Konstrukcija kompilatora

1. Nastanak i namena programskih prevodioca. Veza kompilatora i programskih jezika. Izazovi u razvoju kompilatora.

Programski prevodilac je program za prevođenje programa iz jezika visokog nivoa u jezik hardvera. Postoji veliki broj različitih programskih jezika i za svaki je potrebno da postoji odgovarajući prevodilac. Postoji veliki broj različitih mašina i za svaku je potrebno da postoji odgovarajući prevodilac. Glavni pristupi implementaciji programskih jezika su **kompajleri** (**kompilatori**) i **interpretatori**. Kompajleri daju mogućnost korišćenja istog koda napisanog na višem programskom jeziku na različitim procesorima. Prvi kompajler nastaje za prvi viši programski jezik - Fortran. Budući kompajleri pratiće osnovne principe Fortrana. Kompajler iz jednog programa pravi drugi program - **ciljni kod**. Ciljni kod može biti asembler, objektni kod, mašinski kod, bajtkod... Izvršni program se nezavisno pokreće nad podacima, kako bi se dobio izlaz i može se pokretati željeni broj puta. Kompajler treba da:

- prati način na koji programeri razmišljaju (specifikacija izvornog programa treba biti bliža ljudima, njihovim govornim jezicima i iskustvu)
- omogući da je pisanje programa jednostavno
- ostvari prostor za što bolji izvorni kod
- omogući da se izbegnu greške i koriste apstrakcije
- prepoznaje i korektno prevodi samo jezički ispravne programe, tj. da pronađe greške u neispravnim programima
- generiše odgovarajući program, semantički ekvivalentan izvornom kodu
- uvek ispravno završi svoj rad, bez obzira na vrstu i broj grešaka u izvornom programu
- napravi što bolji izvršni kod (kratak, brz, kompaktan i energetski efikasan kod niskog nivoa)
- bude efikasan (posebno da bude brz)

Dodatno treba omogućiti da kompajleri, osim što izvršavaju komande na standardnim mikroprocesorima, mogu kontrolisati fizičku opremu, te automatski generisati hardver.

2. Vrste prevodioca. Osnovne karakteristike, prednosti i mane interpretatora i kompilatora. Moguće kombinacije interpretatora i kompilatora.

Glavni pristupi implementaciji programskih jezika:

- **kompajleri** (**kompilatori**): Kompajleri daju mogućnost korišćenja istog koda napisanog na višem programskom jeziku na različitim procesorima. Kompajler iz jednog programa

pravi drugi program - **ciljni kod**. Ciljni kod može biti asembler, objektni kod, mašinski kod, bajtkod... Izvršni program se nezavisno pokreće nad podacima, kako bi se dobio izlaz i može se pokretati željeni broj puta. Ovde je u pitanju **AOT (Ahead-of-Time) kompilacija**.
- **interpretatori**: Interpretatori prevode naredbu po naredbu programa. Optimizacije vrše samo na nivou svake pojedinačne naredbe. U fazi izvršavanja uvek je neophodno da imamo kod. Prevođenje se svaki put vrši ispočetka. Ne procesiraju program pre izvršavanja, pa ih karakteriše sporije izvršavanje. Koriste se često za jezike skript paradigme. Ovde je u pitanju **JIT (Just-in-Time) kompilacija**.

Prednosti i mane kompilatora:

- Detekcija grešaka unapred.
- Kvalitetniji kod (efikasniji kod sa manjim zauzećem memorije).
- Duže vreme prevođenja.
- Svaka izmena u kodu zahteva celokupno novo prevođenje.
- Svaka izmena arhitekture računara zahteva novo prevođenje.

Prednosti i mane interpretatora:

- Fleksibilnost.
- Prenosivost.
- Izmene u kodu su odmah dostupne.
- Detekcija grešaka kada se dese, ako se dese.
- Sporije izvršavanje.

Moguće je kombinovati AOT i JIT kompilaciju. Primer je JVM koja prvo koristi AOT kompilaciju da prevede Java kod u bajtkod, a zatim tokom izvršavanja primenjuje JIT kompilaciju da najčešće korišćene delove bajtkoda dodatno prevede i optimizuje u mašinski kod.

3. Struktura kompilatora.

Prednji deo kompilatora (front end) - zavisi od jezika. Vrši **leksičku**, **sintaksnu** i **semantičku analizu**. Transformiše ulazni program u **međureprezentaciju (intermediate representation - IR)** koja se koristi u srednjem delu kompajlera. Ova međureprezentacija je obično reprezentacija nižeg nivoa u odnosu na izvorni kod.

Srednji deo kompilatora (middle end) - nezavisan od jezika i ciljne arhitekture. Vrši optimizacije na međureprezentaciji koda sa ciljem da unapredi performanse i kvalitet mašinskog koda koji će kompilator proizvesti. Izvršava optimizacije koje su nezavisne od CPU arhitekture za koju je krajnji kod namenjen. Da bi izvršio kvalitetnu optimizaciju, najpre vrši analizu koda. Koristeći rezultate analize, vrši se odgovarajuca optimizacija. Izbor optimizacija koje će biti izvršene zavisi od želja korisnika i argumenata koji se zadaju

prilikom pokretanja kompilatora. Podrazumevan nivo optimizacija obuhvata optimizacije nivoa **O2**.

Zadnji deo kompilatora (back end) - zavisi od ciljne arhitekture. Osnovne faze zadnjeg dela kompilatora obuhvataju arhitekturalno specifičnu optimizaciju i generisanje koda.

4. Pretprocesiranje i linkovanje.

Osnovne faze prevođenja, osim kompilacije, obuhvataju i pretprocesiranje i linkovanje. Faza **pretprocesiranja** je pripremna faza kompilacije. Kompilator ne obrađuje tekst programa koji je napisao programer, već tekst programa koji je nastao pretprocesiranjem. Jedan od najvažnijih zadataka pretprocesora je da omogući da se izvorni kod pogodno organizuje u više ulaznih datoteka. Pretprocesor izvorni kod iz različitih datoteka objedinjava u tzv. jedinice prevođenja i prosleđuje ih kompilatoru. Pretprocesor analizira samo pretprocesorske direktive, tj. vrši samo jednostavne operacije nad tekstualnim sadržajem programa i ne koristi nikakvo znanje o samom programskom jeziku. Faza **linkovanja** je neophodna faza kako bi se na osnovu proizvoda kompilacije napravio izvršivi program. Povezivanje je proces kreiranja jedinstvene izvršne datoteke od jednog ili više objektnih modula koji su nastali ili kompilacijom izvornog koda programa ili su objektni moduli koji sadrže mašinski kod i podatke standardne ili neke nestandardne biblioteke. Pored **statičkog povezivanja**, koje se vrši nakon kompilacije, postoji i **dinamičko povezivanje**, koje se vrši tokom izvršavanja programa, tj. na njegovom početku.

5. Leksička analiza.

Leksika je podoblast sintakse koja se bavi opisivanjem osnovnih gradivnih elemenata jezika. U okviru leksike, definišu se reči i njihove kategorije. U programskom jeziku, reči se nazivaju **lekseme**, a kategorije **tokeni**. Tokenima mogu biti pridruženi i neki dodatni atributi. Dakle, pojedinačni karakteri se grupišu u nedeljive celine lekseme koje predstavljaju osnovne leksičke elemente, koji bi bili analogni rečima govornog jezika i pridružuju im se tokeni koji opisuju leksičke kategorije kojima te lekseme pripadaju. U programskom jeziku, tokeni mogu da budu identifikatori, ključne reči, operatori i slično. Identifikator ne može biti neka od ključnih reči. Postoje ključne reči koje zavise od konteksta, tj. koje su ključne reči na određenim specifičnim mestima programa, ali mogu biti identifikatori na drugim mestima. **Leksička analiza** je proces izdvajanja leksema i tokena iz niza ulaznih karaktera. Leksičku analizu vrše moduli kompilatora koji se nazivaju **leksički analizatori (lekseri, skeneri)**. Oni obično prosleđuju spisak izdvojenih leksema (i tokena kojima pripadaju) drugom modulu **(sintaksičkom analizatoru**) koji nastavlja analizu teksta programa. Leksički analizator najčešće radi na zahtev sintaksičkog analizatora tako što se sintaksički analizator obraća leksičkom analizatoru kada god mu zatreba naredni token. Token, kao skup svih mogućih leksema, opisuje se formalno pogodnim obrascima koji mogu da uključuju cifre, slova, specijalne simbole... Ti obrasci za opisivanje tokena su obično **regularni izrazi**, a mehanizam za izdvajanje leksema iz ulaznog teksta zasniva se na konačnim automatima. **Lex** je alat koji generiše leksere na programskom jeziku C, na osnovu zadatog opisa tokena u obliku regularnih izraza.

6. Sintaksička analiza.

Sintaksa definiše formalne relacije između elemenata jezika, time pružajući strukturne opise ispravnih niski jezika. Sintaksa se bavi samo formom i strukturom jezika bez bilo kakvih razmatranja u vezi sa njihovim značenjem. Sintaksom programa obično se bavi deo programskog prevodioca koji se naziva **sintaksički analizator** ili **parser**. Rezultat njegovog rada se isporučuje daljim fazama u obliku **sintaksičkog stabla**. Sintaksa jezika se obično opisuje gramatikama. Za opis sintaksičkih konstrukcija programskih jezika koriste se uglavnom **kontekstno-slobodne gramatike**. Kontekstno-slobodne gramatike su izražajniji formalizam od regularnih izraza i određene su skupom **pravila**. Svako pravilo ima levu i desnu stranu. Sa leve strane pravila nalaze se **pomoćni simboli (neterminali)**, dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo **završni simboli (terminali)**. Svakom pomoćnom simbolu pridružena je neka sintaksička kategorija. Jedan od pomoćnih simbola se smatra istaknutim i naziva se **početni simbol (aksioma)**. Na osnovu gramatike jezika formira se potisni automat na osnovu kojeg se jednostavno implementira program koji vrši sintaksičku analizu. Formiranje automata od gramatike se obično vrši automatski, uz korišćenje **generatora parsera** poput sistema Yacc, Bison ili Antlr. Za opis sintakse koriste se i određene varijacije kontekstno-slobodnih gramatika kao što su **BNF** (Bakus-Naurova forma), **EBNF** (proširena Bakus-Naurova forma) i **sintaksički dijagrami**. BNF je pogodna notacija za zapis kontekstno-slobodnih gramatika, EBNF proširuje BNF operacijama regularnih izraza, dok sintaksički dijagrami predstavljaju slikovni meta jezik za predstavljanje sintakse.

