1. Odnos programskih jezika i programskih paradigmi

1.1 Značenje reči paradigma i programska paradigma.

Reč paradigma znači uzor, obrazac, šablon; obično se koristi da označi vrstu objekata koji imaju zajedničke karakteristike.
Programska paradigma - programski obrazac, stil, šablon; način programiranja.

1.2 Uloga programskih paradigmi.

Izučavanjem programskih paradigmi upoznaju se globalna svojsta jezika koji pripadaju toj paradigmi. Informacija da neki jezik pripada nekoj paradigmi nam govori o osnovnim svojstvima i mogućnostima jezika. Poznavanje određene paradigme nam značajno olakšava da savladamo svaki programski jezik koji toj paradigmi pripada.

1.3 Definicija programskog jezika.

Programski jezik je skup sintaktičkih i semantičkih pravila koji se koriste za opis (definiciju) računarskih programa.

1.4 Povezanost paradigmi i jezika.

Programske paradigme su usko povezane sa programskim jezicima. Svakoj programskoj paradigmi pripada više programskih jezika, ali i jedan programski jezik može podržati više paradigmi. Potrebno je izučiti svojstva najistaknutijih predstavnika pojedinih programskih paradigmi.
Koliko paradigmi znaš, toliko vrediš! Preciznije: Koliko predstavnika različitih paradigmi znaš, toliko vrediš!

1.5 Razvoj programskih jezika.

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

Postoji veliki broj programskih jezika, neki su široko rasprostranjeni, neki se više ne koriste.

1.6 Zašto su nastajale i nastaju nove programske paradigme?

Nove programske paradigme nastajale su uz težnju da se olakša proces programiranja. Istovremeno, nastanak novih paradgmi povezan je sa efikasnim kreiranjem sve kompleksnijeg softvera. Svaka novonastala paradigma, bila je promovisana preko nekog programskog jezika. Razvoj programskih paradigmi (kao i programskih jezika) skopčan je i sa razvojem hardvera.

1.7 Šta karakteriše proceduralnu paradigmu?

Osnovni zadatak programera je da opiše način (proceduru) kojim se dolazi do rešenja problema.

1.8 Šta karakteriše deklarativnu paradigmu?

Osnovni zadatak programera je da precizno opiše problem, dok se mehanizam programskog jezika bavi pronalaženjem rešenja problema.

1.9 Koje su osnovne programske paradigme?
Imperativna, objektno-orijentisana, funkcionalna i logička paradigma.

1.10 Nabroj bar četiri dodatne programske paradigme.
Komponentna, konkurentna, skript, generička, vizuelna, programiranje ograničenja...

1.11 Koje su osnovne karakteristike imperativne paradigme?
Imperativna paradigma nastala je pod uticajem Fon Nojmanove arhitekture računara. Može se reći da se zasniva na tehnološkom konceptu digitalnog računara. Proces izračunavanja se odvija slično kao neke svakodnevne rutine (zanovan je na algoritamskom načinu rada). Može da se okarakteriše rečenicom: *“prvo uradi ovo, zatim uradi ono”.* Procedurom se saopštava računaru kako se problem rešava, tj navodi se precizan niz koraka (algoritam) potreban za rešavanje problema. Osnovni pojam imperativnih jezika je naredba. Naredbe se grupišu u procedure i izvršavaju se sekvencijalno ukoliko se eksplicitno u programu ne promeni redosled izvršavanja naredbi. Upravljačke strukture su naredbe grananja, naredbe iteracije, i naredbe skoka (goto). Oznake promenljivih su oznake memorijskih lokacija pa se u naredbama često mešaju oznake lokacija i vrednosti - to izaziva bočne efekte.

1.12 Nabroj bar tri jezika koji pripadaju imperativnoj paradigmi.
C, Pascal, Basic, Fortran, Algol, PL...

1.13 Koje su osnovne karakteristike objektno-orijentisane paradigme?
Sazrela je početkom osamdesetih godina prošlog veka, kao težnja da se jednom napisani softver koristi više puta. Simualcija (modeliranje) spoljašnjeg sveta pomoću objekata. Objekti interaguju međusobno razmenom poruka. Mogla bi da se okarakteriše rečenicom:
“Uputi poruku objektima da bi simulurao tok nekog fenomena”.
Podaci i procedure (funkcije) se učauravaju (enkapsuliraju) u objekte. Koristi se skrivanje podataka da bi se zaštitila unutrašnja svojstva objekata. Objekti su grupisani po klasama (klasa predstavlja šablon (koncept) na osnovu kojeg se kreiraju konkretni objekti, tj. instance). Klase su najčešće hijerarhijski organizovane i povezane mehanizmom nasledivanja.

1.14 Nabroj bar tri jezika koji pripadaju objektno-orijentisanoj paradigmi.
Java, SmallTalk, C++, Eiffel, C#, Simula67...

1.15 Koje su osnovne karakteristike funkcionalne paradigme?
Izračunavanja su evaluacije matematičkih funkcija. Zasnovana je na pojmu matematičke funkcije i ima formalnu strogo definisanu matematičku osnovu u lambda računu. Mogla bi se okarakterisati narednom rečenicom: *“Izračunati vrednost izraza i koristiti je”.*
Eliminisani su bočni efekti što utiče na lakše razumevanje i predviđanje ponašanja programa — izlazna vrednost funkcije zavisi samo od ulaznih vrednosti argumenata funkcije. Nastala pedesetih i početkom šezdesetih godina prošlog veka, stagnacja u razvoju sedamdesetih godina prošlog veka, oživljavanje funkcionane paradigme programskim jezikom Haskell.

1.16 Nabroj bar tri jezika koji pripadaju funkcionalnoj paradigmi.
Lisp, Haskell, Scheme, ML, Scala, OCaml...

1.17 Koje su osnovne karakteristike logičke paradigme?

Nastaje kao težnja da se u kreiranju programa koristi isti način razmišljanja kao i pri rešavanju problema u svakodnevnom životu. Deklarativna paradigma. Opisuju se odnosi izmedu činjenica i pravila u domenu problema; koriste se aksiome, pravila izvodenja i upiti. Logička paradigma se dosta razlikuje od svih ostalih po načinu pristupa rešavanju problema. Nije jednako pogodna za sve oblasti izračunavnja, osnovni domen je rešavanje problema veštačke inteligencije. Izvršavanje programa zasniva se na sistematskom pretraživanju skupa činjenica uz korišćenje odredenih pravila zaključivanja. Zasnovana na matematičkoj logici, tj. na predikatskom računu prvog reda. Zasnovana na automatskom dokazivanju teorema (metod rezolucije). Mogla bi da se okarakteriše rečenicom:

“Odgovori na pitanje kroz traženje rešenja”

1.18 Nabroj bar tri jezika koji pripadaju logičkoj paradigmi.

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

1.19 Koje su osnovne karakteristike komponentne paradigme?

Ideja je da se uprosti proces programiranja i da se jednom kreirane komponente mnogo puta koriste. Komponenta je jedinica funkcionalnosti sa “ugovorenim” interfejsom. Interfejs definiše način na koji se komunicira sa komponentom, i on je u potpunosti odvojen od implementacije. Komponente se međusobno povezuju da bi se kreirao kompleksan softver. Način povezivanja komponenti treba da bude jednostavan, po mogućnosti prevlačenjem i spuštanjem na željenu lokaciju. Kreiranje programa se vrši biranjem komponenti i postavljanjem na pravo mesto, a ne pisanjem “linije za linijom”.

1.20 Nabroj tri jezika koji podržavaju komponentnu paradigmu.

Java, C++, C#, Python...

1.21 Koje su osnovne karakteristike konkurentne paradigme?

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

- Konkurentnost u užem smislu — jedan procesor, jedna memorija
- Paralelno programiranje — više procesora, jedna memorija
- Distribuirano programiranje — više procesora, više memorija

1.22 Nabroj tri jezika koji podržavaju konkurentnu paradigmu.

Ada, Modula, ML, Java, Scala...

1.23 Koje su osnovne karakteristike paradigme programiranja ograničenja?

U okviru paradigme programiranja ograničenja zadaju se relacije između promenljivih u formi nekakvih ograničenja. Ograničenja mogu biti raznih vrsta (logička, linearna...). Ova ograničenja ne zadaju sekvencu koraka koji treba da se izvrše već osobine rešenja koje treba da se pronađe. Deklarativna paradigma.

1.24 Nabroj tri jezika koji podržavaju paradigmu programiranja ograničenja.

C, Java, C++, Python, BProlog, OZ, Claire, Curry...

1.25 Koje su osnovne karakteristike skript paradigme?

Skript jezik je programski jezik koji služi za pisanje skriptova. Skript je spisak (lista) komandi koje mogu biti izvršene u zadatom okruženju bez interakcije sa korisnikom. U prvobitnom obliku pojavljuju se kao komandni jezici operativnih sistema (npr Bash). Skript jezici imaju veliku primenu na Internetu. Skript jezici mogu imati specifičan domen primene, ali mogu biti i jezici opšte namene (npr Python). Skript jezici se ne kompiliraju već interpreteraju. Često se koriste za povezivanje komponenti unutar neke aplikacije. Omogućavaju kratak kod. Najčešće nisu strogo tipizirani. Kôd i podaci često mogu zameniti uloge. Nije uvek lako napraviti razliku između skript-jezika i drugih programskih jezika.

1.26 Nabroj tri jezika koji podržavaju skript paradigmu.

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

1.27 Koje su osnovne karakteristike paradigme upitnih jezika?

Upitni jezici mogu biti vezani za baze podataka ili za pronalaženje informacija (information retrieval). Deklarativna paradigma.

- Upitni jezici baza podataka — oni na osnovu strukturiranih činjenica zadatih u okviru strukturiranih baza podataka daju konkretne odgovore koji zadovoljavanju nekakve tražene uslove.
- Upitni jezici za pronalaženje informacija — upitni jezici koji pronalaze dokumenta koji sadrže informacije relevantne za oblast istraživanja.

1.28 Nabroj tri jezika koji podržavaju paradigmu upitnih jezika.

SQL, XQuery; CQL...

1.29 Koje su osnovne karakteristike reaktivne paradigme?

Rekativno programiranje je usmereno na tok podatka u smislu prenošenja izmena prilikom promene podataka.

- programiranje u okviru tabela
- jezici za opis hardvera

1.30 Nabroj tri jezika koji podržavaju reaktivnu paradigmu.

Excel, LibreOffice, Calc; Verilog, VHDL...

1.31 Koje su osnovne karakteristike vizuelne paradigme?

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

1.32 Nabroj tri jezika koji podržavaju vizuelnu paradigmu.

UML

2. Funkcionalna paradigma

2. 1 Na koji način je John Backus uticao na razvoj funkcionalnih jezika?

Backus je održao predavanje sa poentom da su čisti funkcionalni programski jezici bolji od imperativnih jer su programi koji se pišu na njima čitljiviji, pouzdaniji i verovatnije ispravni. Suština njegove argumentacije bila je da su funkcionalni jezici lakši za razumevanje, i za vreme i nakon razvoja, najviše zbog toga što vrednost izraza nezavisna od konteksta u kojem se izraz nalazi.

