $(A \Leftrightarrow B)[x \mapsto t_x]$ = $(A[x \mapsto t_x] \Leftrightarrow B[x \mapsto t_x])$

- $(\forall x A)[x \mapsto t_x]$ = $(\forall x A)$
- $(\exists x A)[x \mapsto t_x]$ = $(\exists x A)$
- ako je $x \neq y$, neka je z promenljiva koja se ne pojavljuje ni u $(\forall y)A$ ni u t_x, tada je $(\forall y A)[x \mapsto t_x]$ = $(\forall z)A[y \mapsto z][x \mapsto t_x]$
- ako je $x \neq y$, neka je z promenljiva koja se ne pojavljuje ni u $(\exists y)A$ ni u t_x, tada je $(\exists y A)[x \mapsto t_x]$ = $(\exists z)A[y \mapsto z][x \mapsto t_x]$

Treba primetiti da supstitucija preslikava promenljivu u term, pa konstante, funkcijski simboli i funkcije ostaju nepromenjeni. **Uopštena supstitucija** σ je skup zamena $[x_1 \mapsto t_1]$, $[x_2 \mapsto t_2]$, ..., $[x_n \mapsto t_n]$, gde su x_i promenljive i t_i su proizvoljni termovi i $x_i \neq x_j$ za $i \neq j$. Takvu zamenu kraće zapisujemo kao $[x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n]$. Uopštena zamena primenjuje se simultano na sva pojavljivanja promenljivih $x_1, x_2, \ldots, x_n$ u polaznom izrazu i samo na njih, odnosno ne primenjuje se na podtermove dobijene zamenama. Izraz koji je rezultat primene zamene σ nad izrazom E označavamo sa $E\sigma$. Primeri:

- 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
  $t\,\sigma = g(a, g(f(y), a))$.
- Za $\sigma = [x \mapsto ana]$ i $s = igra(x, sah)$ važi
  $s\,\sigma = igra(ana, sah)$.
- Za $\sigma = [x \mapsto panda, y \mapsto bambus]$ i $s = jede(x, y)$ važi
  $s\,\sigma = jede(panda, bambus)$.
- Za $\sigma = [x \mapsto zivotinja(x), y \mapsto bambus, z \mapsto jabuka]$,
  $t = hrana(x, y, z)$ važi
  $t\,\sigma = hrana(zivotinja(x), bambus, jabuka)$.

Problem **unifikacije** je problem ispitivanja da li postoji supstitucija koja čini dva izraza jednakim. Ako su e_1 i e_2 izrazi i ako postoji supstituacija σ takva da važi $e_1\sigma = e_2\sigma$, onda kažemo da su ovi izrazi **unifikabilni**, a da je ta supstitucija **unifikator** za njih. Dva unifikabilna izraza mogu da imaju više unifikatora. |Primeri:

- Neka je term t_1 jednak $igra(x, y)$,
  neka je term t_2 jednak $igra(ana, z)$
  i neka je σ supstitucija $[x \mapsto ana, z \mapsto y]$.
  Tada je i $t_1\sigma$ i $t_2\sigma$ jednako $igra(ana, y)$, 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 $[x \mapsto ana, y \mapsto sah, z \mapsto sah]$.
- Termovi $igra(x, x)$ i $igra(ana, sah)$ nisu unifikabilni.
- 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.

- dodatni primeri|

Među svim unifikatorima za dva izraza postoji jedan koji je najopštiji, tj. svi drugi se mogu dobiti iz njega primenom dodatne supstitucije. Na primer, ako imamo termove $f(x)$ i $f(y)$ i supstitucije $[x \mapsto a, y \mapsto a]$, $[x \mapsto b, y \mapsto b]$ i $[x \mapsto y]$, treća je **najopštiji unifikator**. Postoji algoitam koji efikasno pronalazi najopštiji unifikator ili vraća neuspeh ukoliko unifikator za date izraze ne postoji.