7. Semantička analiza. Formalna i neformalna semantika. Semantičke greške. Semantička upozorenja. Primeri.

Semantika pridružuje značenje ispravnim konstrukcijama na nekom programskom jeziku. Može da se opiše formalno i neformalno. Uloga **neformalne semantike** je da programer može da razume kako se program izvršava pre njegovog pokretanja. **Formalne semantike** koriste se za izgradnju alata koji se koriste za naprednu semantičku analizu softvera. Ovi alati se mogu koristiti kao dopuna semantičkoj analizi koju sprovode kompajleri. Semantička analiza treba da odbaci što više nekorektnih programa, prihvati što više korektnih programa i uradi to brzo. Na izbor semantičkih provera koje kompajler implementira utiče domen projekta, ali pre svega i efikasnost analize - kompajler mora da radi efikasno i kompleksna semantička analiza nije poželjna jer usporava proces kompilacije. Najčešće se izbor semantičkih provera može kontrolisati opcijama kompajlera. Jednostavna semantička analiza zasniva se na ispitivanju karakteristika apstraktnog sintaksnog stabla. Za dodavanje jednostavnih semantičkih provera, u okviru clang-a postoje čak tri interfejsa. Kompleksnija semantička analiza uključuje i analizu koda i grafa kontrole toka. Semantičke provere mogu generisati **semantičke greške** i **semantička upozorenja**. Primeri grešaka su upotreba nedefinisanog simbola, simboli definisani više puta u istom dosegu, greške u tipovima, greške u pozivima funkcija i prosleđivanju parametara. Tokom provere tipova proverava se da li je svaka operacija primenjena na operande odgovarajućeg tipa. U zavisnosti od jezika se u nekim slučajevima, gde je potrebno, u sintaksičko drvo umeću implicitne konverzije ili

se prijavljuje greška. Primeri upozorenja su neiskorišćena promenljiva, predlog upotrebe zagrada oko izraza i deljenje nulom.

8. Uloga međukoda i generisanje međukoda. Primer gcc.

Većina kompilatora prevodi sintaksičko stablo, provereno i dopunjeno tokom semantičke analize, u određeni **međukod (intermediate code/representation)**, koji se onda dalje analizira i optimizuje i na osnovu koga se u kasnijim fazama gradi rezultujući asemblerski i mašinski kod. Uloga međureprezentacije je u pojednostavljivanju optimizacija, mogućnost imanja više prednjih delova za isti zadnji deo, kao i više zadnjih delova iz istog prednjeg dela. Veoma je teško dizajnirati dobar IR jezik. Potrebno je balansirati potrebe visokog jezika i potrebe jezika niskog nivoa mašine za koju je izvršavanje namenjeno. Ako je nivo previsok nije moguće optimizovati neke implementacione detalje, a ako je prenizak nije moguće korisititi znanje visokog nivoa da se izvrše neke agresivne optimizacije. Kompajleri često imaju više nego jednu međureprezentaciju. Postoje različiti oblici za međureprezentaciju - linearni, grafički, troadresni i tako dalje. Najčešći oblik međureprezentacije je **troadresni kod** u kome se javljaju dodele promenljivama u kojima se sa desne strane dodele javlja najviše jedan operator. U svakoj instrukciji se navode adrese najviše dva operanda i rezultata operacije. Naredbe kontrole toka se uklanjaju i svode na uslovne i bezuslovne skokove (naredba goto). Da bi se postigla troadresnost, u međukodu se vrednost svakog podizraza smešta u novouvedenu privremenu promenljivu. Njihov broj je potencijalno neograničen, a tokom faze generisanja koda (tj. registarske alokacije) svim promenljivama se dodeljuju fizičke lokacije gde se one skladište. **GCC** koristi tri najvažnije međureprezentacije da predstavi program prilikom kompilacije:

- **GENERIC** - nezavisna od jezika. Svaki gcc podržan jezik se može prevesti na ovu međureprezentaciju.
- **GIMPLE** - troadresna međureprezentacija nastala od GENERIC tako što se svaki izraz svodi na troadresni ekvivalent. Koristi **SSA (Static Single Assignment)** - svaka promenljiva se dodeljuje samo jednom.
- **RTL (Register Transfer Language)** - međureprezentacija bliska mašinskom kodu.

9. Optimizacije međukoda. Uloga i primeri.

Optimizacija podrazumeva poboljšanje performansi koda, zadržavajući ekvivalentnost sa polaznim kodom. Optimizovani kod za iste ulaze mora da vrati iste izlaze kao i originalni kod i mora da proizvede iste sporedne efekte. Generisanje IR-a uključuje redundantnost u naš kod, a i sami programeri su lenji, pa je optimizacija neophodna. Cilj je unaprediti IR generisan prethodnim koracima da bi se bolje iskoristili resursi. Fazi optimizacije prethodi faza analize na osnovu koje se donose zaključci i sprovode optimizacije. Optimizacija se najčešće odvija na dva nivoa:

- **optimizacija međukoda** - mašinski nezavisne optimizacije tj. optimizacije koje ne uzimaju u obzir specifičnosti ciljne arhitekture.

- **optimizacija ciljnog koda** - zasniva se na detaljnom poznavanju ciljne arhitekture i asemblerskog i mašinskog jezika na kome se izražava ciljni program.

Primeri optimizacije međukoda:

- **constant folding** - konstantni izrazi se mogu izračunati.
- **constant propagation** - izbegava se upotreba promenljivih čija je vrednost konstantna.
- **strength reduction** - operacije se zamenjuju onim za koje se očekuje da će se izvršiti brže.
- **common subexpression elimination** - izbegava se vršenje istog izračunavanja više puta.
- **copy propagation** - izbegava se uvođenje promenljivih koje samo čuvaju vrednosti nekih postojećih promenljivih.
- **dead code elimination** - izračunavanja vrednosti promenljivih koje se dalje ne koriste se eliminišu.
- **optimizacija petlji** - npr. izdvajanje izračunavanja vrednosti promenljivih koje su invarijantne za tu petlju ispred same petlje.

10. Generisanje koda. Izazovi. CISC i RISC arhitekture.

Tokom generisanja koda optimizovani međukod se prevodi u završni asemblerski tj. mašinski kod. Tri osnovne faze su:

1. **faza odabira instrukcija** - određuje se kojim mašinskim instrukcijama se modeluju instrukcije troadresnog koda.
2. **faza alokacije registara** - određuje se lokacija na kojoj se svaka od promenljivih skladišti.
3. **faza raspoređivanja instrukcija** - određuje se redosled instrukcija koji doprinosi kvalitetnijem iskorišćavanju protočne obrade i paralelizacije na nivou instrukcija.

Osnovni problem generisanja koda je što je generisanje optimalnog programa za dati izvorni kod neodlučiv problem. Koriste se razne heurističke tehnike koje generišu dobar, ali ne garantuju da će izgenerisati optimalan kod. Najbitniji kriterijum za **generator koda** je da on mora da proizvede ispravan kod. Ispravnost ima specijalni značaj posebno zbog velikog broja specijalnih slučajeva sa kojima se generator koda susreće i koje mora adekvatno da obradi. Ulaz u generator koda je IR izvornog programa koji je proizveo prednji deo kompajlera, zajedno sa tablicama simbola koje se koriste za utvrđivanje run-time adresa objekata koji se koriste po imenu u okviru IR-a. Generatori koda razdvajaju IR instrukcije u **basic blocks**, koji se sastoje od sekvenci instrukcija koje se uvek izvršavaju zajedno. U okviru faza optimizacije i generisanja koda najčešće postoje višestruki prolazi kroz IR koji se izvršavaju pre finalnog generisanja ciljnog programa.

Arhitektura procesora definiše, pre svega, skup instrukcija i registara. Skup instrukcija ciljne mašine ima značajan uticaj na teškoće u konstruisanju dobrog generatora koda koji je u

stanju da proizvede mašinski kod visokog kvaliteta. Broj i uloge registara takođe imaju značajan uticaj. Najčešće arhitekture procesora su:

- **CISC** - karakteriše ih bogat skup instrukcija, koji ima za cilj da se smanje troškovi memorije za skladištenje programa. Broj instrukcija po programu se smanjuje ugradnjom više operacija u jednu instrukciju, praveći tako različite kompleksnije instrukcije. Instrukcije mogu biti različitih dužina. Uglavnom se koriste na ličnim računarima, radnim stanicama i serverima, a primer ovakvih procesora je arhitektura Intel x86. Imaju više različitih načina adresiranja, od kojih su neki veoma kompleksni. Obično nemaju veliki broj registara opšte namene.
- **RISC** - zasnivaju se na pojednostavljenom i smanjenom skupu instrukcija koji je visoko optimizovan. Zbog jednostavnosti instrukcija, potreban je manji broj tranzistora za proizvodnju procesora, pri čemu procesor instrukcije može brže izvršavati. Međutim, redukovanje skupa instrukcija umanjuje efikasnost pisanja softvera za ove procesore, što ne predstavlja problem u slučaju automatskog generisanja koda kompajlerom. Ne postoje složene instrukcije koje pristupaju memoriji, već se rad sa memorijom svodi na load i store instrukcije. Najveća prednost je **protočna obrada (pipeline)**, koja se lako može implementirati. Protočna obrada je jedna od jedinstvenih odlika arhitekture RISC, koja je postignuta preklapanjem izvršavanja nekoliko instrukcija. Zbog protočne obrade RISC arhitektura ima veliku prednost u performansama u odnosu na CISC arhitekture. RISC procesori se uglavnom koriste za aplikacije u realnom vremenu.

