Projektovanje elektronskih sistema

Predavanje 4

Kompajleri (Prevodioci)

Doc.dr Borisav Jovanović

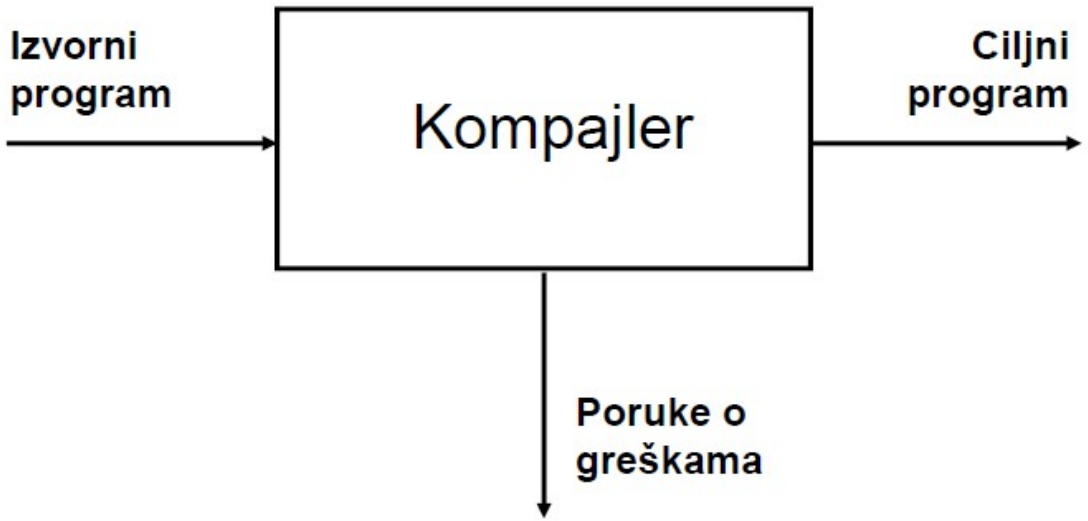
preuzeto iz predavanja prof. Milunke Damnjanovic i prof. Miluna Jevtica

Sadržaj:

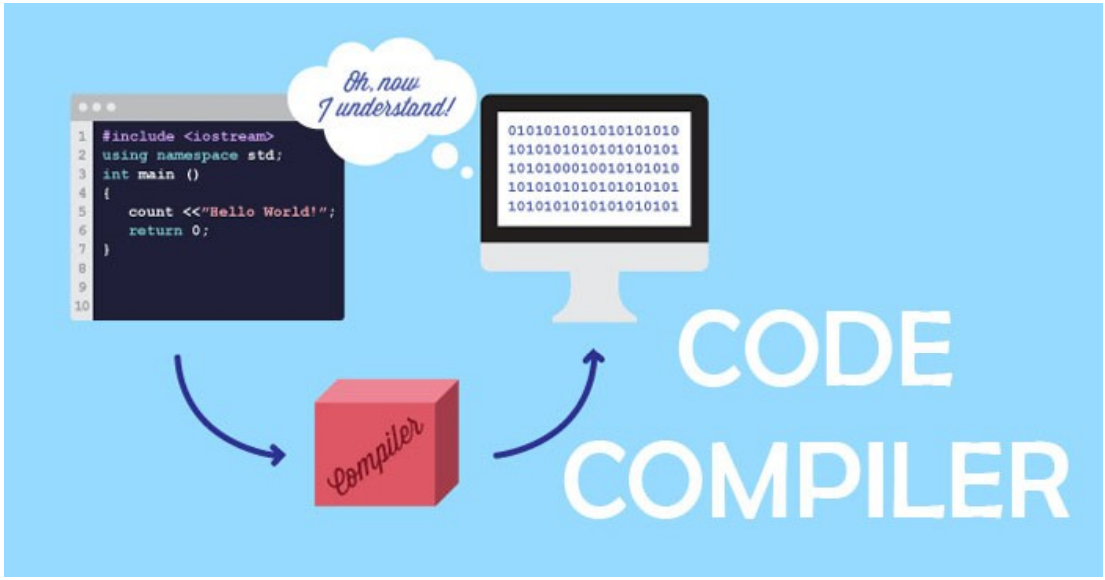
- Funkcija kompajlera
- Generacije računarskih jezika
- Analiza-sinteza, model kompilacije
- Analiza izvornog programa
  - Linearna analiza
  - Sintaksna analiza
  - Semantička analiza
  - Generisanje međukoda
- Sinteza koda
  - Optimizacija
  - Generisanje mašinskog koda
- Preprocesiranje
- Linker/loader
- Interpreteri

Funkcija kompajlera

- Kompajler (prevodilac) je program koji čita program napisan na jednom jeziku – izvornom i prevodi ga u ekvivalentan program u drugom jeziku – ciljnom jeziku.