Metod rezolucije je metod za izvođenje zaključaka u logici prvog reda koji se zasniva na **pravilu rezolucije**: $\frac{A \vee B, \neg B \vee C}{A \vee C}$. Na primer, posledica formula $A \Rightarrow B$ i $B \Rightarrow C$ je formula $A \Rightarrow C$. Pravilo rezolucije iskazuje se u terminima klauza, tj. umesto $A \Rightarrow B$ koristi se $\neg A \vee C$. |Primeri:

- S = Vreme je sunčano
- K = Kampovanje je zabavno
- D = Domaći zadatak je urađen
- M = Matematika je laka

Prevesti narednu formulu u najpribližniji prirodni iskaz $(M \Rightarrow D) \wedge (S \Rightarrow K)$
Ako je matematika laka onda je domaći zadatak urađen i ako je vreme sunčano onda je kampovanje zabavno.
ili
Domaći zadatak je urađen kada je matematika laka i kampovanje je zabavno kada je vreme sunčano.

Matematika je laka ili je kampovanje zabavno, sve dok je vreme sunčano i domaći zadatak urađen.
Ako je vreme sunčano i domaći zadatak urađen, onda je matematika laka ili je kampovanje zabavno.
$(S \wedge D) \Rightarrow (M \vee K)$

Ako pada kiša, Ivan ponese kišobran. Ako Ivanima kišobran, on nije mokar. Ako ne pada kiša, Ivannije mokar. Dokaži da Ivan nije mokar.

Pada kiša r (*rain*) Ivan ponese kišobran u (*umbrella*) Ivan je mokar w (*wet*)

- Ako pada kiša, Ivan ponese kišobran.
  $(r \Rightarrow u)$ KNF: $\neg r \vee u$
- Ako Ivan ima kišobran, on nije mokar.
  $(u \Rightarrow \neg w)$ KNF: $\neg u \vee \neg w$
- Ako ne pada kiša, Ivan nije mokar.
  $(\neg r \Rightarrow \neg w)$ KNF: $r \vee \neg w$
- Dokaži da Ivan nije mokar.
  $(\neg w)$

---

1. $\neg r \vee u$ (premisa)
2. $\neg u \vee \neg w$ (premisa)
3. $r \vee \neg w$ (premisa)
4. $\neg r \vee \neg w$ (L1, L2, rezolucija)
5. $\neg w \vee \neg w$ (L3, L4, rezolucija)
6. $\neg w$ (L5, idempotencija)
7. QED

Obično se dokaz izvodi dobijanjem kontradikcije tako što se dodaje negacija uslova koji želimo da dokažemo. U poslednjem primeru dodaje se izraz w.

Jelena prisustvuje sastanku ili Jelena nije pozvana na sastanak. Ako šef želi da Jelena bude na sastanku, on će je pozvati. Jelena nije prisustvovala sastanku. Ako šef nije želeo da Jelena bude na sastanku, i ako je šef nije pozvao, onda će ona biti unapređena. Dokazati da će Jelena biti unapređena.

S (Jelena prisustvuje **S**astanku.) P (Jelena je **P**ozvana.)
Z (Šef **Ž**eli da Jelena bude na sastanku.) U (Jelena će biti **U**napređena.)

1. Jelena prisustvuje sastanku ili Jelena nije pozvana na sastanak.
   $S \vee \neg P$
2. Ako šef želi da Jelena bude na sastanku, on će je pozvati.
   $Z \Rightarrow P$ KNF: $\neg Z \vee P$
3. Jelena nije prisustvovala sastanku.
   $\neg S$
4. Ako šef nije želeo da Jelena bude na sastanku, i ako je šef nije pozvao, onda će ona biti unapređena.
   $(\neg Z \wedge \neg P) \Rightarrow U$ KNF: $Z \vee P \vee U$
5. Dokazati da će Jelena biti unapređena.
   U

---

1. $S \vee \neg P$ (premisa)
2. $\neg Z \vee P$ (premisa)
3. $\neg S$ (premisa)
4. $Z \vee P \vee U$ (premisa)
5. $\neg U$ (negacija zaključka)
6. $Z \vee P$ (L4, L5, rezolucija)
7. P (L2, L6, rezolucija, idempotencija)
8. S (L1, L7, rezolucija)
9. $\bot$ (L3, L8, rezolucija)

Prethodna primena pravila rezolucije odgovara primeni rezolucije u iskaznoj logici. U logici prvog reda koristi se klauzalna forma. Ako imamo izraze $A \Rightarrow B'$ i $B'' \Rightarrow C$ i ako su B' i B'' unifikabilni, onda se oni mogu učiniti jednakim nekim unifikatorom σ pa iz formula $A\sigma \Rightarrow B'\sigma$ i $B''\sigma \Rightarrow C\sigma$ dobijamo posledicu $A\sigma \Rightarrow C\sigma$. Primeri:

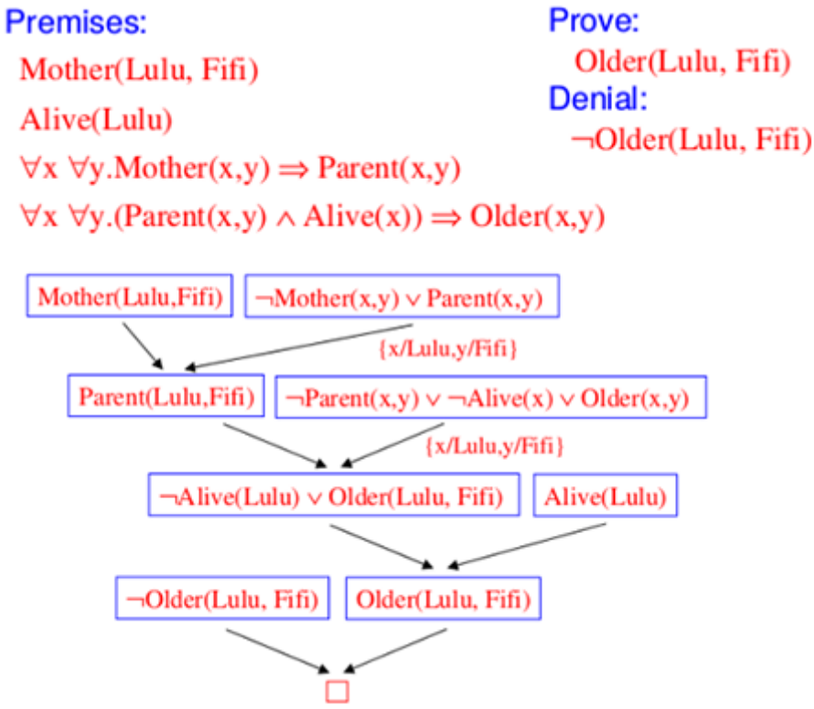
- Voli(x,y) znači "x voli y"
- Putnik(x) znači "x je putnik"
- Grad(x) znači "x je grad"
- Živi(x,y) znači "x živi u y"

Prevesti:

$$\exists x \forall y \forall z (Grad(x) \wedge Putnik(y) \wedge Zivi(z,x)) \Rightarrow (Voli(y,x) \wedge \neg Voli(z,x))$$

Postoji grad koji svi putnici vole ali svako ko živi u njemu ga ne voli.
Neki gradovi su voljeni od svih putnika ali ne i od osoba koje žive u njemu.
Prevesti u predikatsku logiku:
Svaki putnik ne voli grad u kojem živi.

$$\forall x \forall y ((Putnik(x) \wedge Grad(y) \wedge Zivi(x,y)) \Rightarrow \neg Voli(x,y))$$



- dodatni primer
- dodatni primer

Prolog

Prolog se oslanja na logiku prvog reda i metod rezolucije, pa je potrebno njihovo dobro poznavanje. Zapis je izveden iz logike prvog reda, ali je redukovan i moguće je zapisati samo **Hronove klauze** - disjunkcije literala sa najviše jednim nenegiranim literalom. To su implikacije oblika $(A_1 \wedge A_2 \wedge \ldots \wedge A_n) \Rightarrow B$, odnosno $\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n \vee B$. Sa druge strane, nije moguće zapisati tvrđenja oblika $\neg A \Rightarrow B$. Prilikom traženja rešenja u Prologu koriste se unifikacija i supstitucija. Programiranje u Prologu sastoji se od:

- definisanja činjenica o objektima i odnosima među njima.
- definisanja pravila o objektima i odnosima mešu njima.
- formiranja upita o objektima i odnosima među njima