2.2 Koji su najpoznatiji funkcionalni programski jezici?

Lisp, Scheme, Miranda, Haskell, F#, Scala...

2.3 Koji je domen upotrebe funkcionalnih programskih jezika?

Obrada baza podatka, finansijsko modelovanje, statistička analiza, bioinformatika, veštačka inteligencija, paralelizacija...

2.4 Koje su osnovne karakteristike funkcionalnih programskih jezika?

Osnovna apstrakcija je funkcija. Funkcionalno programiranje je stil koji se zasniva na izračunavanju izraza kombinovanjem funkcija.
Nepostojanje stanja, rekurzija je prirodna, mogućnost pravljenja eksplicitnog stanja po potrebi, referentna prozirnost...

2.5 Šta je svojstvo transparentnosti referenci i na koji način ovo svojstvo utiče na redosled naredbi u funkciji?

Referentna transparentnost (prozirnost) je osobina da je vrednost izraza svuda jedinstveno određena. Govori da redosled naredbi nije bitan.

2.6 Koje su osobine programa u kojima se poštuje pravilo transparentnosti referenci?

Takvi programi su formalno koncizni, prikladni za formalnu verifikaciju, manje podložni greškama i lakše ih je paralelizovati.

2.7 Da li je moguće u potpunosti zadržati svojstvo transparentnost referenci?

Da, ali to ima svoju cenu.

2.8 Koji je odnos transparentnosti referenci sa bočnim efektima?

Ako želimo da imamo u potpunosti referencijalnu transparentnost, ne smemo dopustiti nikakve bočne efekte. U praksi je to veoma teško jer postoje neki algoritmi koji se suštinski temelje na promeni stanja, dok neke funkcije postoje samo zbog svojih bočnih efekata.

2.9 Da li je moguće obezbediti promenu stanja programa i istovremeno zadržati svojstvo transparentnosti referenci?

Da, uvođenjem eksplicitnih stanja.

2.10 Šta su funkcionalni jezici? Šta su čisto funkcionalni jezici?

Funkcionalna paradigma se zasniva na pojmu matematičkih funkcija. Osnovni cilj funkcionalne paradigme je da oponaša matematičke funkcije. Čisto funkcionalni programski jezici ne dopuštaju baš nikakve bočne efekte.

2.11 Navesti primere čisto funkcionalnih jezika?
Clean, Haskell, Miranda.

2.12 Koje su osnovne aktivnosti u okviru funkcionalnog programiranja?
1. Definisanje funkcije - pridruživanje imenu funkcije vrednosti izraza pri čemu izraz može sadržati pozive drugih funkcija
2. Primena funkcije - poziv funkcije sa zadatim argumentima
3. Kompozicija funkcija - navodenje niza poziva funkcija; kreiranje programa

2.13 Kako izgleda program napisan u funkcionalnom programskom jeziku?
Program je niz definicija i poziva funkcija.

2.14 Šta je potrebno da obezbedi funkcionalni programski jezik za uspešno proramiranje?
Treba ugraditi neke osnovne funkcije u sam programski jezik, obezbediti mehanizme za formiranje novih funkcija, obezbediti strukture za prezentovanje podataka i formirati biblioteku funkcija koje mogu biti korišćene kasnije.

2.15 Šta je striktna/nestriktna semantika?
Izraz je *striktan* ako nema vrednost kad bar jedan od njegovih operanada nema vrednost.
Izraz je *nestriktan* kada ima vrednost čak i ako neki od njegovih operanada nema vrednost.
Semantika je *striktna* ako je svaki izraz tog jezika striktan. Inače je *nestriktna*.

2.16 Kakvu semantiku ima jezik Haskell?
Nestriktnu, kao i Miranda.

2.17 Kakvu semantiku ima jezik Lisp?
Striktnu, kao i OCaml i Scala.

2.18 Koje su prednosti funkcionalnog programiranja?
- Za funkcionalne programe lakše je konstruisati matematički dokaz ispravnosti.
- Stil programiranja nameće razbijanje koda u manje delove koji imaju jaku koheziju i izraženu modularnost.
- Pisanje manjih funkcija omogućava veću čitljivost koda i lakšu proveru grešaka.
- Testiranje i debagovanje je jednostavnije.
- Mogu se jednostavno graditi biblioteke funkcija.

2.19 Koje su mane funkcionalnog programiranja?
- Efikasnost se dugo navodila kao mana, ali zapravo efikasnost odavno nije problematična.
- Debagovanje ipak može da bude komplikovano (kada je nestriktna semantika u pitanju).
- Često se navodi da je funkcionalno programiranje teško za učenje.

2.20 Šta uključuje definisanje funkcije?
Definicija funkcije obično uključuje ime funkcije za kojom sledi lista parametara u zagradama, a zatim i izraz koji zadaje preslikavanje.

2.21 Šta su funkcije višeg reda? Navesti primere.

Funkcije višeg reda imaju jednu ili više funkcija kao parametre ili imaju funkciju kao rezultat, ili oba.
Primeri: kompozicija funkcija i α funkcija

2.22 Da li matematičke funkcije imaju propratne efekte?

Nemaju.

2.23 Koji je formalni okvir funkcionalnog programiranja?

Lambda račun.

2.24 Koji se jezik smatra prvim funkcionalnim jezikom?

Lambda račun.

2.25 Koja je ekspresivnost lambda računa?

Ekvivalentna je ekspresivnosti Tjuringovih mašina.

2.26 Koji su sve sinonimi za lambda izraz?

Funkcijska astrakcija, anonimna funkcija ili bezimena funkcija.

2.27 Navesti definiciju lambda terma.

Promenljive: Promenljiva je validni lambda term.
λ-apstrakcija: Ako je *t* lambda term, a *x* promenljiva, onda je *λx.t* lambda term.
λ-primena: Ako su *t* i *s* lambda termovi, onda je *(t s)* lambda term.
Lambda termovi se mogu konstruisati samo konačnom primenom prethodnih pravila.

2.28 Da li čist lambda račun uključuje konstante u definiciji?

Ne.

2.29 Navesti primer jednog lambda izraza, objasniti njegovo značenje i primeniti dati izraz na neku konkretnu vrednost.

λx.x · x + 3 — intuitivno, kvadriranje i uvećanje za 3: x → x · x + 3
(λx.x · x + 3)2 → 2 · 2 + 3 → 4 + 3 → 7

2.30 Koja je asocijativnost primene a koja apstrakcije?

Primena funkcije je levo asocijativna, dok je apstrakcija desno asocijativna.

2.31 Navesti definiciju slobodne promenljive. Koje promenljive su vezane?

Promenljive: Slobodna promenljiva terma *x* je samo *x*.
Apstrakcija: Skup slobodnih promenljivih terma *λx.t* je skup slobodnih promenljivih terma *t* bez promenljive *x*.
Primena: Skup slobodnih promenljivih terma *(t s)* je unija skupova slobodnih promenljivih terma *t* i terma *s*.
Promenljive koje nisu slobodne su vezane.

2.32 Koja je uloga pojma alfa ekvivalentnosti?

α ekvivalentnost se definiše za lambda termove, sa ciljem da se uhvati intuicija da izbor imena vezane promenljive u lambda računu nije važan.

2.33 Šta su redukcije?
Redukcije daju uputstva kako transformisati izraze iz početnog stanja u neko finalno stanje.

2.34 Šta je delta redukcija? Navesti primer.
δ redukcija se odnosi na trasformaciju funkcija koje kao argumente sadrže konstante.
Primer: $3 + 5 \rightarrow 8$

2.35 Šta je alfa redukcija? Navesti primer.
α redukcija ili α preimenovanje, dozvoljava da se promene imena vezanim promenljivama.
Primer: $\lambda x.x \rightarrow \lambda y.y$

2.36 Kada se koristi alfa redukcija?
Alfa preimenovanje je nekada neophodno da bi se izvršila beta redukcija.

2.37 Šta je beta redukcija? Navesti primer.
β-redukcija u telu lambda izraza formalni argument zamenjuje aktuelnim argumentom i vraća telo funkcije (dakle svako pojavljivanje promenljive u telu se zamenjuje sa datim izrazom).
Primer: $(\lambda x.x + 1)5 \rightarrow [5/x](x + 1) = 5 + 1 \rightarrow 6$

2.38 Definisati supstituciju.
Supstitucija [I/P]T je proces zamene svih slobodnih pojavljivanja promenljive *P* u telu lambda izraza *T* izrazom *I* na sledeći način (*x* i *y* su promenljive, a *M* i *N* lambda izrazi):
Promenljive: $[N/x]x = N$
$[N/x]y = y$ pri čemu je $x \neq y$
Apstrakcija: $[N/x](\lambda x.M) = \lambda x.M$
$[N/x](\lambda y .M) = \lambda y.([N/x]M)$ ukoliko je $x \neq y$ i *y* ne pripada skupu slobodnih promenljivih za *N*
Primena: $[N/x](M_1M_2) = ([N/x])(M_1)([N/x])(M_2)$

2.39 Navesti primer lambda izraza koji definiše funkciju višeg reda koja prima funkciju kao argument.
Funkcija koja očekuje funkciju kao argument, tu će funkciju primeniti negde u okviru svog tela.
Primer: $\lambda x.(x\ 2) + 1$

2.40 Navesti primer lambda izraza koji definiše funkciju višeg reda koja ima funkciju kao povratnu vrednost.
Funkcija koja vraća funkciju će u svom telu sadržati drugi lambda izraz.
Primer: $\lambda x.(\lambda y.2 \cdot y + x)$.

2.41 Čemu služi Karijev postupak?
Za definisanje funkcija sa više argumenata. Funkcija koja treba da uzme npr. dva argumenta, prvo će uzeti jedan argument, pa će od njega napraviti funkciju koja će potom da uzme drugi argument.

2.42 Kako se definišu funkcije sa više argumenata?
Funkcije oblika $f(x_1, x_2,..., x_n) = telo$, u lambda računu se definišu kao $\lambda x_1.(\lambda x_2.(...(\lambda x_n.telo)...))$, tj. kraće $\lambda x_1x_2...x_n.telo$.

2.43 Šta je normalni oblik funkcije?
Višestrukom beta redukcijom izračunavamo 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.

2.44 Da li svi izrazi imaju svoj normalni oblik?
Ne. *Primer:* $(\lambda x.x\ x)(\lambda x.x\ x)$.

2.45 Navesti svojstvo konfluentnosti.
Svojstvo konfluentnosti ili Church-Rosser teorema:
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*.

2.46 Da li izraz može imati više normalnih oblika?
Ne. (posledica prethodne teoreme)

2.47 Koja je razlika između aplikativnog i normalnog poretka?
Aplikativni poredak - odogovara pozivu po vrednosti (call-by-value) — izračunavamo vrednost argumenta i tek kada ga izračunamo šaljemo ga u funkciju i funkcija dočeka u svom telu izračunati argument.
Normalni poredak - beta redukcijom uvek redukovati najlevlji izraz. Ovo odgovara evaluaciji po imenu (call-by-name) ili evaluaciji po potrebi (call-by-need).