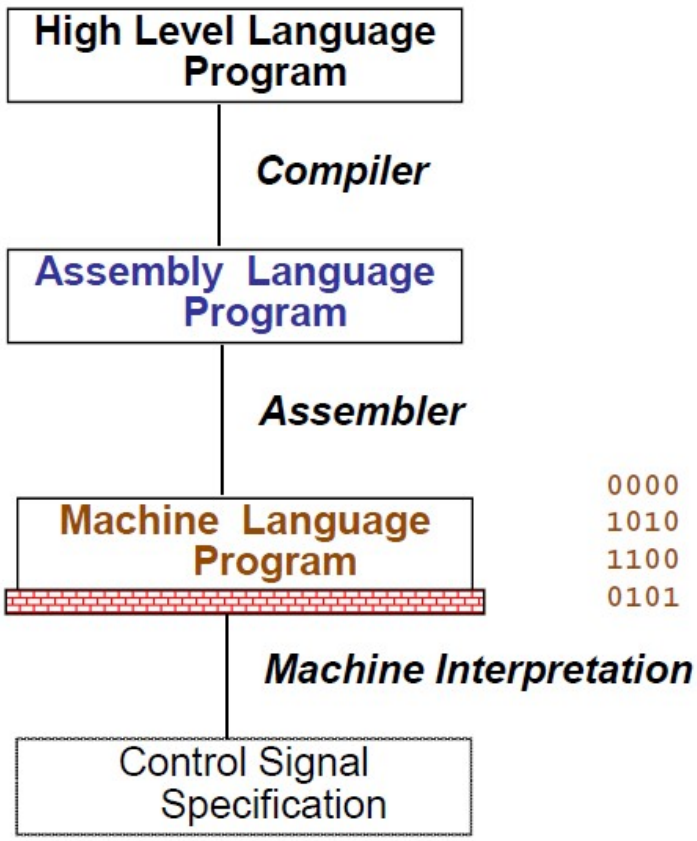
- Prijavljuje korisniku prisustvo grešaka u izvornom programu.
- Ciljni jezik može biti neki drugi programski jezik ili mašinski jezik
- Kompajleri mogu da budu jednoprolazni, višeprolazni, load-and-go, debugging, optimizing, u zavisnosti za šta su projektovani
- Prvi kompajleri pojavili su se ranih 1950, koristili su se za prevođenje aritmetičkih formula u mašinski kod



Generacije računarskih jezika

Globalno, razlikujemo četiri klase računarskih jezika:

- prva generacija - mašinski jezik
- druga generacija - asemblerski jezik
- treća generacija viši programski jezici (HLL)
- četvrta generacija - novi jezici



Prva generacija – mašinski jezik

- svaka instrukcija, na hardverskom nivou, direktno upravlja radom mašine, tj. pojedinim gradivnim blokovima.
- instrukcije su numeričke, predstavljene u formi 0 i 1
- programiranje je naporno i podložno velikom broju grešaka
- efikasnost programiranja je niska
- programi nerazumljivi korisniku
- direktno se pristupa resursima mašine

