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:

- **kompajleri** (**kompilatori**)
- **interpretatori**

Kompajleri daju mogućnost korišćenja istog koda napisanog na višem programskom jeziku na različitim procesorima. Prvi kompajler nastaje za prvi viši programski jezik - Fortran (1957). 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. Izvršni program se može pokretati željeni broj puta.

Kompajler treba da:

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

Dodatno treba omogućiti da kompajleri, osim što izvršavaju komande na standardnim mikroporcesorima, mogu kontrolisati fizičku opremu, te automatski generisati hardver. Takođe, treba omogućiti da specifikacija izvornog programa bude još bliža ljudima, njihovim govornim jezicima i iskustvu.

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. Izvršni program se može pokretati željeni broj puta. Ovde je u pitanju **AOT (Ahead-of-Time) kompilacija**.
- **interpretatori**: Interpretatori prevode naredbu po naredbu programa. Optimizacija vrše samo na nivou svake pojedinačne naredbe. U fazi izvršavanja uvek je neophodno da imamo kod. Prevođenje se svaki put vrši iz početka. 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).
- Vreme prevođenja.
- Svaka izmena u kodu zahteva celokupno novo prevođenje.
- Svaka izmena arhitekture računara zahteva novo prevođenje.

Prednosti i mane interpretatora:

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

Moguće je kombinovati AOT i JIT kompilaciju. Primer je JVM koji 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 programa 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. Srednji deo kompilatora, 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 genereisanje 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. Suštinski, pretprocesor vrši samo jednostavne operacije nad tekstualnim sadržajem programa i ne koristi nikakvo znanje o samom programskom jeziku. Pretprocesor analizira samo pretprocesorske direktive. 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**. 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... **Leksička analiza** je proces izdvajanja leksema i tokena, osnovnih jezičkih elemenata, 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. Tokenima mogu biti pridruženi i neki dodatni atributi. 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 program koji generiše leksere na programskom jeziku C, na osnovu zadatog opisa tokena u obliku regularnih izraza. Razlikujemo **ključne reči** i **identifikatore**: identifikator ne može biti neka od ključnih reci. Postoje ključne reči koje zavise od konteksta, koje su ključne reci na određenim specifičnim mestima programa, ali mogu biti identifikatori na drugim mestima.

6. Sintaksička analiza.

Sintaksa definiše formalne relacije izmedu 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. Kontekstno-slobodne gramatike su određene skupom **pravila**. Svako pravilo ima levu i desnu stranu. Sa leve strane pravila nalaze se tzv. **pomoćni simboli (neterminali)**, dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo tzv. **završni simboli (terminali)**. Svakom pomoćnom simbolu pridružena je neka sintaksička kategorija. Jedan od pomoćnih simbola se smatra istaknutim, naziva se **početnim simbolom (aksiomom)**. 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 koriscenje tzv. **generatora parsera**, poput sistema Yacc, Bison ili Antlr. Za opis sintakse koriste se i određene varijacije kontekstno-slobodnih gramatika. Najpoznatije od njih 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 čime se dobija pogodniji zapis, 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. Semantika programskog jezika određuje značenje jezika. 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. **Semantičke greške** se otkriju nakon leksičke i sintaksne analize. Primeri su upotreba nedefinisanog simbola, simboli definisani više puta u istom dosegu, greške u tipovima, greške u pozivima funkcija i prosleđivanju parametara.

Semantička analiza treba da odbaci što više nekorektnih programa, prihvati što više korektnih programa i uradi to brzo. Rezultati semantičkih provera često se prijavljuju programeru kao upozorenja. **Semantička upozorenja** su često nepotpuna. Na izbor semantičkih provera koje kompajler implementira utiče pre svega efikasnost analize - kompajler mora da radi efikasno i kompleksna semantička analiza nije poželjna jer usporava proces kompilacije. Na izbor semantičkih provera koje kompajler sprovodi na nekom projektu utiče pre svega domen projekta. Najčešće se izbor semantičkih provera može kontrolisati opcijama kompajlera. Primeri upozorenja su neiskorišćena promenljiva, predlog upotrebe zagrada oko izraza i deljenje nulom.