2.48 Šta govori teorema standardizacije?
Ako je *Z* normalni oblik izraza *E*, onda postoji niz redukcija u normalnom poretku koji vodi od *E* do *Z*.

2.49 Šta se dobija lenjom evaluacijom?
Normalnim poretkom redukcija ostvaruje se lenja evaluacija — izrazi se evaluiraju samo ukoliko su potrebni. Lenjom evaluacijom se izbegavaju nepotrebna izračunavanja. Lenja evaluacija nam garantuje završetak izračunavanja uvek kada je to moguće

2.50 Koje su osnovne karakteristike Haskell-a?
Haskell je čist funkcionalni jezik. Ima lenju evaluaciju, moćni sistem tipova (automatsko zaključivanje tipova). Strogo tipiziran jezik. Podržava paralelno i konkurentno programiranje. Podržava parametarski polimorfizam i preopterećivanje. Podržava kompaktan i ekspresivan način definisanja listi kao osnovnih struktura funkcionalnog programiranja. Naglašava upotrebu rekurzije. Funkcije višeg reda omogućavaju visok nivo apstrakcije i korišćenja funkcijskih oblikovanih obrazaca. Ima podršku za monadičko programiranje koje omogućava da se propratni efekti izvedu bez narušavanja referentne transparentnosti. Ima razrađenu biblioteku standardnih funkcija i dodatnih modula.

3. Konkurentno programiranje

3.1 Šta je konkurentna paradigma?
Konkurentnu paradigmu karakteriše više procesa koji se izvršavaju u istom vremenskom periodu, a koji imaju zajednički cilj.

3.2 Da li su ideje o konkurentnosti nove? Zbog čega je konkurentnost važna?
Ideje nisu nove. Veliki deo teorijskih osnova datira još iz 1960tih godina, a već Algol 68 sadrži podršku za konkurentno programiranje.
Konkurentni mehanizmi su originalno izmišljeni za rešavanje odredenih problema u okviru operativnih sistema, ali se sada koriste u raznim aplikacijama.

3.3 Koji su osnovni nivoi konkurentnosti?
1. nivo instrukcije - izvršavanje 2 ili više instrukcija istovremeno
2. nivo naredbe - izvršavanje 2 ili više naredbi jezika višeg nivoa istovremeno
3. nivo jedinica - izvršavanje 2 ili više jedinica istovremeno
4. nivo programa - izvršavanje 2 ili više programa istovremeno

3.4 Koje su vrste konkurentnosti u odnosu na hardver? Kako se to odnosi na programere i dizajn programskog jezika?
- Konkurentna paradigma je najširi pojam, tj konkurentnu paradigmu karakteriše više procesa koji se izvršavaju u istom vremenskom periodu.
- Paralelna paradigma je specijalizacija koja obuhvata postojanje više procesora i omogućava istovremeno izvršavanje.
- Distribuirana paradigma specijalizacija paralelene paradigme u kojoj su procesori i fizički razdvojeni.

3.5 Šta je osnovni cilj koji želimo da ostvarimo razvijanjem konkurentnih algoritama?
Konkurentnim programiranjem treba da se proizvedu skalabilni i portabilni algoritmi.

3.6 Koji su osnovni razlozi za korišćenje konkurentnog programiranja?
Podrška logičkoj strukturi problema, dobijanje na brzini i rad sa fizički nezavisnim uređajima.

3.7 Navesti primer upotrebe konkurentnog programiranja za podršku logičkoj strukturi programa.
Primer aplikacje čiji se dizajn oslanja na konkurentnost je veb razgledač (eng. web browser). Brauzeri moraju da izvršavaju više različitih funkcionalnosti istovremeno (konkurentnost na nivou jedinica), npr primanje i slanje podataka serveru, prikaz teksta i slika na ekranu, reagovanje na korisničke akcije mišom i tastaturom...

3.8 Da li je dobijanje na brzini moguće ostvariti i na jednoprocesorskoj mašini?
Da, konkurentnim izvršavanje.

3.9 Koji je hijerarhijski odnos u okviru konkurentne paradigme?

Sa stanovišta semantike hijerarhijska podela nema značajnu ulogu, bitna je po pitanju implementacije i performansi.

3.10 Šta je zadatak? Na koji način se zadatak razlikuje od potprograma?

Zadatak ili posao je jedinica programa, slična potprogramu, koja može da se izvrši konkurento sa drugim jedinicama istog programa. Zadatak se razlikuje od potprograma na tri načina:

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

3.11 Koje su osnovne kategorije zadataka i koje su karakteristike ovih kategorija?

Zadaci se dele na dve opšte kategorije: *heavyweight* i *lightweight*. Teški imaju svoj sopstveni adresni prostor, dok laki dele isti adresni prostor.
Teškima upravlja operativni sistem obezbeđujući deljenje procesorskog vremena, pristup datotekama, adresni prostor. Prelaz sa jednog teškog zadatka na drugi vrši se posredstvom operativnog sistema uz pamćenje stanja prekinutog procesa (skupa operacija). Stanje teškog zadatka obuhvata: podatke o izvršavanju, vrednosti registara, brojač instrukcija, informacije o upravljanju resursima.
Koncept *lakih* zadataka se uvodi kako bi se omogućio efikasniji prelaz sa jednog na drugi zadatak. Za razliku od teških zadataka, laki ne zahtevaju posebne računarske resurse, već postoje untar jednog teškog zadatka. Podaci koji se vode na nivou teškog zadatka su zajednički za sve lake zadatke koje on obuhvata. Podaci o lakom zadatku obuhvataju samo njegovo stanje izvršavanja, brojač instrukcija i vrednosti radnih registara. Prelaz sa jednog lakog zadatka na drugi je stvar izmene sadržaja radnih registar i pamćenja brojača instrukcija i stanja izvršavanja i vrši se brzo.

3.12 Šta je paralelizacija zadataka?

Paralelizacija zadataka - najčešća strategija, koja dobro radi na malim mašinama. Koriste se različite niti za svaki od glavnih programerskih zadataka ili funkcija. Nedostatak ove strategije je da ne skalira prirodno dobro ukoliko imamo veliki broj procesora.

3.13 Šta je paralelizacija podataka?

Za dobro skaliranje na velikom broju procesora, potreban je *paralelizam podataka*.
Kod paralelizma podataka, iste operacije se primenjuju konkurentno na elemente nekog velikog skupa podataka.

3.14 Navesti primere paralelizacije zadataka i paralelizacije podataka. Koji je odnos ovih paralelizacija?

Primer (paralelizacija zadataka): kod procesora reči, jedan zadatak bi mogao da bude zadužen za prelamanje paragrafa u linije, drugi za određivanje strana i raspored slika, treći za pravopisnu proveru i proveru gramatičkih grešaka, četvrti za renderovanje slika na ekranu...
Primer (paralelizacija podataka): program za manipulaciju slikama, može da podeli ekran na *n* manjih delova i da koristi različite niti da bi procesirao svaki taj pojedinačni deo.

3.15 Šta je komunikacija?
Komunikacija se odnosi na svaki mehanizam koji omogućava jednom zadatku da dobije informacije od drugog.

3.16 Koji su osnovni mehanizmi komunikacije?
Komunikacija se može ostvariti preko zajedničke memorije (ukoliko zadaci dele memoriju) ili slanjem poruka.

3.17 Šta karakteriše slanje poruka u okviru iste mašine, a šta ukoliko je slanje poruka preko mreže?
Ukoliko je slanje poruka u okviru iste mašine, onda se ono smatra pouzdanim. Slanje poruka u distribuiranim sistemima nije pouzdano jer podaci putuju kroz mrežu gde mogu da se zagube i tu su potrebni dodatni mehanizmi i protokoli komunikacije.

3.18 Šta je sinhronizacija?
Sinhronizacija se odnosi na mehanizam koji dozvoljava programeru da kontroliše redosled u kojem se operacije dešavaju u okviru različitih zadataka.

3.19 Kakva je sinhronizacija u okviru modela slanja poruka?
Sinhronizacija je obično *implicitna* u okviru modela slanja poruka: poruka mora prvo da se pošalje da bi mogla da se primi, tj ako zadatak pokuša da primi poruku koja još nije poslata, mora da sačeka pošiljaoca da je najpre pošalje.

3.20 Kakva je sinhronizacija u okviru modela deljene memorije?
Sinhronizacija *nije implicitna* u okviru modela deljene memorije.

3.21 Koje su dve osnovne vrste sinhronizacije u okviru modela deljene memorije?
Postoje dve vrste sinhronizacije kada zadaci dele podatke: saradnja i takmičenje.

3.22 Objasniti sinhronizaciju saradnje.
Sinhronizacija saradnje je neophodna izmedu zadatka A i zadatka B kada A mora da čeka da B završi neku aktivnost pre zadatka A da bi zadatak A mogao da počne ili da nastavi svoje izvršavanje.
Primer: proizvodač-potrošač

3.23 Šta je uslov takmičenja?
Sinhronizacija takmičenja je neophodna između dva zadatka kada oba zahtevaju nekakav resurs koji ne mogu istovremeno da koriste, npr ako zadatak A treba da pristupa deljenom podatku x dok B pristupa x, tada zadatak A mora da čeka da zadatak B završi procesiranje podatka x.
Primer: korišćenje štampača

3.24 Koji su načini implementiranja sinhronizacije?
- Busy-waiting synchronization — zadatak u petlji stalno proverava da li je neki uslov ispunjen. Nema smisla na jednom procesoru.
- Blokirajuća sinhronizacija — zadatak svojevoljno oslobađa procesor, a pre toga ostavlja poruku u nekoj strukturi podataka zaduženoj za sinhronizaciju. Zadatak koji ispunjava uslov u nekom trenutku naknadno, pronalazi poruku i sprovodi akciju da se prvi zadatak odblokira, tj da nastavi sa radom.

3.25 Šta je koncept napredovanja?
Kod sekvencionalnog izvršavanja programa, program ima karakteristiku napredovanja ukoliko nastavlja sa izvršavanjem, dovodeći do završetka rada programa u nekom trenutku. Opštije, koncept napredovanja govori da ukoliko neki događaj treba da se desi da će se on i desiti u nekom trenutku, odnosno da se stalno pravi nekakav napredak.

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

3.27 Navesti primer živog blokiranja.
Primer: Kada se dve osobe nađu u uskom prolazu i treba da se mimoiđu. Obe osobe se istovremeno pomeraju u levu pa u desnu stranu pokušavajući da propuste jedna drugu. Osobe ne napreduju, ali ni ne stoje u mestu.

3.28 Navesti primer individualnog izgladnjivanja.
Primer: Imamo raskrsnicu. Ako su automobili koji se kreću u kolonama na putu sa prvenstvom prolaza, ostali neće imati mogućnost da uđu u raskrsnicu. Kada svi automobili iz kolona, koje se nalaze na putu sa prvenstvom prolaza prođu raskrsnicu, stvoriće se uslovi da i ostali automobili prođu raskrsnicu i nastave svojim putem.

