Konstrukcija kompilatora
- odgovori na ispitna pitanja -

Mina Milošević
mi17081@alas.matf.bg.ac.rs

2020/2021

Materijal je preuzet sa stranice prof. dr. Milene Vujošević-Janičić.

Sadržaj

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, programski prevodioci) i *interpretatori* (ne procesiraju program pre izvršavanja; karakteriše ih sporije izvršavanje, koriste se često za jezike skript paradigme).
Kompajleri daju mogućnost korišćenja istog koda napisanog na višem programskom jeziku na različitim procesorima. Prvi kompajler nastaje za prvi viši programski jezik, tj. za 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).
Dodatni izazovi u razvoju kompajlera. Omogućiti da kompajleri, osim što izvršavaju komande na standardnim mikroporcesorima, mogu kontrolisati fizičku opremu, te automatski generisati hardver. Omogućiti da specifikacija izvornog programa bude još bliža ljudima, njihovim govornim jezicima i iskustvu.

2 Struktura kompajlera

Prednji deo kompajlera (front end) - zavisi od jezika: vrši leksičku, sintaksnu i semantičku analizu. Transformiše ulazni program u međureprezentaciju (engl. 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 kompajlera (middle end) - nezavisan od jezika i ciljne arhitekture i vrši optimizacije. Vrši optimizacije na međureprezentaciji koda sa ciljem da unapredi performanse i kvalitet mašinskog koda koji će kompajler proizvesti. Izvršava optimizacije na međureprezentaciji koje su nezavisne od CPU arhitekture za koju je krajnji kod namenjen. Srednji deo kompajlera, da bi izvršio kvalitetnu optimizaciju, najpre vrši analizu koda. Koristeći rezultate analize, vrši se odgovarajuća optimizacija. Izbor optimizacija koje će biti izvršene zavisi od želja korisnika i argumenata koji se zadaju prilikom pokretanja kompajlera. Podrazumevan nivo optimizacija obuhvata optimizacije nivoa O2.
Zadnji deo kompajlera (back end) - zavisi od ciljne arhitekture i vrši generisanje koda. Zadnji deo kompajlera je takođe odgovoran za optimizacije koje su arhitekturalno specifične. Osnovne faze zadnjeg dela kompajlera obuhvataju arhitekturalno specifičnu optimizaciju i genereisanje koda.

3 Pretprocesiranje i linkovanje

Osnovne faze prevođenja, osim kompilacije, obuhvataju i pretprocesiranje i linkovanje. Faza *pretprocesiranja* je pripremna faza kompilacije. Faza *linkovanja* je neophodna faza kako bi se na osnovu proizvoda kompilacije napravio izvršivi program.
Pretprocesiranje. 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.
Povezivanje. 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 (zapravo na njegovom početku).

4 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, a na osnovu zadatog opisa tokena u obliku regularnih izraza.
Razlikujemo ključne reči i identifikatore: identifikator ne može biti neka od ključnih reči. Postoje ključne reči koje zavise od konteksta, koje su ključne reči na određenim specifičnim mestima programa, ali mogu biti identifikatori na drugim mestima.

5 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 *sintakstičkog 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 (ili aksiomom).
Na osnovu gramatike jezika formira se *potisni automat* na osnovu kojeg se jednostavno implementira program koji vrši sintakstičku analizu. Formiranje automata od gramatike se obično vrši automatski, uz korišćenje 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.

6 Semantička analiza

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**:
upotreba nedefinisanog simbola; simboli definisani više puta u istom dosegu; greške u tipovima; greške u pozivima funkcija, prosleđivanju parametara.
Izazovi semantičke analize - odbaciti što više nekorektnih programa, prihvatiti što više korektnih programa i uraditi 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 "Options to Request or Suppress Warnings". **Primeri:**
neiskorišćena promenljiva; predlog upotrebe zagrada oko izraza; 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 ali u sklopu clang projekta. Kompleksnija semantička analiza uključuje i analizu koda i grafa kontrole toka.
Jedan od osnovnih zadataka semantičke analize je provera tipova (engl. typchecking). Tokom provere tipova proverava se da li je svaka operacija primenjena na operande odgovarajućeg tipa. U zavisnosti od jezika se u nekim slučajevima u sintaksičko drvo umeću implicitne konverzije, gde je potrebno ili se prijavljuje greška.