11. Izbor instrukcija. Izbor registara. Raspoređivanje instrukcija.

Pri **izboru instrukcija**, generator koda mora da mapira IR program u sekvencu koda koji može da bude izvršen na ciljnoj arhitekturi. Kompleksnost mapiranja zavisi od:

- Nivoa apstrakcije/preciznosti IR - ukoliko je IR visokog nivoa, generator koda može prevoditi svaku IR instrukciju u sekvencu instrukcija. Takav kod kasnije verovatno mora da se optimizuje. Ukoliko je IR niskog nivoa, onda se očekuje da takav kod bude efikasan.
- Prirode instrukcija arhitekture - uniformnost i kompletnost skupa instrukcija su bitni faktori.
- Željenog kvaliteta generisanog koda - ako nas nije briga za efikasnost ciljnog programa, odabir instrukcija je trivijalan. Neophodno je da znamo cene instrukcija kako bismo mogli da dizajniramo dobre sekvence koda.

Tokom **faze registarske alokacije** određuju se lokacije na kojima će biti skladištene vrednosti svih promenljivih koje se javljaju u međukodu. Cilj je da što više promenljivih bude skladišteno u registre procesora, međutim, to je često nemoguće, jer je broj registara ograničen i često prilično mali. Izbor registara je NP-kompletan problem. Problem korišćenja registara je obično podeljen u dva podproblema:

1. **Alokacija registara** - biraju se skupovi promenljivih koji treba da borave u registrima u svakoj tački programa.

2. **Dodela registara** - biraju se određeni konkretni registri u kojima će promenljiva boraviti.

Faza raspoređivanja instrukcija pokušava da doprinese brzini izvršavanja programa menjanjem redosleda instrukcija. Naime, nekim instrukcijama je moguće promeniti redosled izvršavanja bez promene semantike programa. Neki rasporedi instrukcija zahtevaju manji broj registara za čuvanje privremenih rezultata. Jedan od ciljeva raspoređivanja je da se upotrebe pojedinačnih promenljivih lokalizuju u kodu, čime se povećava šansa da se registri oslobode dugog čuvanja vrednosti nekih promenljivih i da se upotrebe za čuvanje većeg broja promenljivih. Izbor najboljeg redosleda je u opštem slučaju NP-kompletan problem. Najjednostavnije rešenje je ne menjati redosled instrukcija u odnosu na ono što je dao generator međukoda. Promena redosleda instrukcija može uticati na to da se protočna obrada bolje iskoristi tj. da se izbegnu čekanja i zastoji u protočnoj obradi nastali zbog zavisnosti između susednih instrukcija.

12. Projekat LLVM. Osnovne karakteristike. Najznačajniji podprojekti. Delovi kompajlera.

LLVM je modularna kompajlerska infrastruktura započeta 2000. godine na Univerzitetu Ilinois. Osnovna ideja je bila razvoj ponovo iskoristivih alata i biblioteka za statičku i dinamičku kompilaciju različitih jezika. Filozofija projekta je "svaki deo je biblioteka", pa su komponente nezavisne i mogu se koristiti zasebno. Kod je otvorenog tipa, a infrastruktura je implementirana u C++ jeziku, koristeći prednosti objektno-orijentisanog i generičkog programiranja. LLVM podržava veliki broj arhitektura (x86, ARM, MIPS, SPARC i druge) i implementira ceo tok kompilacije - od izvornog koda do mašinskog programa. Najznačajniji podprojekti su **LLVM Core libraries** i **Clang**. Core biblioteke čine jezgro sistema i sadrže moderni optimizator koji je nezavisan od izvornog jezika i ciljne arhitekture. One koriste LLVM međureprezentaciju (IR) kao univerzalni oblik koda. Clang je prednji deo kompajlera za jezike C, C++ i Objective-C. Odlikuje ga brzo kompajliranje, jasne i detaljne poruke o greškama i API koji omogućava pravljenje različitih alata, poput **Clang Static Analyzer**-a i **Clang-tidy**-a, dok **libClang** pruža interfejs za integraciju u druge projekte.

Tok rada kompajlera u LLVM-u se sastoji iz tri dela:

- **front-end (prednji deo)** - prevodi izvorni kod u LLVM međureprezentaciju (IR). Taj proces uključuje leksičku, sintaksičku i semantičku analizu. Clang je najpoznatiji front-end koji prevodi C-olike jezike u LLVM IR.
- **middle-end (srednji deo)** - radi nad generisanim IR-om i obavlja analize i optimizacije nezavisne od ciljne arhitekture. U ovoj fazi sprovode se transformacije poput eliminacije mrtvog koda i inline funkcija ili optimizacija petlji. Alat **opt** omogućava izvođenje ovih prolaza i eksperimentisanje sa optimizacijama.
- **back-end (zadnji deo)** - preuzima optimizovani IR i generiše mašinski kod za konkretnu arhitekturu. U tom delu se vrši izbor i raspored instrukcija, alokacija registara, dodavanje prologa i epiloga funkcija, kao i finalne optimizacije. Alat **llc** prevodi LLVM IR u asemblerski kod ciljne arhitekture.

LLVM međureprezentacija je centralni element cele infrastrukture i predstavlja most između prednjeg i zadnjeg dela. Može se prikazati u memoriji (tokom rada kompajlera), kao kompaktan bitcode u datotekama sa ekstenzijom *.bc* ili kao čitljiv pseudo-assembly u datotekama sa *.ll* ekstenzijom. IR je hijerarhijski organizovan: na vrhu se nalazi modul koji odgovara jednoj jedinici prevođenja i sadrži funkcije i globalne entitete. Funkcije su podeljene u osnovne blokove, a oni sadrže instrukcije. **Osnovni blok** je sekvenca instrukcija koja se izvršava linearno, sa ulazom samo na početku i izlazom samo na kraju. Instrukcije su troadresne i zasnovane na SSA formi, što značajno olakšava optimizacije.

13. Semantička analiza. Ime, doseg i tabela simbola. Operacije nad tabelom simbola.

Semantičkom analizom proverava se, na primer:

- Da li su svi identifikatori deklarisani na mestima na kojima se upotrebljavaju?
- Da li se poštuju navedeni tipovi podataka?
- Da li su odnosi nasleđivanja u objektno orijentisanim jezicima korektni?
- Da li se klase definišu samo jednom?
- Da li se metode sa istim potpisom u klasama definišu samo jednom?

Semantička analiza treba da odbaci što više nekorektnih programa, prihvati što više korektnih programa i uradi to brzo, kao i da prikupi druge korisne informacije o programu koje su potrebne za kasnije faze. Postoje dve vrste semantičke analize:

- **Scope-Checking (provera dosega)**
- **Type-Checking (provera tipova)**

Isto ime u programu može da referiše na više različitih stvari. **Doseg** nekog objekta je skup lokacija u programu gde se uvedeno ime može koristiti za imenovanje datog objekta. Uvođenje nove promenljive u doseg može da sakrije ime neke prethodne promenljive. **Tabela simbola** je preslikavanje koje za svako ime određuje čemu to ime konkretno odgovara. Kako se izvršava semantička analiza, tabela simbola se osvežava i menja. Služi za praćenje vidljivosti promenljivih. Tabela simbola je tipično implementirana kao stek mapa (kataloga). Svaka mapa odgovara jednom konkretnom dosegu. Osnovne operacije su:

- **push scope** - ulazak u novi doseg.
- **pop scope** - napuštanje dosega, izbacivanje svih deklaracija koje je doseg sadržao.
- **insert symbol** - ubacivanje novog imena u tabelu tekućeg dosega.
- **lookup symbol** - traženje čemu neko konkretno ime odgovara.

Da bi se obradio deo programa koji kreira neki doseg, potrebno je ući u doseg, dodati sve deklarisane promenljive u tabelu simbola, obraditi telo bloka/funkcije/klase i izaći iz dosega.

14. Doseg i tabela simbola u OOP. Određivanje dosega kod nasleđivanja. Razrešavanje višeznačnosti.

Doseg nekog objekta je skup lokacija u programu gde se uvedeno ime može koristiti za imenovanje datog objekta. **Tabela simbola** je preslikavanje koje za svako ime određuje čemu to ime konkretno odgovara. U OOP, doseg izvedene klase obično čuva link na doseg njene bazne klase (npr. `super` u C++). Traženje polja klase prolazi kroz lanac dosega i zaustavlja se kada se pronađe odgovarajući identifikator ili kada se pojavi semantička greška. Kod nasleđivanja je potrebno održavati još jednu tabelu pokazivača koja pokazuje na stek dosega. Kada se traži vrednost u okviru specifičnog dosega, pretraga počinje od tog konkretnog dosega. Doseg trenutne klase se može direktno referisati (npr. `this` u C++). Neki jezici omogućavaju skakanje do proizvoljne bazne klase (npr. C++). Pojednostavljena pravila dosega za C++:

- U okviru klase, pretraži celu hijerarhiju da pronađeš koji skupovi imena se tu mogu naći, koristeći standardnu pretragu dosega.
- Ako se pronađe samo jedno odgovarajuće ime, pretraga je završena bez dvosmislenosti.
- Ako se pronađe više od jednog odgovarajućeg imena, pretraga je dvosmislena i mora se zahtevati razrešavanje višeznačnosti.
- U suprotnom, počni ponovo pretragu, ali van klase.