3.29 Koje vrste uzajamnog isključivanja postoje?
Semafori i monitori.

3.30 Koji je odnos konkurentnosti i potprograma/klasa.
- Jedinstveno pozivanje — ne sme se istovremeno upotrebljavati u različitim nitima
- Ulazne (engl. reentrant) — može se upotrebljavati na različitim nitima ali uz dodatne pretpostavke (da se ne koriste nad istim podacima)
- Bezbedne po niti (engl. thread-safe) — sme se bezbedno upotrebljavati bez ograničenja
- Slično važi i za klase

3.31 Opisati problem filozofa za večerom.
Pet filozofa je za stolom na kojem se nalazi pet tanjira i pet viljuški. Da bi filozof mogao da jede mora da uzme obe viljuške ispred sebe. Problem je ako svaki filozof uzme po jednu viljušku. U tom slučaju dolazi do zaglavljivanja i niko ne može da jede.

3.32 Semantika muteksa i katanaca.
Semafori su prvi oblik sinhronizacije, implementiran već u Algol 68, i dalje prisutni. Semafori imaju dve moguće operacije, P i V (wait i release). Nit koji poziva P atomično umanjuje brojač i čeka da *n* postane nenegativan. Nit koji poziva V atomično uvećava brojač i budi čekajuću nit, ukoliko postoji. Iako se široko koriste, semafori se takođe smatraju kontrolom niskog nivoa, nepogodnom za dobro struktuiran i lako održiv kod. Korišćenje semafora lako dovodi do grešaka i do uzajamnog blokiranja.
Drugi koncept su *monitori* koji enkapsuliraju deljene strukture podataka sa njihovim operacijama, tj čine deljene podatke apstraktnim tipovima podataka sa specijalnim ograničenjima. I monitori se takode smatraju kontrolom niskog nivoa.

4. Logičko programiranje

4.1 Šta čini teorijske osnove logičkog programiranja?
Logika prvog reda.

4.2 Na koji način se rešavaju problemi u okviru logičke paradigme?
U logičkom programiranju, logika se koristi kao deklarativni jezik za opisivanje problema, a dokazivač teorema kao mehanizam za rešavanje problema. Rešavanje problema je podeljeno između programera koji opisuje problem i dokazivača teorema koji problem rešava.

4.3 Koji su osnovni predstavnici logičke paradigme?
Prolog, ASP, Datalog.

4.4 Za koju vrstu problema je pogodno koristiti logičko programiranje?
Pogodno je za rešavanje problema matematičke logike, obradu prirodnih jezika, podršku relacionim bazama podataka, automatizaciju projektovanja, simboličko rešavanje jednačina, razne oblasti veštačke inteligencije...

4.5 Za koju vrstu problema nije pogodno koristiti logičko programiranje?
Za I/O algoritme, grafiku, numeričke algoritme...

4.6 Definisati logičke i nelogičke simbole logike prvog reda.
Logički deo jezika prvog reda čine (logički simboli):
- skup promenljivih V
- skup logičkih veznika $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$
- skup kvantifikatora $\{\exists, \forall\}$
- skup logičkih konstanti $\{T, \bot\}$
- skup pomoćnih simbola {(,), , }.

Nelogički deo jezika prvog reda čine skupovi Σ - skup funkcijskih simbola i Π - skup predikatskih simbola, a sve njihove elemente zovemo nelogičkim simbolima.

4.7 Definisati term logike prvog reda.
Skup termova nad signaturom $L = (\Sigma, \Pi, ar)$ i skupom promenljivih V je skup za koji važi:
- svaki simbol konstante (tj. svaki funkcijski simbol arnosti 0) je term;
- svaki simbol promenljive je term;
- ako je f funkcijski simbol za koji je $ar(f) = n$, i $t_1,..., t_n$ su termovi, onda je i $f(t_1,..., t_n)$ term.

Termovi se mogu dobiti samo konačnom primenom prethodnih pravila.

4.8 Definisati atomičku formulu logike prvog reda.
Skup atomičkih formula nad signaturom $L = (\Sigma, \Pi, ar)$ i skupom promenljivih V je skup za koji važi:
- logičke konstante T i $\bot$ su atomičke formule;
- ako je p predikatski simbol za koji je $ar(p) = n$, i $t_1,..., t_n$ su termovi, onda je $p(t_1,..., t_n)$ atomička formula.

4.9 Šta je literal? Šta je klauza?

Literal je atomička formula ili negacija atomičke formule.
Klauza je disjunkcija literala.

4.10 Definisati supstituciju za termove.

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

- ako je t simbol konstante, onda je $t[x \to t_x] = t$;
- ako je $t = x$, onda je $t[x \to t_x] = t_x$;
- ako je $t = y$, gde je $y \neq x$, onda je $t[x \to t_x] = t$;
- ako je $t = f(t_1,..., t_n)$, onda je $t[x \to t_x] = f(t_1[x \to t_x],..., t_n[x \to t_x])$.

4.11 Definisati supstituciju za atomičke formule.

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

- $T[x \to t_x] = T$;
- $\bot[x \to t_x] = \bot$;
- ako je $A = p(t_1,..., t_n)$, onda je $A[x \to t_x] = p(t_1[x \to t_x],..., t_n[x \to t_x])$;
- $(\neg A)[x \to t_x] = \neg(A[x \to t_x])$;
- $(A \wedge B)[x \to t_x] = (A[x \to t_x] \wedge B[x \to t_x])$;
- $(A \vee B)[x \to t_x] = (A[x \to t_x] \vee B[x \to t_x])$;
- $(A \Rightarrow B)[x \to t_x] = (A[x \to t_x] \Rightarrow B[x \to t_x])$;
- $(A \Leftrightarrow B)[x \to t_x] = (A[x \to t_x] \Leftrightarrow B[x \to t_x])$;
- $(\forall xA)[x \to t_x] = (\forall xA)$;
- $(\exists xA)[x \to t_x] = (\exists xA)$;
- ako je $x \neq y$, neka je z promenljiva koja se ne pojavljuje ni u $(\forall y)A$ ni u t_x; tada je $(\forall y A)[x \to t_x] = (\forall z)A[y \to z][x \to t_x]$;
- ako je $x \neq y$, neka je z promenljiva koja se ne pojavljuje ni u $(\exists y)A$ ni u t_x; tada je $(\exists y A)[x \to t_x] = (\exists z)A[y \to z][x \to t_x]$.

4.12 Šta je problem unifikacije?

Problem unifikacije je problem ispitivanja da li postoji supstitucija koja čini dva izraza (dva terma ili dve formule) jednakim.

4.13 Kada kažemo da su izrazi unifikabilni?

Ako su e_1 i e_2 izrazi i ako postoji supstitucija σ takva da važi $e_1\sigma = e_2\sigma$, onda kažemo da su izrazi e_1 i e_2 unifikabilni i da je supstitucija σ unifikator za ta dva izraza.

4.14 Da li za dva izraza uvek postoji unifikator?

Ne.

4.15 Ukoliko za dva izraza postoji unifikator, da li on mora da bude jedinstven?

Ne.

4.16 Šta je metod rezolucije?

Metod rezolucije je metod za izvođenje zaključaka (logičkih posledica) u logici prvog reda koji se zasniva na pravilu rezolucije.

4.17 Šta je Hornova klauza i čemu ona odgovara?

Hornova klauza - disjunkcija literala sa najviše jednim nenegiranim literalom. Hornova klauza odgovara implikaciji: $(A_1 \wedge A_2 \wedge ... \wedge A_n) \Rightarrow B$ jer je to ekvivalentno sa $\neg(A_1 \wedge A_2 \wedge ... \wedge A_n) \vee B$, odnosno sa $\neg A_1 \vee \neg A_2 \vee ... \vee \neg A_n \vee B$.

4.18 Šta je supstitucija?

Supstitucija je preslikavanje koje promenljive preslikava u termove. Supstitucija se sastoji od konačnog broja pravila preslikavanja.

4.19 Kada se dva terma mogu unifikovati?

Dva terma *t* i *u* se mogu unifikovati akko postoji supstitucija σ tako da važi $t\sigma = u\sigma$.

4.20 Od čega se sastoji programiranje u Prologu?

- obezbeđivanje činjenica o objektima i odnosima među njima
- defnisanje pravila o objektima i odnosima među njima
- formiranje upita o objektima i odnosima među njima

4.21 Šta su činjenice, šta se pomoću njih opisuje?

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

4.22 Šta su pravila i šta se pomoću njih zadaje?

Pravila - Hornove klauze u punom obliku. Pravila predstavljaju opšta tvrđenja o objektima i relacijama među njima.

4.23 Šta određuju činjenice i pravila?

Činjenice i pravila se zajedničkim imenom nazivaju tvrđenja. Skup činjenica i pravila određuje *bazu znanja* (bazu podataka u širem smislu).

4.24 Šta govori pretpostavka o zatvorenosti?

Pretpostavka zatvorenosti (Closed-World Assumption) — netačno je sve što nije eksplicitno navedeno kao tačno.

4.25 Šta su upiti i čemu oni služe?

Upiti - Hornove klauze bez nenegiranog literala ($A_1, A_2, ..., A_n$). Upiti su konstrukcije Prologa preko kojih korisik komunicira sa bazom znanja.

4.26 Šta su termovi?

Termovi su osnovni gradivni elementi Prologa i širi su od pojma terma logike prvog reda. Strukture ili predikati se grade od atoma i termova. Ako je p atom, a $t_1, t_2, ..., t_n$ termovi, onda je $p(t_1, t_2, ..., t_n)$ term.

4.27 Kako se vrši unifikacija nad termima u Prologu?

Neformalno, unifikacija nad termima T i S vrši se na sledeći način:

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

4.28 Šta je lista?

Lista predstavlja niz uređenih elemenata proizvoljne dužine. Element liste može biti bilo koji term pa čak i druga lista. Lista je:

- prazna lista u oznaci []
- struktura .(G,R) gde je G ma koji term, a R lista.

4.29 Napisati pun i skraćen zapis liste od tri elementa a, b i c.

Pun zapis: .(a, .(b, .(c, [])))
Skraćen zapis: [a, b,c]

4.30 Šta omogućava metaprogramiranje?

Metaprogramiranje - kreiranje programa koji u fazi izvršavanja mogu da stvaraju ili menjaju druge programe, ili da sami sebe proširuju.

4.31 Napisati deklarativno tumačenje narednih Prolog konstrukcija:

1. p(a) (činjenica) — *p(a) je istinito*
2. p(a, X) (pravilo) — *za svako X istinito je p(a, X)*
3. p(X) :- q(X), r(X) (pravilo) — *za svako X, p(X) je istinito ako je važe q(X) i r(X)*
4. p(X) (upit) — *da li postoji vrednost promenljive X za koje važi p(X)*

4.32 Napisati proceduralno tumačenje narednih Prolog konstrukcija:

5. p(a) (činjenica) — *zadatak p(a) je izvršen*
6. p(a, X) (pravilo) — *za svako X zadatak p(a, X) je izvršen*
7. p(X) :- q(X), r(X) (pravilo) — *da bi se izvršio zadatak p(X), prvo treba da se izvrši zadatak q(X), a zatim zadatak r(X)*
8. p(X) (upit) — *izračunavanjem ostvari cilj p(X) i nađi vrednost promenljive X za koju važi svojstvo p*

4.33 Šta je stablo izvođenja i čemu ono odgovara u smislu deklarativne/proceduralne semantike?