Druga generacija – asemblerski jezik

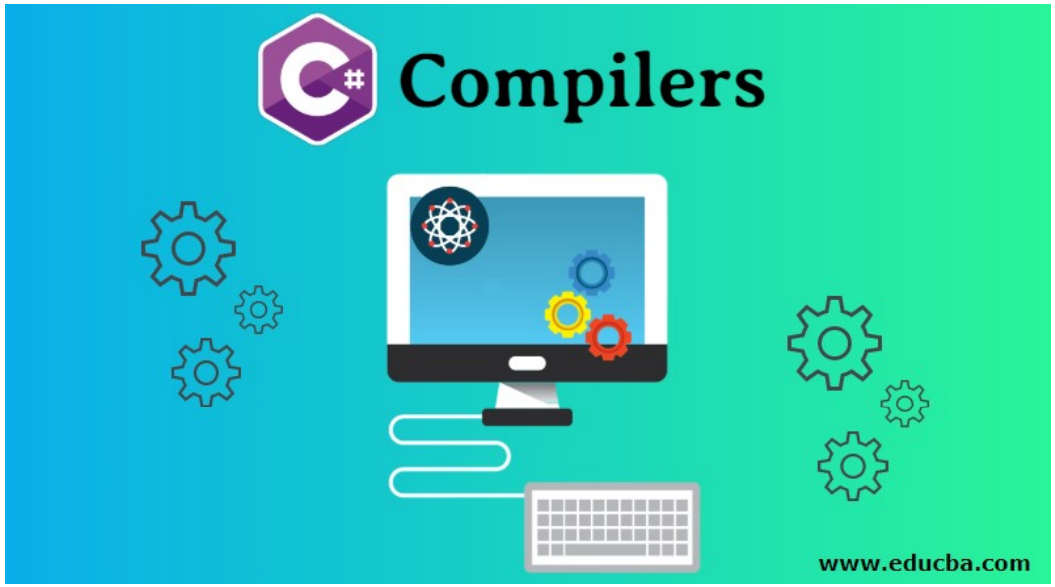
- svaka instrukcija se predstavlja mnemonikom, kao na primer ADD/MUL/MOV
- korespondencija izmedju asemblerskih i mašinskih instrukcija je jedan-na-prema-jedan
- postoje i direktive koje nemaju izvršno dejstvo, ali programu na asemblerskom jeziku olakšavaju prevodjenje
- dodela memorije i segmentacija programa
- direktno se pristupa resursima mašine
- veća brzina izvršenja programa
- efikasnije korišćenje memorije

Treća generacija – Viši programski jezici

- kompajler prevodi programske iskaze u sekvence instrukcija na mašinskom nivou
- u principu jedan iskaz na HLL-u (High Level Language) se prevodi u n (n >= 1) instrukcija na mašinskom (asemblerskom) jeziku
- programiranje je jednostavnije
- efikasnost je veća
- ispravljanje grešaka lakše
- nema direktni pristup resursima mašine
- neefikasno iskorišćenje memorije
- duži su programi