15. Pravila za određivanje tipova u izrazima.

Notacija tipova zavisi od programskog jezika, ali uvek uključuje skup vrednosti i skup operacija nad tim vrednostima. Greška u radu sa tipovima javlja se kada se primenjuje operacija nad vrednostima koje ne podržavaju tu operaciju. Postoje tri osnovna pristupa proveri tipova:

- **Statička provera tipova** - analizira program u fazi kompilacije kako bi se pokazalo da nema grešaka u tipovima. Cilj je da se nikada ne dozvoli da dođe do loših situacija u fazi izvršavanja.
- **Dinamička provera tipova** - proverava operacije u fazi izvršavanja, neposredno pre nego što se one zaista izvrše. Preciznija je od statičke, ali je manje efikasna.
- **Bez analize tipova** - program se izvršava bez ikakve provere, pa greške mogu nastati tek pri izvršavanju.

Pravila koja definišu šta je dozvoljena operacija nad tipovima formiraju **sistem tipova**. U **jako tipiziranim jezicima** svi tipovi moraju da se poklapaju, dok se u **slabo tipiziranim jezicima** greške u tipovima mogu javiti u fazi izvršavanja. Jako tipizirani jezici su robusniji, ali slabo tipizirani su obično brži.

Statičko zaključivanje tipova se zasniva na dva osnovna koraka:

1. Zaključivanje tipa za svaki izraz na osnovu tipova njegovih komponenti.
2. Potvrđivanje da se tipovi izraza u datom kontekstu poklapaju sa očekivanim.

Ova dva koraka se u praksi često kombinuju u jednom prolazu kroz program. Sam proces podseća na logičko zaključivanje: polazi se od skupa aksioma i primenjuju se pravila zaključivanja da bi se odredio tip izraza. Zbog toga se mnogi sistemi tipova mogu posmatrati kao sistemi za dokazivanje. Primeri jednostavnih pravila zaključivanja:

- Ako je x promenljiva koja ima tip t, tada i izraz x ima tip t.
- Ako je e celobrojna konstanta, tada e ima tip int.
- Ako izrazi e_1 i e_2 imaju tipove int i int, tada i izraz $e_1 + e_2$ ima tip int.

Ovakva pravila se formalizuju pomoću zapisa u obliku $\frac{\text{Preduslov}}{\text{Postuslov}}$. Ako je preduslov tačan, možemo da zaključimo postuslov. Primeri:

- **Početne aksiome**:

$$\frac{}{\vdash true : bool}, \quad \frac{}{\vdash false : bool}$$

- **Jednostavna pravila**:

$$\frac{i \text{ is an integer constant}}{\vdash i : int}, \quad \frac{s \text{ is a string constant}}{\vdash s : string}, \quad \frac{d \text{ is a double constant}}{\vdash d : double}$$

- **Složenija pravila**:

$$\frac{\vdash e_1 : int, \ \vdash e_2 : int}{\vdash e_1 + e_2 : int}, \quad \frac{\vdash e_1 : double, \ \vdash e_2 : double}{\vdash e_1 + e_2 : double}$$

Ovakve definicije su precizne i nezavisne od konkretne implementacije. One daju maksimalnu fleksibilnost, jer provera tipova može biti implementirana na bilo koji način, sve dok se poštuju zadati principi. Zbog toga pravila omogućavaju i **formalno dokazivanje ispravnosti programa**. Potrebno je dodatno ojačati pravila zaključivanja tako da pamte u kojim situacijama su rezultati validni. To se postiže uvođenjem **konteksta (dosega)**: $S \vdash e : T$. Ovo znači da u dosegu S izraz e ima tip T. Na taj način tipove dokazujemo relativno u odnosu na doseg u kojem se nalazimo.

16. Tipovi i nasleđivanje. Tip null.

Da bi se model OOP jezika formalno opisao, potrebno je uvesti nasleđivanje u sistem tipova i uzeti u obzir oblik **hijerarhije klasa**. Osobine nasleđivanja i konvertovanja:

- **Refleksivnost (nasleđivanje):** svaki tip nasleđuje samog sebe.
- **Tranzitivnost (nasleđivanje):** ako A nasleđuje B i B nasleđuje C, onda i A nasleđuje C.
- **Antisimetričnost (nasleđivanje):** ako A nasleđuje B i B nasleđuje A, onda su A i B istog tipa.
- Ako A nasleđuje B, objekat tipa A se može pretvoriti u objekat tipa B ($A \leq B$).
- **Refleksivnost (konvertovanje):** svaki tip se može pretvoriti u samog sebe ($A \leq A$).
- **Tranzitivnost (konvertovanje):** ako se A može pretvoriti u B i B u C, onda se i A može

pretvoriti u C ($A \leq B$ i $B \leq C$ povlači $A \leq C$).
- **Antisimetričnost (konvertovanje):** ako se A može pretvoriti u B i B u A, onda su A i B istog tipa ($A \leq B$ i $B \leq A$ povlači $A = B$).
- Ako je A osnovni tip ili niz, A se može pretvoriti samo u samog sebe.

U sistem tipova se uvodi poseban tip koji odgovara literalu **null**. Definiše se pravilo:

$$\text{null} \leq A, \text{ za svaki klasni tip } A$$

Ovaj tip se obično ne vidi u samom programskom jeziku (nije dostupan programerima), već se koristi interno u okviru sistema tipova radi formalnog definisanja pravila konverzije.

17. Pravila za određivanje tipova u naredbama.

Da bismo proširili sistem tipova i modelovali naredbe, definiše se pravilo **dobro formirane naredbe (well formed)**: kažemo da je $S \vdash WF(\text{stmt})$ ako je naredba $stmt$ dobro formirana u dosegu S. Sistem tipova je zadovoljen ako za svaku funkciju f sa telom B koje je u dosegu S, možemo pokazati da važi $S \vdash WF(B)$. Pravila:

- **Pravilo za break** - naredba `break` je validna samo ako se nalazi unutar petlje.

$$\frac{S \text{ is in a for or while loop}}{S \vdash WF(\text{break})}$$

- **Pravilo za petlje** - petlja se smatra dobro formiranom ako je uslov (`expr`) logički izraz tipa `bool` i ako je telo petlje (`stmt`) takođe dobro formirano u svom lokalnom dosegu.

$$\frac{S \vdash expr : bool,\ S' \text{ is the scope inside the loop},\ S' \vdash WF(\text{stmt})}{S \vdash WF(\text{while (expr) stmt})}$$

- **Pravilo za blokovske naredbe** - blok naredbi formira novi doseg dodavanjem lokalnih deklaracija (`decls`) na postojeći doseg. Sve naredbe unutar bloka moraju biti dobro formirane u tom novom dosegu.

$$\frac{S' \text{ is the scope formed by adding decls to S},\ S' \vdash WF(\text{stmt})}{S \vdash WF(\{\text{decls stmt}\})}$$

- **Pravilo za return** - ako funkcija vraća tip T, izraz u `return expr;` mora biti kompatibilan sa tipom T. Ako funkcija vraća `void` , može se koristiti samo `return;` .

$$\frac{S \text{ is in a function returning } T,\ S \vdash \text{expr} : T',\ T' \leq T}{S \vdash WF(\text{return expr;})}, \quad \frac{S \text{ is in a function returning void}}{S \vdash WF(\text{return;})}$$

18. Tipovi i propagiranje greške.

Da bismo proverili da li je program ispravno formiran, prolazimo **rekurzivno kroz AST stablo**. Za svaku naredbu proveravamo tipove svih podizraza koje sadrži. Ako neki izraz nema odgovarajući tip ili je pogrešno tipiziran, prijavljuje se greška. **Statička greška tipova** se dešava kada ne možemo da dokažemo da izraz ima odgovarajući tip. Greške u tipovima

se lako propagiraju kroz program. U sistem tipova uvodimo novi tip koji predstavlja grešku. Ovaj tip je manji od svih drugih tipova i označava se sa $\bot$ (nekad se naziva i **bottom tip**). Pravila zaključivanja se unapređuju tako da uključuju ovaj tip. Po definiciji važi

$$\bot \ \leq A, \text{ za svaki tip } A$$

U semantičkom analizatoru potrebno je omogućiti neku vrstu **oporavka od greške**, kako bi analiza mogla da se nastavi. Jedan od pristupa je korišćenje tipa `Error`. Kada detektujemo tip `Error`, tretiramo ga kao da je dokazano da izraz ima tip $\bot$. Postoje i drugi slučajevi koje treba rešiti kao što su poziv nepostojeće funkcije i deklaracija promenljive neispravnog tipa. Ne postoje striktno ispravni i pogrešni odgovori na pitanja oporavka od grešaka, postoje samo bolji i lošiji izbori.

19. Preopterećivanje funkcija.

Preopterećenje funkcija znači imati više funkcija sa istim imenom, ali različitim argumentima. Tokom kompilacije, analizom tipova argumenata određuje se koja funkcija treba da se pozove. Ako se ne može odrediti funkcija, kompajler prijavljuje grešku. Proces izbora funkcije počinje sa skupom svih preopterećenih funkcija. Prvo filtriramo funkcije koje se ne poklapaju sa pozivom i tako dobijamo skup kandidata:

- Ako je skup prazan, prijavljuje se greška.
- Ako skup sadrži samo jednu funkciju, bira se ona.
- Ako skup sadrži više funkcija, bira se najbolja.