Stablo izvođenja omogućava slikovit prikaz načina rešavanja problema u Prologu. Stablo izvođenja — stablo pretrage. U smislu deklarativne semantike, odgovara poretku primene pravila rezolucije na postojeći skup činjenica i pravila. U smislu procveduralne semantike, odgovara procesu ispunjavanja ciljeva i potciljeva.

4.34 Koji su osnovni elementi stabla izvođenja?

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

4.35 Na koji način redosled tvrđenja u bazi znanja utiče na pronalaženje rešenja u Prologu?

Redosled tvrđenja u bazi znanja utiče na efikasnost pronalaženja rešenja. Redosled može da utiče i na konačnost najlevlje grane, pa time i na to da li će rešenje uopšte biti pronađeno. Na efikasnost utiče i davanje progodnog redosleda potciljeva u pravilima.

4.36 Koja je uloga operatora sečenja?

Operator sečenja - sistemski operator koji omogućava brže izvršavanje programa i uštedu memorijskog prostora kroz eksplicitnu kontrolu backtracking-a. Ovaj operator je zapravo cilj koji uvek uspeva.

4.37 Šta je crveni, a šta zeleni operator sečenja?

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.

4.38 Da li se Hornovim klauzama mogu opisati sva tvrđenja logike prvog reda?

Hornovim klauzama ne mogu se opisati sva tvrđenja logike prvog reda.

4.39 Šta Prolog ne može da dokaže?

Prolog ne može da dokaže da neki cilj nije ispunjen.

4.40 Koje su osobine NOT operatora?

Operator NOT nije operator negacije u smislu logičke negacije. Operator NOT se naziva „negacija kao nuspeh” — NOT(cilj) uspeva ukoliko cilj ne uspeva. Zapravo, logičko NOT ne može da bude sastavni deo Prologa što je posledica oblika Hornovih klauza. To znači da, ukoliko imamo npr. naredni oblik NOT(NOT(cilj)) to ne mora da bude ekvivalentno sa *cilj*, što može da dovede do raznih problema i neočekivanih rezultata.

4.41 Da li Prolog može da obezbedi generisanje efikasnih algoritama?

Ne.

4.42 Kakva je kompatibilnost između različitih Prolog kompajlera?

Postoje nekompatibilnosti između sistema modula različitih Prolog kompajlera.

4.43 Da li je Prolog Tjuring kompletan jezik?

Da.

4.44 Kakav je sistem tipova u Prologu?

Prolog je netipiziran jezik.

5. Programiranje ograničenja

5.1 Šta je programiranje ograničenja?

Programiranje ograničenja je deklaratinvo programiranje. Programiranje ograničenja je programiranje u kojem se relacije između promenljivih zadaju u vidu ograničenja. Od sistema se zatim očekuje da izračuna rešenje tj. da izračuna vrednosti promenljivih koje zadovoljavaju data ograničenja.

5.2 Koji su predstavnici paradigme programiranja ograničenja?

B-Prolog, Ciao, GNU Prolog, Picat, SWI Prolog... Postoje različite biblioteke za programiranje ograničenja za jezike C, C++, JAVA, Phyton, za .NET platformu, Ruby...

5.3 Po čemu se razlikuje izraz x < y u imperativnoj paradigmi i paradigmi ograničenja?

x < y u imperativnoj paradigmi se evaluira u tačno ili netačno, dok u paradigmi ograničenja zadaje relaciju između objekata x i y koja mora da važi.

5.4 Od čega se sastoji programiranje ograničenja nad konačnim domenom?

- generisanje promenljivih i njihovih domena
- generisanje ograničenja nad promenljivama
- obeležavanje tj. instanciranje promenljivih

5.5 Napisati program u B-Prologu koji pronalazi sve vrednosti promenljivih X, Y i Z za koje važi da je X <= Y i X + Y <= Z pri čemu promenljive pripadaju narednim domenima X ∈ {1, 2, ..., 50}, Y ∈ {5, 10, ..100} i Z ∈ {1, 3, 5, ...99}.

```
pronadji(Vars) :- Vars = [X, Y, Z],
                  X in 1..50,
                  Y in 5..5..100,
                  Z in 1..2..99,
                  X #=< Y,
                  X+Y #=< Z,
                  labeling(Vars).
```

6. Imperativna paradigma

6.1 Pod kakvim uticajem je nastala imperativna paradigma?
Nastala je pod uticajem fon Nojmanove arhitekture računara.

6.2 Šta karakteriše imperativno programiranje?
Opisuje izračunavanje u terminima naredbi koje menjaju stanje programa.

6.3 Šta je stanje programa?
Stanje programa čine sve sačuvane informacije u datom trenutku vremena kojima program ima pristup preko memorije.

6.4 Koja je osnovna komanda za modifikaciju stanja memorije i šta ona vrši?
To je naredba dodele. Naredba dodele vrši povezivanje imena i neke vrednosti, odnosno upisivanje konkretne binarne reči na odgovarajuću memorijsku lokaciju.

6.5 Koje su faze razvoja imperativne paradigme?
Operaciona, strukturna, proceduralna, modularna faza (potparadigma).

6.6 Koje su karakteristike operacione paradigme?
Karakteriše je programiranje zasnovano na trikovima. Programi su pisani bez nekih opštih pravila. Korišćenje GOTO naredbe, što čini čitljivost koda teškim (špageti programiranje).

6.7 Koji je odnos operacione paradigme i asmeblerskih jezika?
Karakteristika imperativne paradigme je da svaka njena faza predstvalja viši nivo apstrakcije u odnosu na arhitekturu računara i udaljavanje od asemblerskog jezika, a kako je operaciona paradigma prva faza programiranja i koristi specifičnosti u radu računara, ona je bliska asemblerskim jezicima.
Apstrakcije na ovom niovu odgovaraju grupisanju asemblerskih naredbi, npr umesto niza instrukcija koji dogovaraju učitavanju vrednosti iz memorije, sabiranju i upisivanju rezultata nazad u memoriju, koristi se naredba a = b + c; koja te korake objedinjuje. Takođe, GOTO naredba je direktno odgovara asemblerskim naredbama skoka.

6.8 Koji je minimalni skup naredbi operaciona pardigme?
komanda::=

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

6.9 Šta je špageti programiranje?
To je pisanje nepreglednih programa, uglavnom preteranim/nekontrolisanim korišćenjem naredbi skoka.

6.10 Koje su karakteristike strukturne paradigme?
Svaka komanda ima jednu ulaznu i jednu izlaznu tačku. Nema nekontrolisanih skokova i koristi se mali broj naredbi za kontrolu toka. Cilj je da se proceduralni jezik više prilagodi čoveku.

6.11 Koji je minimalni skup naredbi strukturne pardigme?

komanda ::=

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

6.12 Koje su karakteristike proceduralne paradigme?

Apstrakcija kontrole toka su podrutine (funkcije, procedure, metodi, korutine, potprogrami). Podrutine predstavljaju apstrakciju niza naredbi. Poziv podrutine je poziv na apstrakciju, približava programiranje deklarativnosti. Podrutina izvršava svoje operacije u ime svog pozivaoca.

6.13 Koji je vremenski odnos strukturne i proceduralne paradigme?

Nastanak potprograma prethodi strukturnom programiranju, dakle proceduralna paradigma je nastala pre strukturne paradigme, mada se deo razvoja ove dve paradigme poklapa.

6.14 Koje vrste prenosa parametara postoje?

- *Prenos po vrednosti* - argument se evaluira i kopira se njegova vrednost u podrutinu (Algol60, Pascal, Delphi, Simula, Modula, C, C++...)
- *Prenos po referenci* - prenosi se referenca na argument, obično adresa
- *Prenos po rezultatu* - na izlasku iz podrutine vrednost parametra se prenosi pozivajućoj rutini
- *Prenos po vrednosti i rezultatu* - vrednost parametra se kopira na ulasku i na izlasku iz podrutine (Algol)
- *Prenos po imenu* - parametri se zamenjuju sa neevaluiranim izrazima (Algol, Scala)
- *Prenos po konstantnoj vrednosti* - isto kao kod prenosa po vrednosti, osim što je parametar konstantan (const u C, C++)

6.15 Kako je u memoriji organizovano izvršavanje potprograma?

U većini jezika, prostor potreban za smeštanje podataka potrebnih za izvršavanje potprograma se rezerviše na steku.

6.16 Čemu služe stek okviri i kako su organizovani?

Uvođenje steka je imalo za ideju uštedu memorije — u memoriji su podaci samo za trenutno aktivne potprograme. Omogućava korišćenje rekurzija. Kreiranje stek okvira povećava cenu poziva potprograma.

6.17 Šta su korutine?

Korutine su potprogrami tj. upravljačke strukture koje omogućavaju preskakanje stek okvira i povratak u određeni stek okvir koji nije nužno stek okvir pozivaoca potprograma.

6.18 Koje su karakteristike modularne paradigme?

Modularnost podrazumeva razbijanje većeg problema na nezavisne celine. Celine sadrže definicije srodnih podataka i funkcija. Često se moduli smeštaju u posebne datoteke, čime se postiže lakše održavanje kompleksnih sistema. Moduli mogu međusobno da komuniciraju, kroz svoje interfejse.

6.19 Šta omogućava modularna paradigma?

Moduli omogućavaju višestruku upotrebu jednom napisanog koda. Moduli se zasebno prevode i kasnije povezuju u jedinstven program.

6.20 Kako se rešavaju problemi u okviru proceduralne paradigme?

Kreiranje programa proceduralne paradigme se zasniva na principu "od opšteg ka posebnom". Polazi se od postavljenog zadatka kao opšteg. Zatim se uočavaju jednostavnije celine i zadatak dekomponuje na jednostavnije delove. Ako su ti delovi i dalje kompleksni, razbijaju se na još jednostavnije delove, dok se ne dođe do nivoa naredbi.

6.21 Šta su bočni efekti?

Bočni efekti se odnose na situacije kada se prilikom izračunavanja nekog izraza istovremeno menja i stanje memorije. Odnose se i na izmenu globalnog stanja memorije nakon izvršavanja neke funkcije.

6.22 Do čega dovode bočni efekti?

Bočni efekti mogu značajno da otežaju razumevanje imperativnih programa i da dovedu do raznih nelogičnosti.

6.23 Da li izrazi *f(x)+f(x)* i *2*f(x)* moraju imati istu vrednost u (a) imperatvinoj (b) funkcionalnoj paradigmi? Objasniti oba slučaja.

Ne moraju da imaju istu vrednost u imperativnoj paradigmi jer je tu moguće javljanje bočnih efekata. Dok u funkcionalnoj paradigmi ne može doći do bočnih efekata, pa će vrednost datih izraza uvek biti ista.