Termovi su osnovni gradivni elementi Prologa i širi su pojam od terma logike prvog reda. U njih spadaju **konstante**, **promenljive** i **strukture**. Promenljive se zapisuju početnim velikim slovom ili simbolom _ ako se radi o **anonimnim promenljivama** čije vrednosti nisu bitne. **Činjenice** su Hornove klauze bez negiranih literala. Sastoje se od **funktora** iza kojeg slede **argumenti** između malih zagrada razdvojeni zarezom. Pomoću činjenica opisuju se svojstva objekata i međusobne relacije između njih. Funktor predstavlja naziv relacije, a argumenti nazive objekata. Imena činjenica treba zadavati u skladu sa njihovom semantikom. Na primer, biljka(bambus), životinja(panda) i jede(panda, bambus). Skup činjenica formira **bazu činjenica**. **Pravila** su Hornove klauze u punom obliku i predstavljaju opšta tvrđenja o objektima i relacijama među njima. Sastoje se iz **glave** i **tela** povezanih **vrat-simbolom** koji odgovara implikaciji: $GLAVA$:- $TELO$ u značenju "važi $GLAVA$ ako važi $TELO$". Na primer, ako želimo da kažemo da Ana poseduje sve što i Pavle, nećemo tvrdnju pisati za svaki mogući predmet, već ćemo koristiti pravilo: poseduje(pavle, X) :- poseduje(ana, X). Za razliku od činjenica, u pravila su uključene i promenljive. Moguće je koristiti rekurziju: na primer, roditelj(A, B) :- dete(B, A) i dete(A, B) :- roditelj(B, A). Ne mogu se koristiti beskonačni ciklusi, a nije dozvoljena ni **levostrana rekurzija**: umesto macka(X) :- macka(Y), majka(Y, X) korisitmo macka(X) :- majka(Y, X), macka(Y). Činjenice i pravila se zajedničkim imenom nazivaju **tvrđenja**. Skup tvrđenja određuje **bazu znanja**. Jedino znanje kojim Prolog raspolaže nalazi se u bazi znanja i vodi se **pretpostavkom zatvorenosti**, odnosno netačno je sve što nije eksplicitno navedeno kao tačno.

Upiti (**ciljevi**) su Hornove klauze bez nenegiranog literala. Preko njih korisnik komunicira sa bazom znanja i najčešće se realizuju u interaktivnom radu korisnika sa računarom. To je moguće ako korisniku na raspolaganju stoji **Prolog-mašina**, tj. interpretator za Prolog. Odzivni znak interpretatora je ?-, pomoć se može dobiti sa $help$, a izlazak iz interpretatora sa $halt$. Činjenice se ubacuju sa $assert$, a spisak činjenica i pravila dobija se sa $listing$. Upiti se završavaju tačkom. Primeri:

- ?-životinja(panda). - da li je panda životinja (yes)
- ?-životinja(bambus). - da li je bambus životinja (no)
- ?-životinja(X). - da li postoji neka životinja (X = panda)
- ?-biljka(Y). - da li postoji neka biljka (Y = bambus)
- ?-jede(panda, Nešto), jede(medved, Nešto). - da li postoji nešto što jedu i panda i medved

Svaki upit Prolog shvata kao neki cilj koji treba ispuniti. On generiše odgovore upoređujući upit sa bazom podataka od početka ka kraju. Složeniji upiti mogu se sastojati iz nekoliko podupita koji se razdvajaju zapetom koja ima ulogu operatora konjunkcije. U tom slučaju, Prolog ispunjava cilj tako što ispunjava svaki potcilj i to s leva u desno. **Unifikacija** je jedna od najvažnijih operacija nad termovima. U Prologu, 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. Isto važi ako je T promenljiva, a S proizvoljan objekat.
- ako su T i S strukture, unifikacija se može izvršiti ako imaju istu arnost, jednake funktore i ako se sve odgovarajuće komponente mogu unifikovati.

Logika prvog reda je apstraktna notacija koja nema algoritamsku prirodu, pa jezici kao što je Prolog sadrže proširenja koja jeziku daju algoritmičnost, kao i proširenja za komunikaciju sa spoljašnjim svetom. **Sistemski predikati** su predikati koji su ugrađeni u sam jezik:

- **operatori jednakosti i nejednakosti**:
  - **unifikacija**: =, \=
  - **dodela**: is
  - **aritmetička jednakost**: =:=, =/=
  - **sintaksni identitet**: ==, \==
- **relacijski operatori**: <, =<, >, =>
- **aritmetički operatori**: +, -, *, /, mod
- **rad sa datotekama** (**bazom znanja**): user, consult, reconsult
- **ostali operatori**: true, fail, not, !