Ako postoji više kandidata, bira se funkcija koja **najspecifičnije odgovara** pozivu. Formalno, za dve funkcije A i B sa argumentima $(A_1, \ \ldots, \ A_n)$ i $(B_1, \ \ldots, \ B_n)$ kažemo da je $(A <: B)$ ako važi $(A_i \leq B_i)$ za svako i. Relacija $<:$ je **parcijalno uređenje**. Funkcija A je **najbolji izbor** ako za svaku drugu funkciju B važi $(A <: B)$, odnosno ako je bar onoliko dobra koliko i svaki drugi kandidat. Ako postoji najbolji izbor, bira se on, a u suprotnom poziv je **višesmislen** i potrebno ga je razrešiti.

Variadic funkcije prihvataju proizvoljan broj argumenata. Postoje dve strategije ako imamo variadic funkcije:

1. Smatrati poziv višesmislenim ako se uklapa u više variadic funkcija.
2. Smatrati boljom funkciju koja nije variadic, jer je specifičnije dizajnirana za konkretan skup argumenata.

Hijerarhijsko preopterećenje podrazumeva pravljenje hijerarhije kandidata:

- Počni sa najnižim nivoom hijerarhije i traži poklapanje.
- Ako postoji jedinstveno poklapanje, bira se ta funkcija.
- Ako postoji više poklapanja, prijavljuje se višesmislenost.
- Ako nema poklapanja, prelazi se na naredni nivo hijerarhije.

Hijerarhijsko preopterećenje konceptualno je slično radu sa dosezima.

20. Izvršno okruženje i podaci. Enkodiranje osnovnih tipova, nizova i višedimenzionih nizova.

Izvršno okruženje predstavlja skup struktura podataka koje se održavaju u fazi izvršavanja, sa ciljem omogućavanja koncepata jezika visokog nivoa. Ono zavisi od osobina izvornog i ciljnog jezika. Enkodiranje osnovnih tipova:

- **Celobrojni tipovi** (byte, char, short, int, long, unsigned i sl.) se obično preslikavaju direktno u odgovarajuće mašinske tipove. Veličina zavisi od arhitekture računara.
- **Realni tipovi** (float, double, long double) se takođe obično preslikavaju u odgovarajuće mašinske tipove.
- **Pokazivači** se obično implementiraju kao celobrojni tip koji čuva memorijske adrese.

Enkodiranje nizova:

- **C stil** - elementi su smešteni jedan za drugim u memoriji.
- **Java stil** - elementi su jedan za drugim, ali prvi element predstavlja broj elemenata niza.
- **D stil** - elementi su jedan za drugim, s tim što svaki element čuva pokazivač na prvi element i na element posle poslednjeg elementa.

Višedimenzioni nizovi se obično predstavljaju kao nizovi nizova. Tačan oblik zavisi od načina na koji su nizovi implementirani.

21. Izvršno okruženje i funkcije. Aktivaciono stablo. Zatvorenja i korutine. Stek izvršavanja.

Pozivi **funkcija** se obično implementiraju korišćenjem **steka aktivacionih slogova**. Kada se pozove funkcija, na stek se gura novi aktivacioni slog, a povratak iz funkcije uklanja aktivacioni slog sa vrha steka. **Aktivaciono stablo** predstavlja strukturu koja opisuje sve pozive funkcija tokom konkretnog izvršavanja programa. Ono zavisi od ponašanja programa i ne može se uvek odrediti u fazi kompilacije. Statički ekvivalent aktivacionog stabla je graf poziva funkcija. Svaki aktivacioni slog čuva kontrolni link prema aktivacionom slogu koji ga je inicirao. **Kontrolni link funkcije** je pokazivač na funkciju koja ju je pozvala i koristi se da bi se odredilo gde se izvršavanje nastavlja kada funkcija završi sa radom. Aktivaciono stablo se često naziva i **špageti stek**. Mogućnost optimizacije špageti steka zasniva se na određenim pretpostavkama: jednom kada se funkcija vrati, njen aktivacioni slog ne može da bude ponovo referenciran, i svaki aktivacioni slog je ili završio izvršavanje ili je predak tekućeg aktivacionog sloga. Međutim, ove pretpostavke ne moraju uvek da važe. Prva pretpostavka ne važi za zatvorenja. **Zatvorenje** predstavlja ugnježdenu funkciju u okviru funkcije F, koja ima pristup slobodnim promenljivama iz funkcije F iako je F završila sa svojim izvršavanjem. Osnovne karakteristike zatvorenja su da je ugnježdena funkcija, da ima pristup slobodnim promenljivama iz spoljašnjeg dosega i da može biti vraćena kao povratna vrednost funkcije u koju je ugnježdena. Jezici koji podržavaju zatvorenja obično nemaju klasičan stek izvršavanja. **Pristupni link** je pokazivač prema aktivacionom slogu u kojem je funkcija kreirana i koriste ga ugnježdene funkcije da bi pristupile promenljivama u spoljašnjem

dosegu. Druga pretpostavka o optimizaciji špageti steka ne važi za korutine. **Korutine** su funkcije koje, kada se pozovu, obave deo posla i vrate kontrolu pozivajućoj funkciji, ali kasnije mogu da nastave rad od istog mesta. Zbog toga se često ne mogu implementirati korišćenjem steka izvršavanja. Svaki aktivacioni slog u steku izvršavanja mora da čuva sve svoje parametre, sve lokalne promenljive i sve privremene promenljive uvedene od strane IR generatora. Kod IR pozivne konvencije, pozivaoc je odgovoran za stavljanje na stek i skidanje sa steka prostora za argumente pozvane funkcije, dok je sama pozvana funkcija odgovorna za stavljanje i skidanje sa steka svojih lokalnih i privremenih promenljivih. Najčešći pristupi prenosu parametara su **call-by-value** i **call-by-reference**.

22. Izvršno okruženje i objekti. Strukture, objekti i nasleđivanje.

Implementacija **objekata** je veoma složena i teško je napraviti istovremeno izražajan i efikasan objektno-orijentisan jezik. Koncepti koje je naročito teško efikasno implementirati su dinamičko raspoređivanje, interfejsi, višestruko nasleđivanje i dinamička provera tipova.

Struktura predstavlja tip koji sadrži kolekciju imenovanih vrednosti. Najčešći pristup u implementaciji je da se svako polje postavi u memoriju redom kojim je deklarisano. Kada se objekat smesti u memoriju, on je samo serija bajtova, pa je neophodno znati gde se nalazi svako polje. Da bi se to omogućilo, kompajler održava internu tabelu sa pomerajima za svako polje. Pristup konkretnom polju se ostvaruje tako što se krene od osnovne adrese objekta i ona se pomeri unapred za odgovarajući pomeraj.

Kod jednostrukog **nasleđivanja**, izgled memorije za izvedenu klasu D je određen tako što se najpre zauzima prostor za baznu klasu B, a zatim sledi prostor za preostale članove klase D. Na ovaj način, pokazivač na objekat bazne klase B koji zapravo pokazuje na objekat izvedene klase D i dalje vidi bazni deo objekta na početku memorije. Operacije koje se izvode nad objektom D kroz pokazivač na objekat B su zbog toga garantovano ispravne, bez potrebe za dodatnim proverama na šta tačno pokazivač dinamički ukazuje.

23. Izvršno okruženje i funkcije članice klasa. Pokazivač this i dinamičko određivanje poziva. Tabela virtuelnih funkcija.

Funkcije članice klasa su u osnovi iste kao i obične funkcije, ali sadrže dve dodatne komplikacije: kako znati za koji objekat je funkcija vezana i kako u fazi izvršavanja znati koju funkciju pozvati. U okviru funkcije članice klase koristi se ime ***this*** za tekući objekat, a ta informacija mora biti prosleđena funkciji. Ideja je da se *this* tretira kao implicitni prvi parametar funkcije, pa svaka funkcija koja ima n argumenata u stvarnosti ima $n + 1$ argument, pri čemu je prvi parametar pokazivač *this*. Prilikom generisanja koda, za poziv funkcije članice prosleđuje se dodatni parametar *this* koji predstavlja odgovarajući objekat. U okviru same funkcije, *this* se tretira kao običan parametar. Kada se implicitno referiše na neko polje objekta, koristi se upravo ovaj parametar da bi se odredilo u kom objektu treba tražiti traženo polje. **Dinamičko određivanje poziva** podrazumeva da se izbor funkcije koja će biti pozvana vrši u fazi izvršavanja, i to u zavisnosti od tekućeg dinamičkog tipa objekta. Jednostavna ideja bila bi da se u fazi kompilacije napravi lista svih klasa i generiše

odgovarajući kod, ali ovaj pristup ima ozbiljne nedostatke: veoma je spor i često neostvariv. Umesto toga, koristi se drugačiji mehanizam, u kojem se za funkcije pravi rešenje slično onom koje se koristi za podatke.

Tabela virtuelnih funkcija (vtable) predstavlja niz pokazivača na implementacije funkcija članica neke klase. Da bi se pozvala funkcija članica klase, potrebno je odrediti statički indeks u virtuelnoj tabeli, zatim pratiti pokazivač koji se nalazi na tom indeksu u okviru vtable objekta da bi se došlo do koda funkcije, a potom pozvati samu funkciju. Prednost ovog pristupa je što se vreme potrebno da se odredi koja funkcija će biti pozvana svodi na $O(1)$. Mana je to što su objekti veći jer svaki objekat mora da čuva $O(M)$ pokazivača, gde je M broj funkcija članica, što usporava njihovo kreiranje. Drugi način je da se kreira jedinstvena instanca vtable za svaku klasu, a svaki objekat čuva samo pokazivač na svoju vtable. Na taj način, objekat može u konstantnom vremenu da pristupi pokazivaču na tabelu i u konstantnom vremenu da pronađe odgovarajući indeks. Cena ovog rešenja je povećanje veličine objekta za $O(1)$ umesto $O(M)$, a upravo se ovaj mehanizam koristi u većini C++ i Java implementacija. Međutim, ovakav pristup podrazumeva implicitne pretpostavke o jeziku, pre svega da su svi metodi poznati statički i da postoji jednostruko nasleđivanje. Zbog toga neki jezici, kao što je PHP, ne mogu da koriste virtuelne tabele.