6.24 Kojoj paradigmi pripada trik programiranje? Šta su osnovne karakteristike ove vrste programiranja?

Trik programiranje pripada operacionoj paradigmi. To je programiranje zasnovano na dosetkama. Programi su pisani bez opštih pravila; korišćene su specifičnosti u radu računara i trikovi za uštedu memorije.

7. Objektno-orijentisana paradigma

7.1 Koji su principi funkcionalne dekompozicije i koji su osnovni problemi ovogo pristupa?

Postupak: podeli veliki problem u manje korake koji su potrebni da bi se problem rešio. Za baš velike probleme, podeli ih u manje potprobleme, pa onda dekomponuj manje potprobleme u odgovarajuće funkcionalne korake. Cilj je da se problem deli dok ne dođe do onog nivoa koji je jednostavan i može da se reši u nekoliko koraka, tada se ti koraci poređaju u odgovarajućem redosledu koji rešava identifikovane potprobleme i na taj način je i veliki problem rešen.
Postoje dva osnovna problema ovog pristupa:

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

7.2 Šta je osnovni uzrok problema kod rešavanja funkcionalnom dekompozicijom?

Rešavanje problema orijentisanjem na procese koji se dešavaju da bi se problem rešio ne vodi do programskih struktura koje mogu da lako reaguju na izmene: izmene u razvoju softvera obično uključuju varijacije na postojeće teme.

7.3 Zašto je uticaj izmena zahteva važan?

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

7.4 Šta je dovelo do nastanka OOP i koji su bili osnovni ciljevi koje je trebalo postići objektno orijentisanim programiranjem?

OOP je nastala kao jedna od posledica softverske krize. OOP uvodi novi način razmišljanja za pronalaženje rešenja problema. Kompleksnost softvera zahtevala je promene u stilu programiranja. Cilj je bio da se: proizvodi pouzdan softver; smanji cena proizvodnje softvera; razvijaju ponovo upotrebljivi moduli; smanje troškovi održavanja; smanji vreme razvoja softvera.

7.5 Šta je kohezija, a šta kopčanje i kako su povezani?

Kohezija je pojam koji se odnosi na to koliko blisko su povezane operacije koje se vrše u rutini ili modulu. Pojednostavljeno, ono što želimo je da svaka rutina radi samo jednu stvar, ili da se svaki modul odnosi na samo jedan zadatak.
Kopčanje se odnosi na jačinu povezanosti dve rutine ili modula.
Kopčanje je pojam komplementaran koheziji, slaba kohezija povlači jako kopčanje i jaka kohezija povlači slabo kopčanje. Želimo jaku kohezivnost i slabo kopčanje.

7.6 Šta je efekat talasanja i da li je on poželjan?
Potrebe za izmenama u okviru celog koda otežavaju debagovanje i razumevanje zadataka koje sistem obavlja. Napravimo izmenu, a neočekivano nešto drugo u sistemu ne funkcioniše — to je neželjeni propratni efekat talasanja.

7.7 Koji su bili simptomi prve softverske krize?
Simptomi softverske krize: kasne isporuke, probijanje rokova i budžeta, loš kvalitet, nezadovoljavanje potreba, slaba pouzdanost.

7.8 Šta je apstrakcija?
Apstrakcija je skup osnovnih koncepata koje neki entitet obezbeđuje sa ciljem omogućavanja rešavanja nekog problema. Apstrakcija uključuje atribute koji oslikavaju osobine entiteta kao i operacije koje oslikavaju ponašanje entiteta. Apstrakcije daju neophodne i dovoljne opise entiteta, a ne njihove implementacione detalje. Apstrakcija rezultuje u odvajanju interfejsa i implementacije.

7.9 Šta je interfejs?
Interfejs je korisnički pogled na to šta neki entitet može da uradi.

7.10 Šta implementacija?
Implementacija vodi računa o internim operacijama interfejsa koji ne moraju da budu poznati korisniku.

7.11 Objasniti odnos interfejsa i implementacije.
Interfejs govori šta entitet može da uradi, dok implementacija govori kako entitet interno radi.

7.12 Šta je enkapsulacija?
Enkapsulacija (učaurivanje) je skup mehanizama koje obezbeđuje jezik (ili skup tehnika za dizajn) za skrivanje implementacionih detalja (klase, modula ili podsistema od ostalih klasa, modula i podsistema). Enkapsulacijom su podaci zaštićeni od neželjenih spoljnih uticaja.

7.13 Koji je odnos apstrakcije i enkapsulacije?
Apstrakcija i enkapsulacija su obe vazne za uspešno ostvarivanje koncepta OOP: apstrakcija korespondira interfejsu, opštim informacijama o funkcionalnosti nekog objekta (i one su dovoljne za korisnički uvid), a enkapsulacija omogućava da se detaljne informacije - implementacija - zaštite od spoljnog sveta (i nepotrebne su krajnjem korisniku).

7.14 Šta je objekat? (filozofski, konceptulano, u objektnoj terminologiji, specifikacijski, implementaciono)
- Filozofski - entitet koji se može prepoznati
- Konceptualno - skup odgovornosti
- U terminima objektne tehnologije - apstrakcija entiteta iz stvarnog sveta
- Specifikacijski - skup metoda
- Implementaciono - podaci sa odgovarajućim funkcijama

7.15 Kako komuniciraju objekti?
Objekti međusobno komuniciraju slanjem poruka. Poruke su komande koje se šalju objektu sa ciljem da izvrši neku akciju. Poruke se sastoje od objekta koji prima poruku, metoda koji treba da se izvrši i argumenata (opciono).

7.16 Šta je klasa?
Klase su šabloni za grupe objekata. Klase daju specifikaciju za sve podatke i ponašanja objekata.

7.17 Koji je odnos klase i objekta?
Za objekat kažemo da je instanca klase.

7.18 Šta su objektno zasnovani, a šta objektno orijenitsani jezici?
Apstrakcija vođena podacima je osnovni koncept objektnih pristupa. U zavisnosti od prisutnosti ostalih koncepata, jezik može da bude obejektno-zasnovan ili objektno-orijentisan.
Objektno zasnovan jezik podržava enkapsulaciju i identitet objekta (jedinstvene osobine koje ga razdvajaju od ostalih objekata) i nema podršku za polimorfizam, nasleđivanje, komunikaciju porukama iako takve stvari mogu da se emuliraju.
Objektno orijentisani jezici imaju sve osobine objektno zasnovanih jezika, zajedno sa nasleđivanjem i polimorfizmom.

7.19 Navesti primer objektno orijentisanog jezika i primer objektno zasnovanog jezika.
Objektno-orijentisan jezik: Java, C++, C#,...
Objektno zasnovan jezik: JavaScript, VBScript,...

7.20 Koji je prvi objektni jezik i kada je nastao?
Simula, 1960.

7.21 Koji su najpopularniji objektno orijentisani jezici?
Java, C#, C++, Objective-C, Eiffel,...

7.22 Šta je nasleđivanje?
Nasleđivanje je kreiranje novih klasa (izvedenih klasa) od postojećih klasa (osnovnih ili baznih klasa). Kreiranje novih klasa omogućava postojanje hijerarhije klasa koje simuliraju koncept klasa i potklasa iz stvarnog sveta.

7.23 Na koje načine se koristi nasleđivanje? Šta je proširivanje, a šta specijalizacija?
Nasleđivanje je korisno za proširivanje i specijalizaciju.
Proširivanje koristi nasleđivanje da se razviju klase od postojećih dodavanjem novih osobina.
Specijalizacija koristi nasleđivanje da se preciznije definiše ponašanje opšte (apstraktne) klase. Potklase obezbeđuju specijalizovano ponašanje na osnovu zajedničkih elemenata koje obezbeđuje bazna klasa.

7.24 Šta omogućava nasleđivanje?
Nasleđivanje omogućava proširivanje i ponovno korišćenje postojećeg koda, bez ponavljanja i ponovnog pisanja koda. Bazne klase se mogu koristiti puno puta.

7.25 Šta je višestruko nasleđivanje?
Klasa može da se izvede kroz nasleđivanje iz više od jedne osnovne klase — višestruko nasledivanje. Instance klase koja koristi višestruko nasleđivanje ima osobine koje nasleđuje od svake bazne klase.

7.26 Koji jezici omogućavaju višestruko nasleđivanje, a koji ne?
C++, Eiffel, Objective-C omogućavaju, ali Java, Simula, SmallTalk, C# ne podržavaju višestruko nasleđivanje.

7.27 Koje su osnovne vidljivosti koje klase definišu?
Klase definišu vidljivost osobina svojim objektima: atribut ili metod može da bude javan, zaštićen ili privatan.

- *Javna* vidljivost omogućava svima da pristupe atributu ili metodu.
- *Zaštićena* vidljivost omogućava pristup samo potklasama date klase.
- *Privatna* vidljivost omogućava pristup samo u okviru same klase.

7.28 Šta je polimorfizam?
Polimorfizam — u okviru objektne terminologije, najčešće se odnosi na tretiranje objekata kao da su objekti bazne klase ali od njih dobijati ponašanje koje odgovara njihovim specifičnim potklasama. U ovom slučaju, polimorfizam se odnosi na kasno vezivanje poziva za jednu od više različitih implementacija metoda u hijerarhiji nasleđivanja.

7.29 Koja je razlika između preopterećivanja i predefinisanja?
Overloading — preopterećivanje — isto ime ili operator može da bude pridružen različitim operacijama, u zavisnosti od tipa podataka koji mu se proslede.
Overriding — predefinisanje — mogućnost različitih objekata da odgovore na iste poruke na različiti način. Ukoliko u baznoj i izvedenoj klasi imamo metod koji ima isti potpis, onda kažemo da je izvedena klasa zapravo predefinisala metod iz bazne klase.

7.30 Kada se koristi statičko, a kada dinamičko vezivanje?
Kada je moguće odlučiti za vreme prevođenja koji će se konkretan metod pozvati, onda se koristi *statičko vezivanje*, a kada to nije moguće (npr. kada nam je potreban objekat u memoriji i pokazivač na njega) onda se povezivanje vrši *dinamički* - za vreme vršenja programa. Statičko vezivanje se koristi kad imamo overriding, a dinamičko kad imamo overloading.

7.31 Šta definišu apstraktne klase?
Apstraktana kalsa je klasa koja ne može imati svoje istance. Ona definiše ponašanje koje je bitno za sve podklase. Definišu potpise metoda koje podklase treba da implementiraju, definišu metode za ponašanja koja su zajednička i definišu podatke koji su zajednički i korisni za sve potklase.

7.32 Koje su mogućnosti generičkog programiranja?
Generičko programiranje je stil programiranja u kojem se algoritmi pišu tako da se tip podataka nad kojim se algoritam primenjuje apstrahuje. To se odnosi na definisanje šablona (definisanje opštih klasa metoda koje ne zavise od tipa), koje nam omogućava višestruko korišćenje softvera.