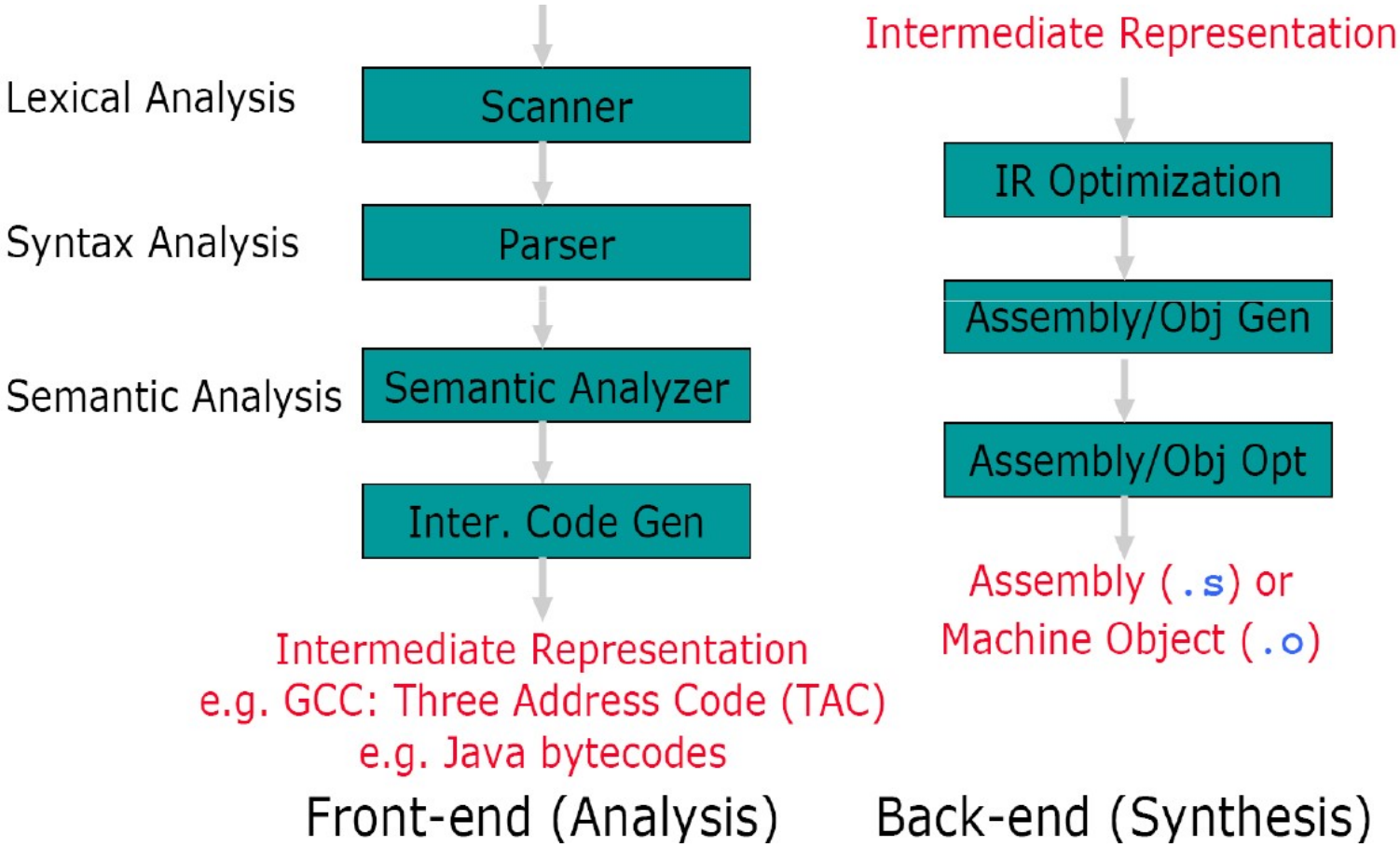
Četvrta generacija – novi jezici

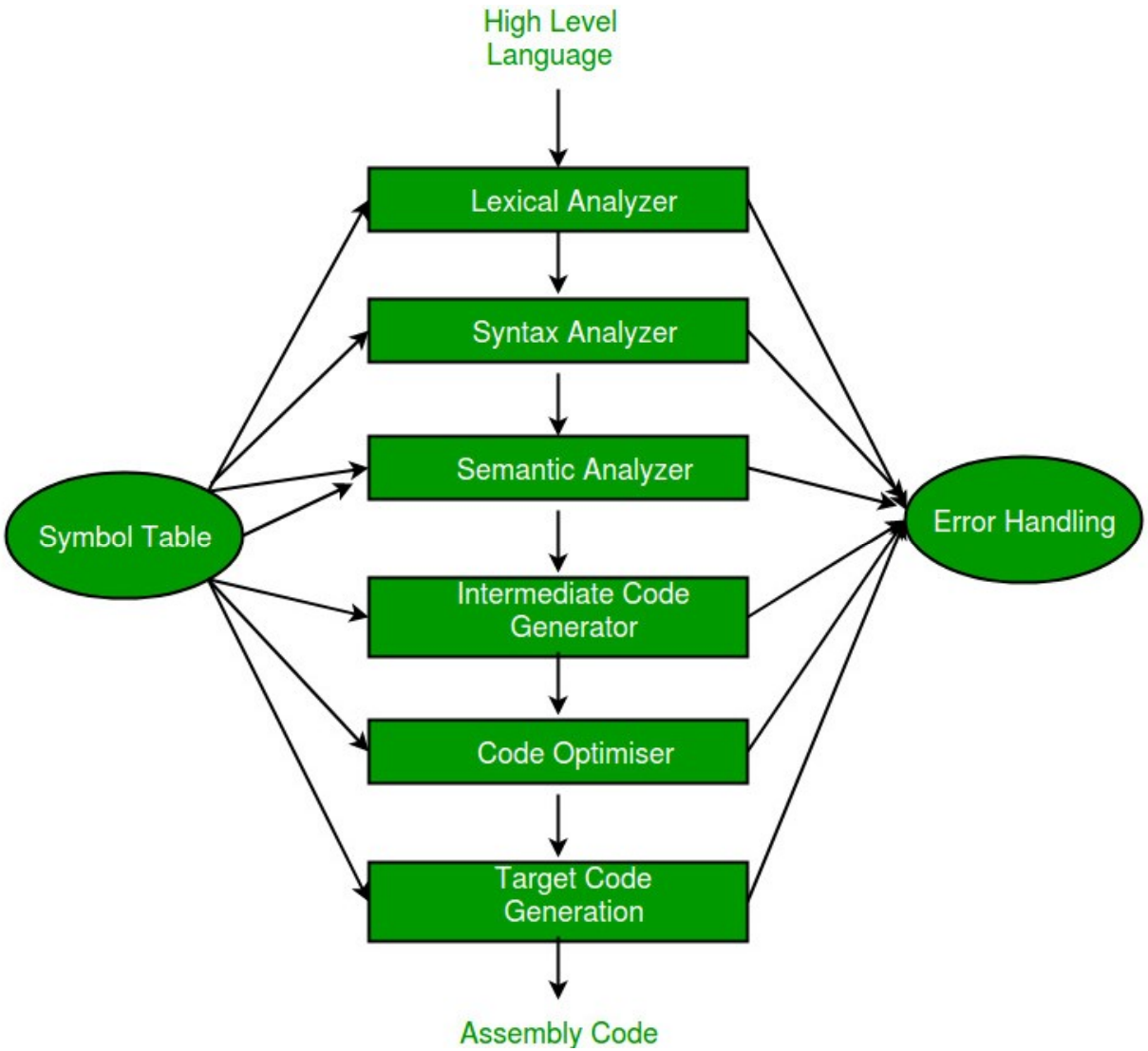
- implementiraju veštačku inteligenciju (primer je TensorFlow)
- jezici za pristup bazama podataka (primer je MySQL)
- objektno-orijentisani jezici (primeri su C++, Java i dr.)