24. Implementiranje dinamičkih provera tipova.

Standardan način za **dinamičku proveru tipova** u jezicima sa objektno-orijentisanim mehanizmima zasniva se na virtuelnim tabelama. U svaku tabelu se dodaje i pokazivač na roditeljsku klasu. Kada se u fazi izvršavanja proverava da li objekat tipa A može da se pretvori u tip S, prate se pokazivači unapred u hijerarhiji: kreće se od tipa objekta i ide se kroz njegove roditelje sve dok se ne naiđe na tip S ili dok se ne stigne do korena hijerarhije. Ova provera je korektna, ali traje $O(d)$, gde je d dubina hijerarhije klasa. Jedna od ideja je da se omogući provera u vremenu $O(1)$. To je moguće ako je broj klasa u hijerarhiji konstantan i ne previše veliki, i ako su svi tipovi poznati statički. Ključna ideja se oslanja na činjenicu da je objekat koji je statički tipa A u izvršavanju tipa B samo ako je A pretvoriv u B, odnosno ako važi relacija $A \leq B$ u hijerarhiji. Jedna zanimljiva tehnika za implementaciju ovoga koristi proste brojeve. Svakoj klasi se dodeljuje jedinstven prost broj. **Ključ klase** se definiše kao proizvod njenog prostog broja i svih prostih brojeva njenih nadklasa. Ako klasa D nasleđuje C, C nasleđuje B, a B nasleđuje A, ključ klase D će biti proizvod prostih brojeva koji odgovaraju A, B, C i D. Kada se u fazi izvršavanja želi proveriti da li je objekat tipa O pretvoriv u tip T, dovoljno je proveriti deljivost: ako je ključ tipa O deljiv ključem tipa T, onda je pretvaranje moguće. Pošto je deljenje nad celobrojnim vrednostima operacija koja traje $O(1)$, cela provera postaje konstantna po vremenu. Ograničenje ove metode je očigledno: proizvodi prostih brojeva veoma brzo rastu, pa se mogu javiti problemi sa prekoračenjem ako se koristi običan integer tip. To znači da je ideja praktična samo dok broj klasa i visina hijerarhije nisu veliki. U suprotnom je potrebno koristiti veće tipove brojeva ili potpuno drugačiji mehanizam.

25. Troadresni kod. Aritmetičke i bulovske operacije. Kontrola toka.

Troadresni kod predstavlja jednu od standardnih međureprezentacija visokog nivoa u kompajlerima. Karakteriše ga to da svaka instrukcija ima najviše tri operanda, što ga čini dovoljno jednostavnim za obradu, a istovremeno dovoljno izražajnim da opiše gotovo sve konstrukcije jezika. U trenutku generisanja ovog koda kompajler se još ne bavi optimizacijom, već isključivo prevodi složenije izraze u niz jednostavnijih instrukcija. Kada izraz ima više od tri podizraza, uvode se privremene promenljive koje čuvaju međurezultate i omogućavaju dalje računanje. U okviru **aritmetičkih operacija** podržani su osnovni operatori sabiranja, oduzimanja, množenja, deljenja i ostatka pri deljenju ($+$, $-$, $*$, $/$, $\%$). **Bulovske promenljive** se u praksi predstavljaju kao celobrojne vrednosti, gde je nula interpretirana kao $false$, a bilo koja nenula vrednost kao $true$. Pored aritmetičkih operatora, troadresni kod podržava i **relacione** i **logičke operatore**, kao što su $<$, $==$ i $\&\&$, čime se omogućava izražavanje uslova i kombinovanje logičkih izraza. **Kontrola toka** se u troadresnom kodu ostvaruje pomoću **imenovanih labela** koje označavaju pojedine delove koda na koje se može skočiti. Instrukcije koje omogućavaju ovakvu kontrolu toka uključuju bezuslovne i uslovne skokove. **Bezuslovni skok** se označava kao `goto label` i uvek prenosi tok izvršavanja na zadatu labelu. **Uslovni skokovi** se realizuju konstrukcijom `ifZ value goto label`, što znači da će se skok izvršiti ako je vrednost izraza jednaka nuli. Na ovaj način se gradi osnova za implementaciju grananja, petlji i ostalih kontrolnih struktura višeg nivoa.

26. Troadresni kod. Funkcije i stek okviri.

Funkcije u troadresnom kodu sastoje se iz četiri osnovna dela. Prvi je **labela** koja označava početak funkcije. Drugi je instrukcija `BeginFunc N;`, koja rezerviše N bajtova za lokalne i privremene promenljive. Treći deo je **telo funkcije**, odnosno niz instrukcija koje definišu njeno ponašanje. Četvrti deo je instrukcija `EndFunc;`, koja obeležava kraj funkcije. Kada se dođe do ove instrukcije, ona je zadužena da očisti stek okvir i vrati kontrolu na odgovarajuće mesto, oslobađajući prostor koji je ranije rezervisan. Odgovornost za smeštanje argumenata funkcije na stek leži na pozivaocu funkcije. S druge strane, pozvana funkcija je odgovorna za raspored svojih lokalnih i privremenih promenljivih unutar stek okvira. Svaki argument se postavlja na stek korišćenjem instrukcije `PushParam var;`, dok se prostor oslobađa od strane pozivaoca instrukcijom `PopParams N;`. Interno, procesor koristi specijalni registar, **program counter (PC)**, koji čuva adresu naredne instrukcije koja treba da se izvrši. Kada funkcija završi sa radom, PC se podešava tako da se izvršavanje nastavi na mestu gde je ono bilo prekinuto. Da bi stek mogao da funkcioniše ispravno, potrebno je znati gde se nalaze lokalne promenljive, parametri i privremene promenljive. Oni se čuvaju u odnosu na **frame pointer (fp)** - registar koji pokazuje na početak tekućeg aktivacionog okvira funkcije. Parametri počinju od adrese `fp + 4` i rastu naviše, dok lokalne i privremene promenljive počinju od adrese `fp - 8` i rastu naniže. Globalne promenljive zauzimaju prostor počevši od adrese `gp + 0` i rastu naviše, gde je gp **global pointer** - registar koji pokazuje na bazu globalnih podataka programa.

27. Algoritam generisanja troadresnog koda.

U ovom stadijumu kompilacije kompajler već ima na raspolaganju **AST stablo** koje je anotirano sa informacijama o dosegu i tipovima promenljivih. Da bi se generisao **troadresni kod (TAC)**, potrebno je rekurzivno obići AST: za svaki podizraz ili podnaredbu generiše se TAC, a zatim se korišćenjem generisanog TAC-a za podizraze i podnaredbe kreira TAC za kompletne izraze i naredbe. Generisanje TAC-a za izraze realizuje se kroz funkciju `cgen(expr)`, koja za dati izraz generiše odgovarajući troadresni kod, čuva vrednost izraza u privremenoj promenljivoj i vraća ime te promenljive. Za **atomičke izraze**, `cgen(expr)` se primenjuje direktno, dok se za **složene izraze** funkcija primenjuje rekurzivno, kombinujući TAC za podizraze kako bi se dobio TAC za ceo izraz. Generisanje TAC-a za naredbe može se ostvariti proširenjem funkcije `cgen` tako da prihvata i naredbe. Za razliku od izraza, `cgen` za naredbe ne vraća ime privremene promenljive, jer se vrednost naredbe obično ne čuva, već se fokusira na generisanje odgovarajućih instrukcija koje menjaju stanje programa.

28. Optimizacije međukoda. Graf kontrole toka.

Termin **optimizacija** označava proces traženja što efikasnijeg koda za dati program. Cilj optimizacije je poboljšanje koda generisanog u prethodnom koraku, što predstavlja najvažniji i najkompleksniji deo modernog kompajlera. Razlozi za optimizaciju su višestruki: prilikom generisanja međukoda često se uvodi redundantnost, a programeri retko razmišljaju o efikasnosti koda. U opštem slučaju, problem pronalaženja optimalnog koda je neodlučiv, pa je praktični cilj optimizatora poboljšanje međureprezentacije, a ne nužno postizanje globalnog optimuma. Karakteristike dobrog optimizatora uključuju očuvanje ponašanja programa, proizvodnju efikasnijeg koda i relativno nisku potrošnju vremena. Čak i najbolji optimizatori ponekad mogu uvesti greške ili propustiti jednostavne optimizacije. Optimizatori mogu ciljati različite aspekte koda, zavisno od željenih osobina programa. Najčešći ciljevi optimizacije uključuju:

- **vreme izvršavanja** - program treba da radi što brže, često na račun memorije ili energije
- **upotreba memorije** - smanjenje veličine programa, ponekad na račun brzine izvršavanja
- **upotreba energije** - smanjenje potrošnje energije biranjem jednostavnih instrukcija, što može uticati na brzinu i veličinu koda

Graf kontrole toka (Control Flow Graph - CFG) predstavlja graf koji kao čvorove sadrži osnovne blokove funkcije. Svaka grana iz jednog osnovnog bloka do drugog označava moguć tok izvršavanja sa kraja prvog bloka na početak drugog. Posebni čvorovi označavaju početak i kraj funkcije. Vrste optimizacija prema opsegu delovanja uključuju:

- **lokalna optimizacija** - primenjuje se unutar jednog osnovnog bloka
- **globalna optimizacija** - funkcioniše na nivou celog grafa kontrole toka funkcije
- **međuproceduralna optimizacija** - uzima u obzir celokupan graf kontrole toka, uključujući pozive funkcija i njihovu međusobnu interakciju.