7.33 Obrazložiti sličnosti i razlike strukturnog i OO programiranja?

Strukturno programiranje podrazumeva: top-down pristup programiranju; fokus na algoritmima i kontrolom toka; program je podeljen na određeni broj podmodela, funkcija ili procedura; funkcije su nezavisne jedna od druge; nema enkapsulacije.
OOP podrazumeva: bottom-up pristup programiranju; fokus na modelu objekta; program je organizovan na određen broj klasa i objekata; svaka klasa je organizovana u hijerarhiju; enkapsulacija pakuje kod i podatke zajedno tj. podaci i funkcionalnost se nalaze zajedno u jednom entitetu.

7.34 Koje su osnovne prednosti OO programiranja u odnosu na strukturno programiranje?

Osnovne prednosti OOP u odnosu na strukturnu paradigmu: lakše održavanje; lakša ponovna upotrebljivost koda; veća skalabilnost.

7.35 Koje su osnovne prednosti OO programiranja u odnosu na imperativno programiranje?

Lakša ponovna upotrebljivost koda; veća skalabilnost.

7.36 Koja je osnovna razlika između OOP i imperativnog programiranja?

Osnovna razlika je u pristupu problemu i organizaciji programa. Kod imperativnog to je funkcionalna dekompozicija i "top-down" pristup, i fokus je na samom algoritmu. Program specifikuje niz koraka koji se moraju obaviti da bi se došlo do rešenja, a funkcije koje se pritom koriste su nezavisne od podataka.
Kod OOP-a imamo suprotan, "bottom-up" pristup i fokus na problemu i podacima: modelira se sam problem sistemom objekata, koji integrišu i podatke i funkcije, a program se gradi interakcijama između ovih objekata.

7.37 Koje su sličnosti OOP i imperativnog programiranja?

Postojanje promeniljivih i funkcija, opisivanje algoritama. Obe paradigme su u osnovi proceduralne, tj zahtevaju opisivanje procedura i algoritama kako bi se doslo do resenja. U praksi, OOP vrlo često i samo koristi imperativni stil i obratno: imperativni često podržavaju OO koncepte.

8. Osnovna svojstva programskih jezika

8.1 Koja su osnovna svojstva programskih jezika?
Sintaksa, semantika, imena, doseg, povezanost, kontrola toka, podrutine, tipovi, kompajlirani/interpretirani jezici, izvršavanje.

8.2 Šta je leksika?
Leksika je podoblast sintakse koja se bavi opisivanjem osnovnih gradivnih elemenata jezika. U okviru leksike, definišu se reči i njihove kategorije.

8.3 Šta su lekseme, a šta tokeni?
U programskom jeziku, reči se nazivaju lekseme, a kategorije tokeni.

8.4 Šta su konteksno zavisne ključne reči?
Postoje ključne reči koje zavise od konteksta koje su ključne reči na određenim specifičnim mestima programa, ali mogu biti identifikatori na drugim mestima.

8.5 Zašto se uvode konteksno zavisne ključne reči?
Većina je uvedena revizijom postojećeg jezika sa ciljem da se definiše novi standard: sa velikim brojem korisnika i napisanog koda, vrlo je verovatno da se neka reč već koristi kao identifikator u nekom postojećem programu. Uvođenjem kontekstualne ključne reči, umesto obične ključne reči, smanjuje se rizik da se postojeći program neće kompilirati sa novom verzijom standarda.

8.6 Posao leksičkog analizatora je da ...
Dodeljuje ulaznim rečima odgovarajuće kategorije, što je bitno za dalji proces prevođenja.

8.7 Regularni izrazi ...
Definišu reči.

8.8 Konačni automati ...
Prepoznaju ispravne reči.

8.9 Šta definiše sinatksa programskog jezika?
Sintaksa programskog jezika definiše strukturu izraza, odnosno načine kombinovanja osnovnih elemenata jezika u ispravne jezičke konstrukcije.

8.10 Koji formalizam se koristi za opisivanje sintakse programskog jezika?
Kontekstno slobodne gramatike.

8.11 Šta definiše semantika programskog jezika?
Semantika pridružuje značanje ispravnim konstrukcijama na nekom programskom jeziku. Semantika programskog jezika određuje značenje jezika.

8.12 Šta nam govori neformalna semantika programskog jezika?
Uloga neformalne semantike je da programer može da razume kako se program izvršava pre njegovog pokretanja.

8.13 Šta nam omogućava formalno zadavanje semantike programskih jezika?
Formalna semantika omogućava formalno rezonovanje o svojstvima programa.

8.14 Koji su formalni okviri za definisanje semantike programskih jezika?
Formalan opis značenja jezičkih konstrukcija: operaciona semantika, denotaciona semantika, aksiomska semantika.

8.15 Operaciona semantika se najčešće koristi kao semantika za jezike ... paradigme.
Imperativne.

8.16 Denotaciona semantika se najčešće koristi kao semantika za jezike ... pradigme.
Funkcionalne.

8.17 Primer statičkih semantičkih provera je:
Da li su sve promenljive koje se koriste u izrazima definisane i da li su odgovarajućeg tipa.

8.18 Primer dinamičkih semantičkih provera je:
Deljenje nulom, pristup elementima niza van granica...

8.19 Šta je ime? Koje su osnovne karakteristike imena?
Ime — string koji se koristi za predstavljanje nečega (promenljive, konstante, operatora, tipova…).
Imena u većini programskih jezika imaju istu formu - slova, cifre i podvlaka. U nekim jezicima imena moraju da počnu specijalnim znacima. Imena su najčešće case sensitive.

8.20 Šta je promenljiva i koje su njene osnovne karakteristike?
Promenljiva - apstrakcija memorijskih jedinica (ćelija), ime promenljive je imenovanje memorijske lokacije.
Karakteristke promenljivih: ime, adresa, vrednost, tip, životni vek, doseg.

8.21 Da li svaka promenljiva ima svoje ime? Objasni.
Imena promenljivih su najčešća imena u programu, ali nemaju sve promenljive imena (tj. nemaju sve meorijske lokacije imena, na primer, memorijske lokacije eksplicitno definisane na hipu kojima se pristupa preko pokazivača).

8.22 Koji je odnos adrese i promenljive?
Adresa promenjive predstavlja adresu fizičke memorije koja je pridružena promenljivoj za skaldištenja podataka, ali promenjiva može imati različite fizičke lokacije prilikom istog izvršavanja programa.

8.23 Šta su aliasi?
Više promenljivih koje imaju istu adresu.

8.24 Šta je tip i koji su osnovni tipovi podatka?
Tip promenljive određuje opseg vrednosti koje promenljiva može da ima kao i operacije koje se mogu izvršiti za vrednosti tog tipa.
Osnovni tipovi, niske, korisnički definisani prebrojivi tipovi (enum, subrange), nizovi, asocijativni nizovi, strukture, torke, liste, unije, pokazivači i reference.

8.25 Šta je vrednost promenljive?
Vrednost promenljive je sadržaj odgovarajuće memorijske ćelije.

8.26 Šta je doseg?
Doseg određuje deo programa u kojem je vidljivo neko ime. Doseg ne određuje nužno životni vek objekta.

8.27 Šta je životni vek i čemu on obično odgovara?
Životni vek promenljive je deo vremena izvršavanja programa u kojem se garantuje da je za tu promenljivu rezervisan deo memorije i da se ta promenljiva može koristiti. On odgovara jednom od tri mehanizma skladištenja podataka (statički objekti, objekti na steku i objekti na hipu).

8.28 Šta je povezivanje?
Povezivanje uspostavlja odnos izme]u imena sa onim što to ime predstavlja.

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

8.30 Šta je kontrola toka?
Kontrola toka definiše redosled izračunavanja koje računar sprovodi da bi se ostvario neki cilj.

8.31 Koji su mehanizmi određivanja kontrole toka?
1. Sekvenca — određen redosled izvršavanja
2. Selekcija — u zavisnosti od uslova, pravi se izbor (if, switch)
3. Iteracija — fragment koda se izvršava više puta
4. Podrutine su apstrakcija kontrole toka: one dozvoljavaju da programer sakrije proizvoljno komplikovan kod iza jednostavnog interfejsa (procedure, funkcije, rutine, metodi, potprogrami)
5. Rekurzija — definisanje izraza u terminima samog izraza (direktno ili indirektno)
6. Konkurentnost — dva ili više fragmenta programa mogu da se izvršavaju u istom vremenskom intervalu, bilo paralelno na različtim procesorima, bilo isprepletano na istom procesoru

7. Podrška za rad sa izuzecima —fragment koda se izvršava očekujući da su neki uslovi ispunjeni, ukoliko se desi suportno, izvršavanje se nastavlja u okviru drugog fragmenta za obradu izuzetka
8. Nedeterminizam — redosled izvršavanja se namerno ostavlja nedefinisan, što povlači da izvršavanje proizvoljnim redosledom dovodi do korektnog rezultata.

8.32 Šta je sistem tipova i šta on uključuje?

Sistem tipova je mehanizam potreban za upravljanjem podacima. Sistem tipova obično uključuje:

- skup predefinisanih osnovnih tipova
- mehanizam gradenja novih tipova
- mehanizam kontrolisanja tipova (pravila za utvrđivanje ekvivalentnosti; pravila za utvrđivanje kompatibilnosti; pravila izvođenja)
- pravila za proveru tipova (statička i dinamička provera)

8.33 Šta je tipiziranje i kakvo tipiziranje postoji?

Jezik je tipiziran ako precizira za svaku operaciju nad kojim tipovima podataka može da se izvrši. Postoji slabo i jako tipiziranje.

8.34 Kada se radi provera tipova?

Za vreme prevođenja programa — statičko tipiziranje.
Za vreme izvršavanja programa — dinamičko tipiziranje.

8.35 Koja je razlika između jako i slabo tipiziranih jezika?

Kod jako tipiziranih jezika izvođenje operacije nad podacima pogrešnog tipa će izazvati grešku. Slabo tipizirani jezici izvršavaju implicitne konverzije ukoliko nema poklapanja tipova.

8.36 Koja je razlika između statički i dinamički tipiziranih jezika?

Razlika je kada se vrši provera tipova. Statičko tipiziranje je manje sklono greškama ali može da bude previše restriktivno, dok je dinamičko tipiziranje sklonije greškama i teško za debagovanje, ali fleksibilnije.

8.37 Koja je razlika između kompiliranja i interpretiranja?

Kompilirani jezici se prevode u mašinski kod koji se izvršava direktno na procesoru računara - faze prevođenja i izvršavanja programa su razdvojene. Jednom preveden kod se može puno puta izvršavati, ali svaka izmena izvornog koda zahteva novo prevođenje.
Interpretirani jezici se prevode naredbu po naredbu i neposredno zatim se naredba izvršava — faze prevođenja i izvršavanja nisu razdvojene već su međusobno isprepletene. Sporiji, ali prilikom malih izmena koda nije potrebno vršiti analizu celokupnog koda.

8.38 Šta je rantajm sistem?

Rantajm sistem se odnosi na skup biblioteka od kojih implementacija jezika zavisi kako bi program mogao ispravno da se izvršava.