Jednostavna semantička analiza zasniva se na ispitivanju karakteristika apstraktnog sintaksnog stabla. Za dodavanje jednostavnih semantičkih provera, u okviru clang-a postoji čak tri interfejsa, dva koja su sastavni deo kompajlera, i treći koji je nezavisan alat u sklopu clang projekta. Kompleksnija semantička analiza uključuje i analizu koda i grafa kontrole toka. Jedan od osnovnih zadataka semantičke analize je **provera tipova (typchecking**). 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.

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 - graphical representations, three-address representation, virtual machine representations, linear representations. 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).
- **RTL (Register Transfer Language)**

9. Optimizacije međukoda. Uloga i primeri.

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

- **optimizacija međukoda** - vrši se na početku faze sinteze i podrazumeva mašinski nezavisne optimizacije tj. optimizacije koje ne uzimaju u obzir specifčnosti ciljne arhitekture.
- **optimizacija ciljnog koda** - izvršava se na samom kraju sinteze i zasniva se na detaljnom poznavanju ciljne arhitekture i asemblerskog i mašinskog jezika na kome se izražava ciljni program.

Primeri:

- **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 zavrsni asemblerski tj. mašinski kod. Tri osnovne faze:

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. **faze 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 žrtvovanjem broja ciklusa po instrukciji, tj. 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.

Kod generator 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, kod generator 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. Na nekim mašinama recimo floating point se rešava sa odvojenim registrima.
- Željenog kvaliteta generisanog koda - ako nas nije briga za efikasnost ciljnog programa, odabir instrukcija je trivijalan. Neophodno je da znamo cene instrukcija kako bi 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 nastala zbog zavisnosti između susednih instrukcija.

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

Projekat LLVM

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 nezavistan 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, a rezultat je generisanje LLVM IR-a. Clang je najpoznatiji front-end koji prevodi C-olike jezike u ovaj oblik.
- **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, 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 ekstenzijom *.ll*. 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 (Static Single Assignment) 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 unosa 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. 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. 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 pretrage.
- 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.

Jednostavna 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. **Početne aksiome**:

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

Jednostavna pravila:

$$\frac{i \text{ is an integer constant}}{\vdash i : int}$$

$$\frac{s \text{ is a string constant}}{\vdash s : string}$$

$$\frac{d \text{ is a double constant}}{\vdash d : double}$$

Složenija pravila:

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

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

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 (`for` ili `while`).

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

- **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 tipovan, 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 $\perp$ (nekad se naziva i **bottom tip**). Po definiciji važi:

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

Kada detektujemo tip `Error` , tretiramo ga kao da je dokazano da izraz ima tip $\perp$. Pravila zaključivanja se unapređuju tako da uključuju ovaj tip. 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` , ali 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 najbolja 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 $(A_1,\ A_2,\ \ldots,\ A_n)$ i $(B_1,\ B_2,\ \ldots,\ B_n)$ 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:

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:

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

Hijerarhijsko preopterećenje konceptualno je slično radu sa **dosezima (scope)**.

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

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

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

23. Izvršno okruženje i funkcije članice klasa. Pokazivač this i dinamičko odredivanje poziva. Tabela virtulenih funkcija.

24. Implementiranje dinamičkih provera tipova.

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

26. Troadresni kod. Funkcije i stek okviri.

27. Algoritam generisanja troadresnog koda.

28. Optimizacije međukoda. Graf kontrole toka.

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

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

31. Lokalna analiza formalno.

32. Globalne optimizacije. Glavni izazovi.

33. Globalna analiza živosti.

34. Primeri meduproceduralnih optimizacija koda.

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

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

37. Alokacija registara. Linarno skeniranje. Razlivanje registara.

38. Alokacija registara. Bojenje grafova. Čajtinov algoritam.

39. Raspoređivanje instrukcija. Graf zavisnosti podataka.

40. Optimizacije koda zasnovane na upotrebi keša.