29. Lokalne optimizacije međukoda. Eliminacija zajedničkih podizraza. Prenos kopiranja. Eliminacija mrtvog koda.

Lokalne optimizacije se primenjuju unutar jednog osnovnog bloka i imaju za cilj uklanjanje nepotrebnih operacija kako bi kod bio efikasniji. Primeri:

- **Eliminacija zajedničkih podizraza** omogućava smanjenje ponovljenih izračunavanja. Na primer, ako imamo: $v1 = a\ op\ b$, ..., $v2 = a\ op\ b$ i vrednosti promenljivih a i b se ne menjaju između ovih dodela, kod se može prepisati kao: $v1 = a\ op\ b$ i $v2 = v1$. Na ovaj način eliminiše se nepotrebno izračunavanje i oslobađa prostor za dalje optimizacije.
- **Prenos kopiranja** se koristi kada postoji dodela oblika $v1 = v2$. Sve dok se ove promenljive ne menjaju, moguće je prepisati izraze koji koriste $v1$ kao da koriste direktno $v2$. Drugim rečima, izraz $a = v1$ može se zapisati kao $a = v2$, pod uslovom da takvo zapisivanje ne menja semantiku programa.
- **Eliminacija mrtvog koda** uklanja dodele koje se nikada kasnije ne koriste. Dodela promenljivoj naziva se mrtvom ako vrednost te promenljive nije iskorišćena u daljem toku programa. Utvrđivanje da li je dodela mrtva zavisi od toga kojim promenljivama se dodeljuju vrednosti i gde se te dodele vrše unutar osnovnog bloka.

30. Implementacija lokalnih optimizacija. Analiza dostupnih izraza. Analiza živosti.

Mnoge optimizacije mogu se primeniti samo uz odgovarajuću analizu ponašanja programa. Dve ključne analize za lokalne optimizacije su:

- **Analiza dostupnih izraza** - osnova za optimizacije poput eliminacije zajedničkih podizraza i prenosa kopiranja. Izraz se smatra **dostupnim** u nekoj tački programa ako neka promenljiva u programu već sadrži njegovu vrednost. Tokom eliminacije zajedničkih podizraza, dostupni izraz se zamenjuje promenljivom koja već nosi tu vrednost. U prenosu kopiranja, promenljive se zamenjuju dostupnim izrazom koji one koriste. Karakteristike:
  - **Smer**: unapred
  - **Domen**: skup izraza dodeljenih promenljivama
  - **Funkcije prenosa**: za dati skup dodela V i naredbu $a = b + c$, ukloni iz V svaki izraz koji koristi a kao podizraz i dodaj u V izraz $a = b + c$
  - **Početne vrednosti**: prazan skup izraza
- **Analiza živosti** - koristi se za eliminaciju mrtvog koda. Promenljiva se smatra **živom** u nekoj tački programa ako se njena vrednost kasnije čita pre nego što joj se dodeli nova vrednost. Eliminacija mrtvog koda funkcioniše tako što se izračunava živost svake promenljive, a dodele mrtvim promenljivama se uklanjaju. Da bi se odredilo koje promenljive su žive u određenoj tački, iterira se kroz sve naredbe osnovnog bloka u obrnutom redosledu. Inicijalno, za mali skup promenljivih se zna da će biti živ. Karakteristike:

- **Smer**: unazad
- **Domen**: skup promenljivih
- **Funkcije prenosa**: za dati skup promenljivih V i naredbu $a = b + c$, ukloni a iz V i dodaj u V promenljive b i c
- **Početne vrednosti**: zavise od semantike jezika

31. Lokalna analiza formalno.

U okviru **lokalne analize** postavljaju se tri osnovna pitanja: u kom smeru se vrši analiza, na koji način se ažuriraju informacije prilikom obrade pojedinačne naredbe i koje informacije su dostupne inicijalno. Lokalna analiza osnovnog bloka može se formalno definisati kao uređena četvorka $(D,\ V,\ F,\ I)$, gde su:

- D - **smer** analize (unapred ili unazad)
- V - **skup vrednosti** koje program ima u svakoj tački bloka
- F - **familija funkcija prenosa** koje definišu značenje svakog izraza kao funkciju $f : V \to V$ na vrhu ili dnu osnovnog bloka
- I - **inicijalne vrednosti** koje su poznate pre početka analize

Ovaj formalizam omogućava sistematično izvođenje lokalnih optimizacija i precizno određivanje živih promenljivih i dostupnih izraza u okviru osnovnog bloka.

32. Globalne optimizacije. Glavni izazovi.

Globalna analiza se primenjuje na graf kontrole toka funkcije i omogućava izvođenje optimizacija koje obuhvataju ceo program ili veći deo funkcije. Mnoge optimizacije koje se mogu izvesti lokalno unutar osnovnog bloka, poput eliminacije zajedničkih podizraza ili prenosa kopiranja, moguće je sprovesti i globalno. Pored toga, neke optimizacije se ne mogu realizovati lokalno, već samo globalno, kao što su globalna eliminacija mrtvog koda, globalno kopiranje konstanti i parcijalna eliminacija redundantnosti. Glavni izazovi globalne analize proizlaze iz kompleksnosti grafova kontrole toka. Svaka naredba može imati više prethodnika, pa analiza mora kombinovati informacije iz svih tih prethodnika. Graf kontrole toka može imati mnogo različitih puteva, pa vrednosti koje pratimo moraju više puta da se preračunavaju dok se ne usaglase. Problem je posebno izražen kod petlji, gde blokovi mogu međusobno zavisiti jedni od drugih. Zbog toga se analiza sprovodi iterativno: svim blokovima se dodeljuju početne (neutralne) vrednosti, a zatim se rezultati ponovo i ponovo ažuriraju dok se ne dostigne stabilno stanje. Na taj način se sprečava ulaženje u beskonačnu petlju i obezbeđuje tačan ishod optimizacije.

33. Globalna analiza živosti.

Za izvođenje **globalne analize živosti** koristi se iterativni algoritam koji određuje koje promenljive su žive u svakoj tački programa. Neka su $IN[s]$ skup promenljivih koje su žive neposredno pre izvršavanja i $OUT[s]$ skup promenljivih koje su žive neposredno nakon izvršavanja naredbe s. Na početku se postavlja $IN[s] = \{\}$ za svaku naredbu s, dok se

$IN[\text{exit}]$ postavlja na skup promenljivih za koje se zna da su žive na izlazu programa. Zatim se ponavlja sledeći postupak dok god dolazi do promena: za svaku naredbu s oblika $a = b + c$ (redosled obilaska naredbi može biti proizvoljan) postavlja se $OUT[s] = \cup_p IN[p]$ za svaki sledbenik p naredbe s i $IN[s] = (OUT[s] - a) \cup \{b,\ c\}$. Ovaj algoritam je dobar i korektan:

1. **Zaustavljanje**: Postoji konačno mnogo promenljivih i konačno mnogo mesta na kojima promenljive mogu postati žive, što garantuje da algoritam mora da se zaustavi.
2. **Korektnost**: Svako pojedinačno pravilo primenjeno na neki skup pravilno ažurira živost promenljivih. Kada se računa unija dva skupa živih promenljivih, promenljiva se smatra živom samo ako je bila živa na nekoj putanji koja vodi do naredbe.

Kada se algoritam zaustavi, on daje **saglasno rešenje** koje precizno odražava koje promenljive su žive u svakom delu programa, što omogućava dalju optimizaciju poput eliminacije mrtvog koda.

34. Primeri međuproceduralnih optimizacija koda.

Međuproceduralna optimizacija uzima u obzir celokupan graf kontrole toka, uključujući pozive funkcija i njihovu međusobnu interakciju. Primeri:

- **Umetanje koda (Inlining)** predstavlja duplikaciju funkcije na više mesta sa ciljem poboljšanja efikasnosti. Prednosti su brže izvršavanje programa i otvaranje prostora za nove optimizacije, a mana je veći kod kada se telo iste funkcije umetne na više mesta. Na osnovu analize koda potrebno je odabrati koje funkcije je korisno umetnuti. Umetanje "pravih" funkcija povećava efikasnost, dok umetanje "pogrešnih" funkcija povećava veličinu izvršnog fajla bez doprinosa performansama.
- **Izdvajanje koda (Outlining)** smanjuje količinu memorijskog prostora koji zauzima program, ali može potencijalno povećati vreme izvršavanja. Ova optimizacija pronalazi segmente koda koji se ponavljaju, izdvoji ih u zasebnu funkciju i menja pojavljivanja segmenta pozivima nove funkcije. Ova optimizacija je posebno korisna za uređaje sa ograničenom memorijom, kao što su pametni satovi i MP3 plejeri. Vreme izvršavanja može se pogoršati ako se izdvoji deo koda koji se veoma često izvršava.
- **Raspoređivanje funkcija (Code Positioning)** podrazumeva da ako funkcija A često poziva funkciju B, dobro je da budu memorijski raspoređene jedna pored druge. Operativni sistem učitava memoriju po stranicama, a broj stranica u fizičkoj memoriji je ograničen. Ako A poziva B mnogo puta, a funkcije se nalaze na različitim stranicama, to može negativno uticati na vreme izvršavanja. U slučaju složenih grafova poziva funkcija, potrebno je odlučiti kako funkcije rasporediti u memoriji. Primer algoritma za optimizaciju raspoređivanja funkcija je Profile Guided Code Positioning, koji koristi podatke o učestalosti poziva funkcija sadržane u profilu.