8.39 Kakav rantajm sistem može biti? Navesti primere.

Rantajm sistem može biti malog obima, kao npr. za programski jezik C gde rantajm sistem uključuje implementaciju standardne biblioteke i učitavanje i pokretanje izvršivog programa obično putem sistemskih biblioteka, ili veoma zahtevan, kao što su to virtuelne mašine koje u potpunosti skrivaju hardver arhitekture nad kojom se program izvršava.
Primer rantajm sistema je JVM za Javu, CLR za C#, crt za C.

9. Programski jezici

9.1 Navesti tri jezika koji koriste sakupljač otpadaka.
Erlang, Java, C#, Go...

9.2 Navesti tri jezika koja su nastala pod uticajem programskog jezika Java.
Kotlin, Groovy, Scala, Clojure, C#, PHP...

9.3 Navesti tri jezika koja su nastala pod uticajem programskog jezika C++.
C#, Clojure, Java, Lua, Perl, PHP, Python...

9.4 Na koji način su povezani jezici Prolog, Erlang i Elixir?
Prolog je uticao na nastanak jezika Erlang, a Erlang je uticao na nastanak jezika Elixir.

9.5 Navesti tri jezika koji su jako tipizirani.
Erlang, Elixir, Lua, Ruby, Swift, F#, Go, Clojure, Kotlin, Java, Scala, Haskell, Python...

9.6 Navesti tri jezika koji su slabo tipizirani.
Objective-C, Perl, PHP, C, C++, C#...

9.7 Navesti tri jezika koji imaju dinamičko tipiziranje.
Erlang, Elixir, Perl, Lua, PHP, Ruby, Objective-C, Clojure, Python, MatLab...

9.8 Navesti tri jezika koji imaju statičko tipiziranje.
Swift, F#, Go, Kotlin, C, Java, C++, Haskell, Scala...

9.9 Navesti tri jezika koji se kompajliraju.
Objective-C, Swift, F#, Go, Clojure, Kotlin, Erlang, Elixir, Lua, C, C++, Haskell, Scala...

9.10 Navesti tri jezika koji se interpretiraju.
Perl, PHP, Ruby, MatLab, Smalltalk, R...

9.11 Na koji način su Java, Lisp i Clojure povezani?
Clojure je direktan potomak Lisp jezika i integrisan je sa Java jezikom i to mu omogućuje da koristi Java klase, metode i objekte.

9.12 Na koji način su C, Objective-C i Swift povezani?
Objective-C je nastao spajanjem jezika C i Smalltalk, a Swift je nastao pod uticajem jezika Objective-C.

9.13 Navesti tri jezika koja su statički tipizirana ali kod kojih tipovi prilikom pisanja programa mogu da se izostave.
Swift, F#, Go, Kotlin, Scala, Haskell...

9.14 Navesti tri jezika koja su dinamički tipizirana i kod kojih tipovi prilikom pisanja programa mogu da se izostave.
Clean, Julia, Python...

9.15 Navesti tri jezika koja odlikuje dinamička i slaba tipiziranost.
Perl, PHP, Objective-C...

9.16 Navesti tri jezika koja odlikuje dinamička i jaka tipiziranost.
Clojure, Erlang, Elixir, Lua, Ruby...

9.17 Navesti tri značajna jezika koja su nastala nakon 2010. godine.
Elixir, Swift, Kotlin...

9.18 Šta je zajedničko za jezike Lua, Perl i Ruby?
Sva tri su skript jezici; podržavaju i objektno-orijentisanu i funkcionalnu paradigmu; dinamički tipizirani.

9.19 Šta je zajedničko za jezike Kotlin, Scala i Clojure?
Jako tipizirani jezici; podržavaju funkcionalnu paradigmu; nastali su pod uticajem Jave.

9.20 Šta je zajedničko za jezike C, C++ i Go?
Pripadaju istoj familiji; statički tipizirani; kompajlirani.

9.21 Kojoj paradigmi pripadaju naredni jezici i koje dodatne paradigme podržavaju?
- *C* — imperativna, strukturna
- *C++* — proceduralna, funkcionalna, objektno-orijentisana
- *C#* — strukturna, imperativna, objektno-orijentisana, funkcionalna, konkuretna
- *Python* — funkcionalna, imperativna, objektno-orijentisana, strukturna
- *Haskell* — funkcionalna
- *Lisp* — funkcionalna, proceduralna
- *Prolog* — logičko programiranje, programiranje ograničenja
- *Java* — objektno-orijentisana, imperativna
- *JavaScript* — funkcionalna, imperativna
- *Scala* — konkuretno programiranje, funkcionalna, imperativna, objektno-orijentisana
- *Erlang* — funkcionalna, deklarativna, konkuretna, distribuirana
- *Elixir* — funkcionalna, konkurentna
- *Perl* — skript programiranje, proceduralna, funkcionalna, objektno-orijentisana
- *Lua* — skript programiranje, proceduralna, funkcionalna, objektno-orijentisana
- *PHP* — skript programiranje, funkcionalna, objektno-orijentisana
- *Ruby* — skript programiranje, imperativna, funkcionalna, objektno-orijentisana
- *Objective-C* — objektno-orijentisana, generička, imperativna
- *Swift* — objektno-orijentisana, imperativna, funkcionalna
- *F#* — funkcionalna, paralelna, imperativna, distribuirana, objektno-orijentisana
- *Go* — imperativna, konkurentna, funkcionalna
- *Clojure* — funkcionalna, konkurentna, reaktivna
- *Kotlin* — objektno-orijentisana, funkcionalna

9.22 Da li je jezik ... dinamički ili statički tipiziran?

- C (dinamički/***statički***)
- C++ (dinamički/***statički***)
- C# (dinamički/***statički***) - nove verzije su dinamičke
- Python (***dinamički***/statički)
- Haskell (dinamički/***statički***)
- Lisp (***dinamički***/statički)
- Prolog (***dinamički***/statički)
- Java (dinamički/***statički***)
- JavaScript (***dinamički***/statički)
- Scala (dinamički/***statički***)
- Erlang (***dinamički***/statički)
- Elixir (***dinamički***/statički)
- Perl (***dinamički***/statički)
- Lua (***dinamički***/statički)
- PHP (***dinamički***/statički)
- Ruby (***dinamički***/statički)
- Objective-C (***dinamički***/statički)
- Swift (dinamički/***statički***)
- F# (dinamički/***statički***)
- Go (dinamički/***statički***)
- Clojure (***dinamički***/statički)
- Kotlin (dinamički/***statički***)

9.23 Da li je jezik ... jako ili slabo tipiziran?

- C (jako/***slabo***)
- C++ (jako/***slabo***)
- C# (jako/***slabo***)
- Python (***jako***/slabo)
- Haskell (***jako***/slabo)
- Lisp (***jako***/slabo)
- Prolog (***jako***/slabo)
- Java (***jako***/slabo)
- JavaScript (jako/***slabo***)
- Scala (***jako***/slabo)
- Erlang (***jako***/slabo)
- Elixir (***jako***/slabo)
- Perl (jako/***slabo***)
- Lua (***jako***/slabo)
- PHP (jako/***slabo***)
- Ruby (***jako***/slabo)
- Objective-C (jako/***slabo***)
- Swift (***jako***/slabo)
- F# (***jako***/slabo)
- Go (***jako***/slabo)
- Clojure (***jako***/slabo)
- Kotlin (***jako***/slabo)

9.24 Da li jezik ... koristi sakupljač otpadataka?

- C (da/***ne***) - postoji biblioteka
- C++ (da/***ne***) - postoji biblioteka
- C# (***da***/ne)
- Python (***da***/ne)
- Haskell (***da***/ne)
- Lisp (***da***/ne) - prvi koji je imao
- Prolog (da/***ne***)
- Java (***da***/ne)
- JavaScript (***da***/ne)
- Scala (***da***/ne)
- Erlang (***da***/ne)
- Elixir (***da***/ne)
- Perl (da/***ne***) - Perl 5
- Lua (***da***/ne)
- PHP (da/***ne***) - ne pre verzije 5.3
- Ruby (***da***/ne)
- Objective-C (da/***ne***) - neke verzije imaju
- Swift (da/***ne***)
- F# (***da***/ne)
- Go (***da***/ne)
- Clojure (***da***/ne)
- Kotlin (***da***/ne)

9.25 Koji jezici su najviše uticali na razvoj jezika ...?

- *C* — B, Algol 68, Asembler, PL/I, Fortran...
- *C++* — Ada, Algol 68, C, ML, Simula, Smalltalk...
- *C#* — C++, Eiffel, F#, Haskell, Java, ML...
- *Python* — Ada, Algol 68, C, C++, Haskell, Java, Lisp, Perl...
- *Haskell* — Clean, FP, Miranda, Lisp, Scheme...
- *Lisp* — IPL
- *Prolog* — Planner
- *Java* — Simula, Lisp, Smalltalk, C++, Eiffel, Objective-C, Ada 83...
- *JavaScript* — C, Java, Lua, Perl, Python, Scheme...
- *Scala* — Common Lisp, Eiffel, Erlang, Haskell, Java, OCaml, Smalltalk...
- *Erlang* — Prolog, Lisp, Smalltalk...
- *Elixir* — Erlang, Clojure, Ruby...
- *Perl* — AWK, C, C++, Lisp, Basic...
- *Lua* — C++, CLU, Modula, Scheme...
- *PHP* — Perl, C, C++, Java, JavaScript...
- *Ruby* — Smalltalk, Perl, C++, Eiffel, Lisp, Python...
- *Objective-C* — Smalltalk, C
- *Swift* — Objective-C, Haskell, Ruby, Python...
- *F#* — OCaml, Haskell, Python, C#, Erlang, ML, Scala...
- *Go* — C, Limbo, Newsqueak, Pascal, Smalltalk...
- *Clojure* — Lisp, Java, C++, Erlang, Haskell, ML, Prolog, Ruby...
- *Kotlin* — Java, C#, Eiffel, Groovy, ML, Python, Scala, Swift...

9.26 Da li se jezik ... interpretira ili kompajlira?

- C (interpretira/***kompajlira***)
- C++ (interpretira/***kompajlira***)
- C# (interpretira/***kompajlira***)
- Python (***interpretira***/kompajlira)
- Haskell (interpretira/***kompajlira***)
- Lisp (interpretira/***kompajlira***) - Common Lisp
- Prolog (***interpretira***/***kompajlira***) - oba, zavisi od verzije
- Java (interpretira/***kompajlira***)
- JavaScript (***interpretira***/kompajlira)
- Scala (interpretira/***kompajlira***)
- Erlang (interpretira/***kompajlira***)
- Elixir (interpretira/***kompajlira***)
- Perl (***interpretira***/kompajlira)
- Lua (interpretira/***kompajlira***)
- PHP (***interpretira***/kompajlira)
- Ruby (***interpretira***/kompajlira)
- Objective-C (interpretira/***kompajlira***)
- Swift (interpretira/***kompajlira***)
- F# (interpretira/***kompajlira***)
- Go (interpretira/***kompajlira***)
- Clojure (interpretira/***kompajlira***)
- Kotlin (interpretira/***kompajlira***)