Bez sistemskih predikata Prolog je čist deklarativni jezik. Svaka verzija ima neke specifične sistemske predikate, što dodaje propratne efekte. Najveći broj operatora zahteva konkretizovane promenljive. Na primer, X < Y zahteva da promenljive imaju konkretne vrednosti, dok poređenje u opštem slučaju nema smisla. Primeri:

```
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 =:= X mod Y.

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.
```

Prolog programi su jednoobrazni, tj. podaci i programi izgledaju isto, što 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 pomoću naredbi za dodavanje na početak i kraj - **asserta** i **assertz**, kao i njihovo izbacivanje iz baze - **retract** i **retractall**. Predikati functor, arg, =.. omogućavaju kreiranje i zaključivanje o novim predikatima za vreme izvršavanja programa.

Lista predstavlja niz uređenih elemenata proizvoljne dužine. Element liste može biti bilo koji term, pa čak i druga lista. Lista je ili prazna lista [] ili struktura .(G, R), gde je G ma koji term, a R je lista. Funktor liste je tačka, prvi argument je **glava**, a drugi **rep**. Zapis preko tačke i zagrada je nepregledan, pa se koristi standardni zapis kao i u ostalim jezicima. Primeri:

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

Preko listi se lako definišu i predikati za rad sa listama:

- dužina liste

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

- suma elemenata liste

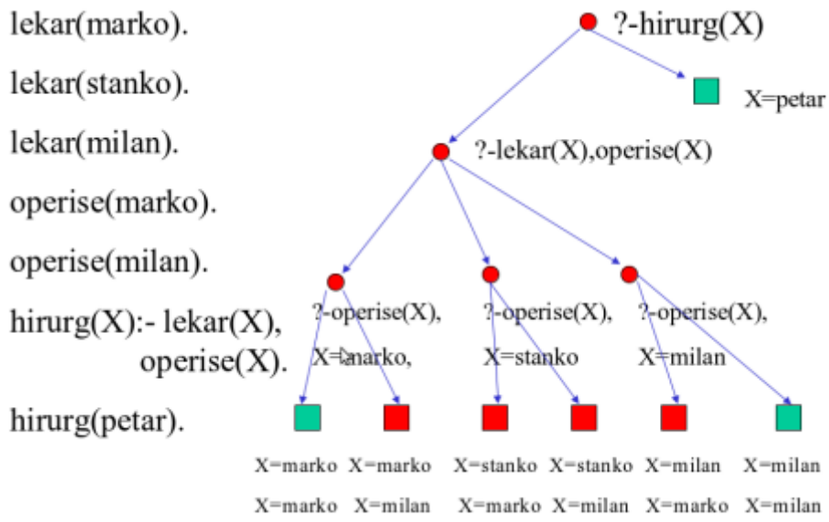
```
?- suma(Lista, Suma).
suma([], 0).
suma([G|R], S) :- suma(R, S1), S is S1 + G.
```

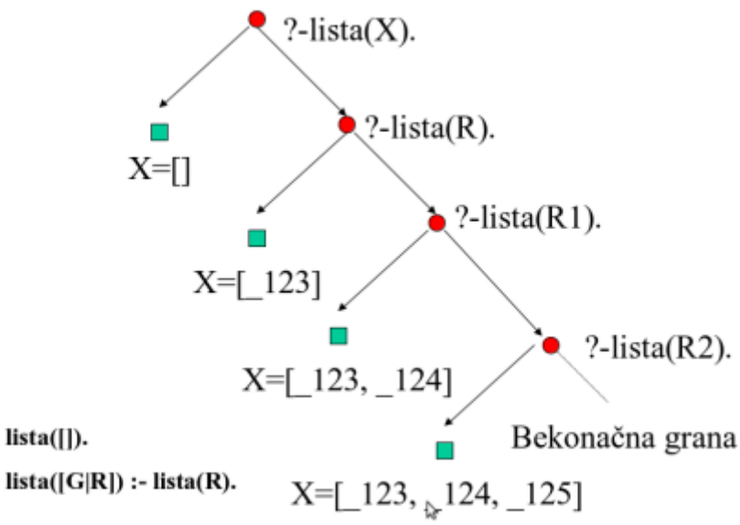
|Prolog-konstrukcije mogu se interpretirati **deklarativno** i **proceduralno**. Primeri:

- životinja(panda)
  - deklarativno: životinja(panda) je istinito.
  - proceduralno: zadatak životinja(panda) je izvršen.
- duga(X) :- kiša(X), sunce(X)
  - deklarativno: za svako X, duga(X) je istinito ako je istinito kiša(X) i sunce(X).
  - proceduralno: da bi se izvršio zadatak duga(X), prvo izvršiti zadatak kiša(X), a zatim izvrši zadatak sunce(X). Proceduralno tumačenje daje i redosled ispunjavanja cilja.
- ?- jede(panda, X)
  - deklarativno: da li postoji vrednost promenljive X za koju važi jede(panda, X).
  - proceduralno: izračunavanjem ostvari cilj jede(panda, X), tj. nađi vrednost promenljive X za koju važi svojstvo jede(panda, X).|

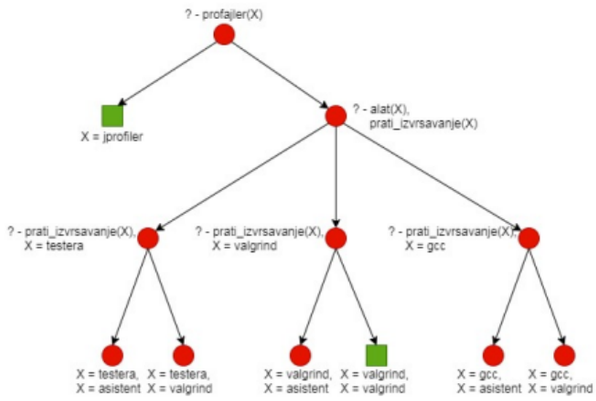
Ostvarivanje cilja Prolog-mašina čini pokušavajući da dokaže saglasnost cilja sa bazom znanja. Baza znanja se pregleda od vrha ka dnu i ako je pronađeno tvrđenje koje se uparuje sa ciljem dolazi do **uspeha**. U tom slučaju, korisnik može zahtevati da se ponovo dokaže saglasnost cilja sa bazom podataka, odnosno da se pronađe još jedno rešenje. Ako nije pronađeno tvrđenje koje se uparuje sa ciljem dolazi do **neuspeha**. **Stablo izvođenja** je stablo pretrage koje omogućava slikovit prikaz načina rešavanja problema u Prologu. Sastoji se iz **grana** i **čvorova**. Grane mogu biti **konačne** ili **beskonačne**. Ako su u stablu sve grane konačne i stablo je konačno, a ako postoji bar jedna beskonačna grana i stablo je beskonačno. Čvorovi koji su označeni upitom su **nezavršni** i iz njih se izvode sledeći čvorovi. Listovi u stablu izvođenja su **završni** i do njih vode konačne grane. Beskonačne grane nemaju završne čvorove. Ako završni čvor daje rešenje naziva se **čvor uspeha**, a odgovarajuća grana je **grana uspeha**. Obrnuto, ako završni čvor ne daje rešenje on je **čvor neuspeha**, a njemu odgovarajuća grana je **grana neuspeha**. 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 potciljeva. Redosled činjenica i pravila u bazi znanja određuje redosled ispunjavanja ciljeva i izgleda stabla izvođenja. Implementacija rezolucije, tj. pretrage, može da se ostvari **izvođenjem u širinu** ili **izvođenjem u dubinu**. Izvođenjem u širinu, radi se paralelno na svim potciljevima datog cilja, dok se izvođenjem u dubinu najpre izvede jedan potcilj, pa se dalje nastavlja sa njegovim potciljevima. U Prologu se vrši izvođenje u dubinu jer se na taj način troši manje memorije. Samim tim, redosled činjenica i pravila u bazi znanja može da utiče na efikasnost pronalaženja rešenja, ali i na konačnost najlevlje grane. Zbog toga leva rekurzija nije dozvoljena. Najčešće se prvo zadaju sve činjenice, pa

zatim pravila. Na efikasnost utiče i davanje pogodnog redosleda potciljeva u pravilima, što narušava pravila deklarativnosti po kojima redosled ne bi trebao da utiče na izvršavanje programa. |Primeri:

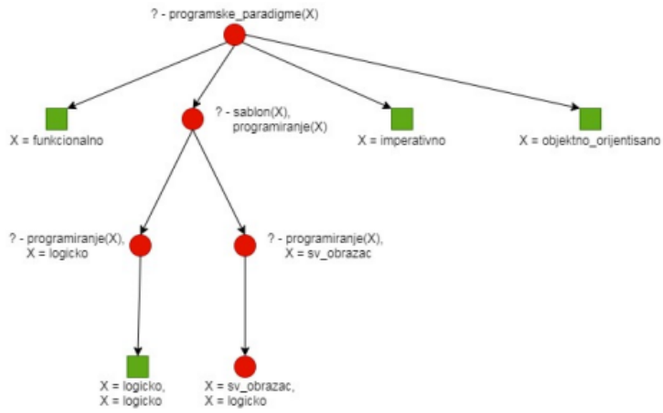




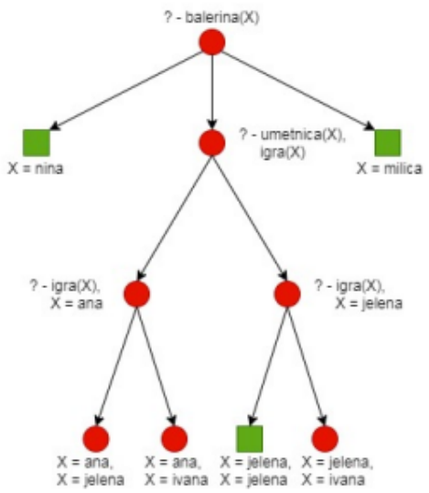
```
alat(testera).
alat(valgrind).
profajler(jprofiler).
prati_izvrsavanje(asistent).
profajler(X):-alat(X),
              prati_izvrsavanje(X).
alat(gcc).
prati_izvrsavanje(valgrind).
```



```
programske_paradigme(funkcionalno).
sablon(logicko).
sablon(sv_obrazac).
programiranje(logicko).
programske_paradigme(X):-sablon(X),
                         programiranje(X).
programske_paradigme(imperativno).
programske_paradigme(objektno_orijentisano).
```



```
umetnica(ana).
balerina(nina).
igra(jelena).
umetnica(jelena).
balerina(X):-umetnica(X), igra(X).
balerina(milica).
igra(ivana).
```