35. Optimizacije vođene profilima. Profili i postupak dobijanja profila. Svrha profila.

Profil je sažet zapis ponašanja programa tokom izvršavanja. Sadrži podatke kao što su koliko puta je metod pozvan, koliko puta je if grana tačna ili netačna, broj poziva virtuelnih metoda u određenom kontekstu i slično. Svrha profila je u donošenju odluka za međuproceduralne optimizacije - koje funkcije umetnuti, gde izdvojiti kod, kako rasporediti funkcije u memoriji i slično. **Dobijanje profila**:

- **JIT kompilacija** - profil se prikuplja automatski tokom izvršavanja aplikacije. Karakteriše ga veće korišćenje memorije i vreme izvršavanja dok se ključni delovi programa ne profiliraju.
- **AOT kompilacija** - profil se prikuplja pomoću instrumentacije programa tokom test izvršavanja. **Instrumentacija programa** podrazumeva ubacivanje koda koji prikuplja informacije bitne za profil. Rezultat se čuva u eksternoj datoteci i koristi pri optimizaciji.

Kvalitet profila zavisi od ulaznih podataka korišćenih tokom prikupljanja. Pogrešni profili mogu smanjiti efikasnost optimizacija. Alternativa je korišćenje heuristika ili metoda mašinskog učenja kada nije moguće dobiti pravi profil.

36. Generisanje koda. Izazovi alokacije registara. Naivni algoritam.

Ciljevi **generisanja koda** su izbor odgovarajućih mašinskih instrukcija za svaku IR instrukciju, raspodela resursa memorije i registara, i implementacija detalja izvršnog okruženja niskog nivoa. Brzina i veličina memorije značajno variraju: RAM je brz ali skup, dok je hard disk jeftin ali spor. Osnovna ideja je maksimalno iskoristiti prednosti različitih vrsta memorije, tj. smestiti objekte tako da se maksimizuju prednosti memorijske hijerarhije, i to potpuno automatski, bez pomoći programera. Većina mašina ima skup registara predviđenih za čuvanje promenljivih, pa je **alokacija registara** proces dodele promenljive registru i upravljanja transferom podataka između registara i memorije. Glavni izazovi alokacije registara su:

- **Mali broj registara** - broj registara je obično mnogo manji od broja promenljivih u IR-u, pa je potrebno maksimalno ponovno korišćenje registara.
- **Komplikovanost registara** - na primer, kod x86 arhitekture, registri se sastoje od više manjih registara koji se ne mogu koristiti istovremeno, a neke instrukcije zahtevaju specifične registre. Na MIPS arhitekturi, neki registri su rezervisani za specijalne namene, što dodatno ograničava izbor registara.

Naivni algoritam za alokaciju registara funkcioniše tako što se svaka vrednost uvek čuva u memoriji, a učitava se samo kada je potrebna. Generisanje koda ide kroz tri koraka:

1. Generiše se instrukcija za učitavanje vrednosti iz memorije u registar
2. Generiše se kod za izvršavanje računanja nad registrima

3. Generiše se instrukcija za upisivanje rezultata nazad u memoriju

Prednosti ovog pristupa su jednostavnost i lakoća implementacije - svaki deo IR koda može se prevesti direktno u asembler u jednom prolazu bez brige o broju registara ili specijalnim registrima. Međutim, mane su velike: naivni algoritam je izrazito neefikasan zbog ogromnog broja učitavanja i upisivanja u memoriju, što rezultira gubitkom vremena i prostora za vrednosti koje bi mogle ostati u registrima. Zbog toga se ovakav pristup retko koristi u realnim kompajlerima, osim za prototipove.

37. Alokacija registara. Linarno skeniranje. Razlivanje registara.

Cilj **alokacije registara** je zadržati u registrima što više promenljivih, kako bi se smanjio broj čitanja i pisanja u memoriju, kao i ukupno korišćenje memorije. Pri tome, u svakoj tački programa svaka promenljiva mora biti na istoj lokaciji, a svaki registar može sadržati najviše jednu živu promenljivu. Promenljiva se smatra **živom** u nekoj tački programa ako se njena vrednost koristi pre nego što se u nju upiše nova vrednost. Živost promenljivih može se odrediti korišćenjem globalne analize živosti. Skup tačaka u programu u kojima je promenljiva živa naziva se **živi opseg promenljive**, dok je **živi interval** najmanji interval IR koda koji obuhvata sve žive opsege promenljive. Ako su dati živi intervali promenljivih, registre je moguće alocirati jednostavnim pohlepnim algoritmom. Ideja je pratiti koji registri su slobodni u svakoj tački programa. Kada počne živ interval promenljive, dodeljuje se slobodan registar. Kada se živ interval završi, odgovarajući registar se oslobađa. Ovaj algoritam se naziva **linearno skeniranje**. Ako za promenljivu ne postoji slobodan registar, njena vrednost se mora **prosuti** u memoriju. Kada nam ponovo zatreba vrednost prosute promenljive, potrebno je iseliti neki postojeći registar u memoriju i u taj registar učitati vrednost prosute promenljive. Nakon upotrebe, sadržaj registra se vraća u memoriju i obnavlja se originalna vrednost registra. Ovaj proces se naziva **razlivanje registara**. Iako je spor, ponekad je neophodan. Prednosti linearnog skeniranja registara su brzina i efikasnost - algoritam se izvršava u linearnom vremenu, daje kvalitetan kod, alokacija se vrši u jednom prolazu i često se koristi u JIT kompajlerima. Mane su nepreciznost zbog korišćenja intervala umesto stvarnih opsega.

38. Alokacija registara. Bojenje grafova. Čajtinov algoritam.

Graf međusobne zavisnosti registara je neusmeren graf u kojem svaki čvor predstavlja jednu promenljivu i gde postoji grana između dve promenljive koje su žive istovremeno u nekoj tački programa. Alokacija registara se svodi na dodelu različitog registra svakoj promenljivoj u odnosu na susede. Problem alokacije registara u ovom kontekstu je ekvivalentan **problemu bojenja grafova**, koji je NP-težak ako postoje bar tri registra. Ne postoji poznat algoritam u polinomskom vremenu koji rešava ovaj problem za opšte slučajeve. **Čajtinov algoritam** funkcioniše na sledeći način: pretpostavimo da želimo obojiti graf sa k boja. Pronađemo čvor sa manje od k suseda. Ako taj čvor uklonimo iz grafa i obojimo preostali graf, možemo kasnije dodeliti boju i uklonjenom čvoru jer će uvek postojati slobodna boja. Ako ne postoji čvor sa manje od k suseda, proizvoljno odaberemo čvor i označimo ga kao problematičan. Kada ga vratimo u graf, možda će mu biti moguće dodeliti

odgovarajuću boju, a ako ne, promenljiva se prosipa u memoriju. Za mnoge CFG-ove algoritam pronalazi odlične dodele registara, a pošto se promenljive razlikuju po korišćenju, dobija se precizan graf međusobne zavisnosti registara. Zbog toga se često koristi u praktičnim kompajlerima. Mane su povezane sa NP-težinom problema bojenja grafova - heuristika može dovesti do patoloških dodela u najgorem slučaju.

39. Raspoređivanje instrukcija. Graf zavisnosti podataka.

Zbog procesorskog pipeline-a, redosled izvršavanja instrukcija može značajno uticati na performanse. **Raspoređivanje instrukcija** podrazumeva kreiranje rasporeda instrukcija sa ciljem poboljšanja performansi i nalazi se u podršci svih modernih kompajlera. **Zavisnosti među podacima** u mašinskom kodu predstavljaju skup instrukcija čije ponašanje zavisi jedna od druge. Intuitivno, to su instrukcije koje ne mogu biti proizvoljno poređane bez narušavanja ispravnog izvršavanja. Postoje tri osnovne vrste zavisnosti - **čitanje nakon pisanja (read after write)**, **pisanje nakon čitanja (write after read)** i **pisanje nakon pisanja (write after write)**.

Graf zavisnosti podataka prikazuje zavisnosti unutar osnovnog bloka. On je direktan aciklični graf - direktan jer jedna instrukcija zavisi od druge i acikličan jer ciklične zavisnosti nisu moguće. Instrukcije se mogu rasporediti unutar osnovnog bloka u bilo kom redosledu, sve dok nijedna instrukcija ne prethodi instrukciji od koje zavisi. Raspoređivanje instrukcija se sada zasniva na topološkom sortiranju grafa zavisnosti podataka, gde se instrukcije ređaju u skladu sa ovim redosledom. Problem je što može postojati mnogo ispravnih topoloških uređenja, a pronalaženje optimalnog rasporeda instrukcija u opštem slučaju je NP-težak problem.

40. Optimizacije koda zasnovane na upotrebi keša.

Upotreba **keša** se zasniva na dve vrste lokalnosti:

- **Vremenska lokalnost** - ako je nekoj memorijskoj lokaciji nedavno pristupano, verovatno će joj biti ponovo pristupano uskoro.
- **Prostorna lokalnost** - ako je nekoj memorijskoj lokaciji nedavno pristupano, verovatno će i njeni susedni objekti biti uskoro korišćeni.

Većina keš memorija je dizajnirana da iskoristi ove lokalnosti tako što u kešu čuva nedavno adresirane objekte, a često se ubacuje i sadržaj memorije u blizini. Programeri obično pišu kod bez razumevanja posledica lokalnosti jer jezici ne prikazuju detalje memorije. Neki kompajleri su sposobni da prepišu kod tako da se lokalnost iskoristi. Primer takve optimizacije je preraspoređivanje petlji, koje koristi prostornu lokalnost tako što menja redosled obilaska podataka u memoriji. Na primer, ako iteriramo matricu po kolonama, kompajler može da zameni petlje tako da se iterira po vrstama i time iskoristi keš i činjenica da su elementi nizova jedni pored drugih u memoriji.