7 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.
Čemu služi ova međureprezentacija? Pojednostavljivanje optimizacija; da bi imali više prednjih delova za isti zadnji deo; da bi imali 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. Previsok nivo: nije moguće optimizovati neke implementacione detalje. Prenizak nivo: nije moguće korisititi znanje visokog nivoa da se izvrše neke gresivne 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 tzv. *troadresni kod* u kome se javljaju dodele promenljivama u kojima se sa desne strane dodele javlja najviše jedan operator. Naziv troadresni: u svakoj instrukciji se navode "adrese"najviše dva operanda i rezultata operacije. Naredbe kontrole tokase 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)) i RTL (Register Transfer Language).

8 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.
Zašto je potrebna optimizacija? Generisanje IR-a uključuje redundatnost u naš kod; programeri su lenji.
Optimizacija se najčešće odvija na dva nivoa:

- *optimizacija međukoda* se generiše na početku faze sinteze i podrazumeva mašinski nezavisne optimizacije tj. optimizacije koje ne uzimaju u obzir specifičnosti ciljne arhitekture
- *optimizacija ciljnog koda* se izvršava 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

9 Generisanje koda. Izazovi. CISC i RISC arhitekture

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

1. faza odabira instrukcija - tada se određuje kojim mašinskim instrukcijama se modeluju instrukcije troadresnog koda
2. faza alokacije registara - tada se određuje lokacija na kojoj se svaka od promenljivih skladišti
3. faze raspoređivanja instrukcija - tada se određuje 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 IRa. Generatori koda razdvajaju IR instrukcije u "basic blocks", koje 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 su RISC i CISC.
CISC arhitekuru procesora karakteriše bogat skup instrukcija. Bogat skup instrukcija 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. CISC procesori se uglavnom koriste na ličnim računarima, radnim stanicama i serverima, a primer ovakvih procesora je arhitektura Intel x86. CISC procesori imaju više različitih načina adresiranja, od kojih su neki veoma kompleksni. CISC procesori obično nemaju veliki broj registara opšte namene.

RISC arhitektura procesora se zasniva 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, koja se lako može implementirati. Protočna obrada (pipeline) 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.

10 Izbor instrukcija. Izbor registara. Raspoređivanje instrukcija

Izbor instrukcija. Kod generator mora da mapira IR program u sekvencu koda koji može da bude izvršen na ciljanoj arhitekturi. Kompleksnost mapiranja zavisi od:

- *Nivoa apstrakcija/Preciznosti IR-a* - 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* - brzina instrukcija je bitan faktor; ako nas nije briga za efikasnost ciljanog programa, odabir instrukcija je trivijalan. Neophodno je da znamo cene instrukcija kako bi mogli da dizajniramo dobre sekvence koda

Izbor registara. 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* - tokom koje se biraju skupovi promenljivih koji treba da borave u registrima u svakoj tački programa
2. *Dodela registara* - tokom koje se biraju 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. Najjednsotavnije 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.

11 Jednoprolazni i višeprolazni kompilatori

Analiza se može obaviti u jednom ili više prolaza. Skeniranje i parsiranje se može odraditi u jednom prolazu. Neki kompajleri kombinuju skeniranje, pasrsiranje, semantičku analizu i generisanje koda u jednom prolazu. Takvi kompajleri se nazivaju ***jednoprolazni kompajleri***. Većina kompajlera ipak prolazi kroz kod više puta i to su ***višeprolazni komapjleri***. Neki jezici su dizajnirani tako da podrže jednoprolazne kompajlere (C i C++). Neki jezici zahtevaju višeprolazne komajlere (Java). Većina modernih komapjlera koristi veoma veliki broj prolaza kroz kod. Pravila dosega u višeprolaznim kompajlerima:

- *Prvi prolaz* - kompletno parsiranje ulaznog koda i kreiranje ASTa
- *Drugi prolaz* - prolazak kroz AST i skupljanje informacija o klasama
- *Treći prolaz* - prolazak kroz AST i provere raznih osobina

12 LLVM osnovne informacije. Značaj i mogućnosti

Projekat ***LLVM*** sastoji se iz biblioteka i alata koji zajedno čine veliku kompajlersku infrastrukturu. Započet je kao istraživački projekat na Univerzitetu Ilinois, 2000. godine, kao istraživački rad sa ciljem proučavanja tehnika kompajliranja i kompajlerskih optimizacija. Ideja: skup modularnih i ponovno iskoristivih kompajlerskih tehnologija, čiji je cilj podršska statičkoj i dinamičkoj kompilaciji proizvoljnih programskih jezika. Osnovna filozofija LLVM-a je da je „svaki deo neka biblioteka" i veliki deo koda je ponovno upotrebljiv. Inicijatori projekta su Kris Latner i Vikram Adve.
LLVM je sveobuhvatni naziv za više projekata koji zajedno čine potpun kompajler: prednji deo, središnji deo, zadnji deo, optimizatore, asemblere, linkere, libc++ i druge komponente. Projekat je napisan u programskom jeziku C++, koristeći prednosti objektno-orijentisane paradigme, generičkog programiranja, a takođe sadrži i svoje implementacije raznih struktura podataka koje se javljaju u standardnim bibliotekama programskog jezika C/C++. *Clang/Clang++* se često koristi kao sinonim za LLVM kompajler. Clang/Clang++ ima odlične karakteristike u poređenju sa kompajlerima kao što su gcc i icc. Kada se vrše poređenja, ona se vrše na odabranim primerima. U zavisnosti od primera (benchmarks), poređenja mogu da daju različite rezultate. Obično Clang/Clang++ ima brže vreme kompilacije u odnosu na pomenute kompajlere.
Licenca koda u okviru LLVM projekta je "Apache 2.0 License with LLVM exceptions". Licenciranje se menjalo tokom razvoja projekta, ali je uvek bila licenca otvorenog koda.
LLVM implementira kompletan tok kompilacije:

- *Front-end* - leksička, sintaksička i semantička analiza (alat Clang)
- *Middle-end* - analize i optimizacije (alat opt)
- *Back-end* - različite arhitekture (alat llc)

Na primer, korisnik može da kreira svoj front-end, i da ga poveže na LLVM, koji će mu dodati middle-end i back-end za izabran postojeći front-end/back-end; da isprobava promene na nivou middle-end-a; za novu arhitekturu obezbedi back-end, i da koristi Clang i postojeći middle-end.

13 LLVM projekti

14 LLVM prednji deo

15 LLVM srednji deo. LLVM-ov međukod

16 LLVM srednji deo. Alat opt i LLVM prolazi

17 LLVM zadnji deo

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

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

20 Određivanje dosega. Dinamički dosezi

21 Pravila za određivanje tipova u izrazima

22 Tipovi i nasleđivanje. Tip null

23 Određivanje tipova kod ternarnog operatora

24 Pravila za određivanje tipova u naredbama

25 Tipovi i propagiranje greške

26 Preopterećivanje funkcija

27 Kompletnost i saglasnost sistema tipova. Kovarijanta povratnog tipa. Kovarijanta po argumentu funkcije. Kotravarijanta po argumentu funkcije.

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

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

30 Izvršno okruženje i objekti. Strukture, objekti i nasleđivanje

31 Izvršno okruženje i funkcije članice klasa. Pokazivač this i dinamičko određivanje poziva

32 Tabela virtulenih funkcija i tabela metoda. Višestruko nasleđivanje i interfejsi