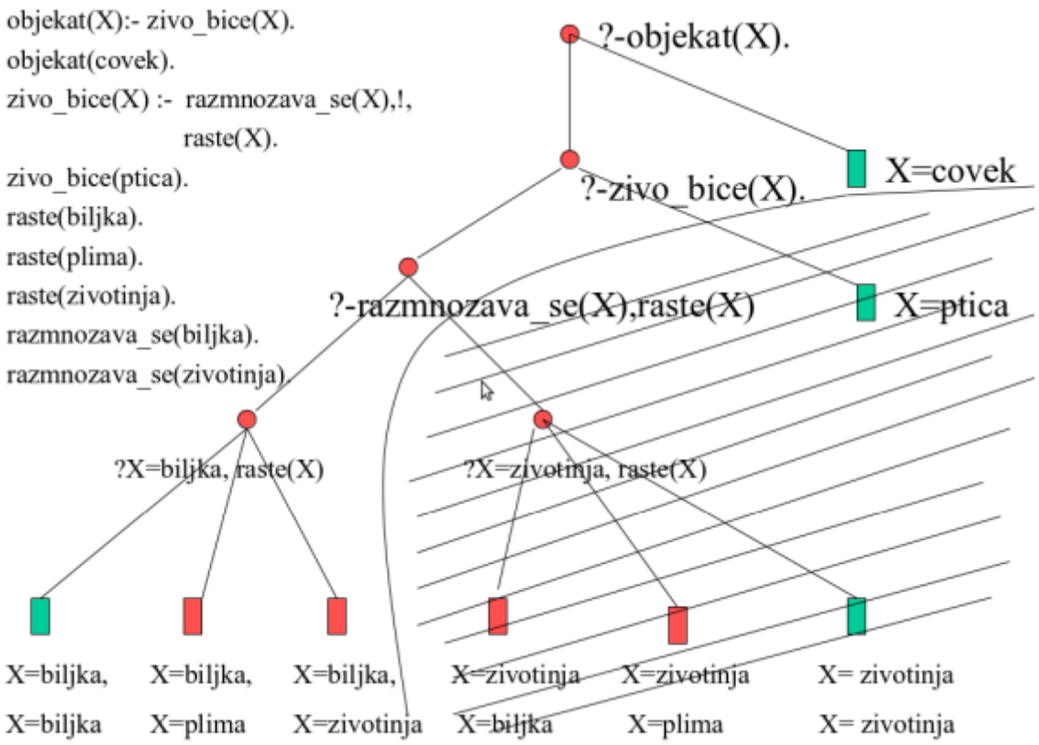
- dodatni primeri|

Operator sečenja je sistemski operator koji omogućava brže izvršavanje programa i uštedu memorijskog prostora kroz eksplicitnu kontrolu backtracking-a, tako što odseca grane u stablu pretraživanja. Ovaj operator je zapravo cilj koji uvek uspeva. Ako imamo pravilo A :- $B_1, B_2, \ldots, B_k, !, \ldots, B_m$, kada se dođe do reza potciljevi pre njega su uspeli, zatim i sečenje uspeva i time se rešenja za $B_1, B_2, \ldots, B_k$ zamrzavaju i više se ne traže alternativna rešenja za njih. Takođe, onemogućava se ujedinjavanje cilja A sa glavom nekog drugog predikata.

Primer: razmotrimo upit ?- G.

```
A :- P, Q, R, !, S, T.
A :- U.
G :- B1, A, B2.
```

- Najpre je potrebno ispuniti cilj B1.
- Dalje, potrebno je ispuniti cilj A, i traženje sa vraćanjem moguće je samo za: P, Q i R.
- Kad uspe R, uspeva i sečenje i alternativna rešenja (za P, Q i R) 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



Predikat sečenja narušava deklarativni stil programiranja jer može da proizvede neželjene efekte. Ako se predikat sečenja upotrebljava tako da ne narušava deklarativno svojstvo predikata, tj. ne menja njegovu semantiku niti skup rešenja, onda se naziva **zeleni predikat sečenja**. U suprotnom, reč je o **crvenom predikatu sečenja**, kao u poslednjem primeru. Primene:

- 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, !.
```

- 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.
```

- Kada želimo da saopštimo Prologu: "Na pogrešnom si putu, završiti pokušaj zadovoljenja cilja!".

```
inostrani(dzon).
inostrani(martin).
dohodak(dzon, 5000).
dohodak(martin, 10000).
dohodak(dragan, 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.

Prolog nije čist deklarativni jezik jer sadrži kontrolu rezolucije i backtracking-a. On predstavlja restrikciju logike prvog reda na Hornove klauze. Prolog može da dokaže da je neki cilj ispunjen, ali ne i da neki cilj nije ispunjen, što je posledica oblika Hornovih klauza. Odnosno, Prolog je **true/fail sistem**, a ne true/false sistem. **Negacija kao neuspeh** (NOT) je operator koji menja uspešnost cilja: NOT(cilj) uspeva ako cilj ne uspeva. On nije operator negacije u smislu logičke negacije, jer logičko not ne može ni biti deo Prologa zbog Hornovih klauza. To znači da NOT(NOT(cilj)) ne mora biti ekvivalentno sa cilj. |Primeri:

```
vozac(janko).
vozac(marko).
vozac(petar).
pije(janko).
pije(marko).
dobar_vozac(X) :- not(pije(X)),
                  vozac(X).
```

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

U ovom primeru dobijamo da Petar jete dobar vozač, ali da ne postoji nijedan dobar vozač. Operator NOT zahteva konkretnu vrednost neke promenljive, pa se iz tog razloga stavlja na poslednje mesto: dobar_vozac(X) :- vozac(X), not(pije(X)). U ovom slučaju će X prvo biti upareno sa konkretnim vozačem, a tek onda će se proveriti da li on pije.

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
```

|

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, što nosi cenu neefikasnosti. Na primer, sortiranje može jednostavno da se definiše, ali je sam algoritam potpuno neefikasan jer nije dat način na koji se vrši sortiranje pa je jedini način enumeracija svih permutacija koja zadovoljava svojstvo sortiranosti liste.

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

Kreiranje modula u Prologu propisano je ISO standardom, ali ne podržavaju svi kompajleri ovo svojstvo pa postoje nekompatibilnosti između sistema modula različitih Prolog kompajlera i teško je razviti kompleksan softver. Prolog je **Tjuring-kompletan** jezik što se može pokazati simuliranjem Tjuringove mašine u Prologu. Prolog je **netipiziran** jezik. Poseduje mehanizam prepoznavanja i **optimizacije repne rekurzije**, tako da se ona izvršava jednako efikasno kao i petlje u proceduralnim jezicima. ISO standard propisuje i **predikate višeg reda** koji uzimaju jedan ili više predikata kao svoje argumente, iako to izlazi iz okvira logike prvog reda.