Analiza-sinteza, model kompilacije

- Kompilacija ima dva dela: analizu i sintezu. Analiza rasčlanjuje izvorni program na delove i kreira međupredstavu izvornog programa.
- Sinteza konstruiše ciljni program na osnovu međupredstave.
- U toku analize, operacije zadate izvornim programom se definišu i zapisuju u obliku hijerarhiske strukture nazvane **stablo**. Često se koristi naziv **sintaksno stablo**, kod koga svaki čvor predstavlja operaciju, a deca argumente operacije.


Lexical Analysis
Scanner
Syntax Analysis
Parser
Semantic Analysis
Semantic Analyzer
Inter. Code Gen
Intermediate Representation
e.g. GCC: Three Address Code (TAC)
e.g. Java bytecodes
Front-end (Analysis)
Intermediate Representation
IR Optimization
Assembly/Obj Gen
Assembly/Obj Opt
Assembly (.s) or
Machine Object (.o)
Back-end (Synthesis)


High Level Language
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Code Optimiser
Target Code Generation
Assembly Code
Symbol Table
Error Handling

Većina softverskih alata koja radi sa izvornim programima obavlja prvo neku vrstu analize. Primeri su:

- **Strukturni editor -** dobija kao ulaz niz komandi kojima gradi izvorni program. Ne obavlja samo funkcije kreiranja i modifikacije teksta kao uobičajeni editor teksta, već analizira tekst, proverava da li je pravilno formatiran, može automatski da dodaje ključne reči. Izlaz ovakvog editora je sličan izlazu analize kod kompajlera.
- **Statički čeker** čita program, analizira ga i pokušava da otkrije potencijalne greške bez izvršavanja programa. Deo analize je sličan onom u optmizirajućim kompajlerima. Statički čeker može da otkrije delove koda koji nikada neće da budu izvršavani ili da li je neka promenljiva korišćena pre nego što je definisana. Pokušava da otkrije i logičke greške, ako se recimo u nekoj operaciji umesto realnog broja koristi pointer.
- **Interpreteri**, umesto dobijanja ciljnog programa kao prevoda, **interpreter obavlja operacije** definisane izvornim programom. Npr. za naredbu dodeljivanja može da formira stablo kao na slici i izvede operacije u čvorovima kako prolazi duž stabla.
- Interpreteri se koriste za izvršavanje komandnih jezika.

Kontekst kompajlera

- Izvorni program može da bude podeljen na module memorisane u odvojenim fajlovima.
- Zadatak prikupljanja izvornih programa može da bude poveren posebnom programu koji se naziva **preprocesor**.
- Preprocesor se takođe koristi da proširi skraćenice, tzv. makroe u naredbe izvornog jezika.

Analiza izvornog programa

- Kod kompajliranja analiza se sastoji iz tri faze:
- Linearna analiza, kod koje se niz karaktera koji čine izvorni program učitava s leva na desno i grupiše u tokene, koji su nizovi karaktera koji imaju zajedničko značenje
- Hijerarhijska analiza kod koje se tokeni grupišu u hijerarhiski ugnježdene skupove sa zajedničkim značenjem
- Semantička analiza, gde se izvode provere da bi se osiguralo da komponente programa imaju isto značenje

Linearna analiza

Linearna (leksička) analiza.

Karakteri u naredbi dodeljivanja biće grupisani u tokene.

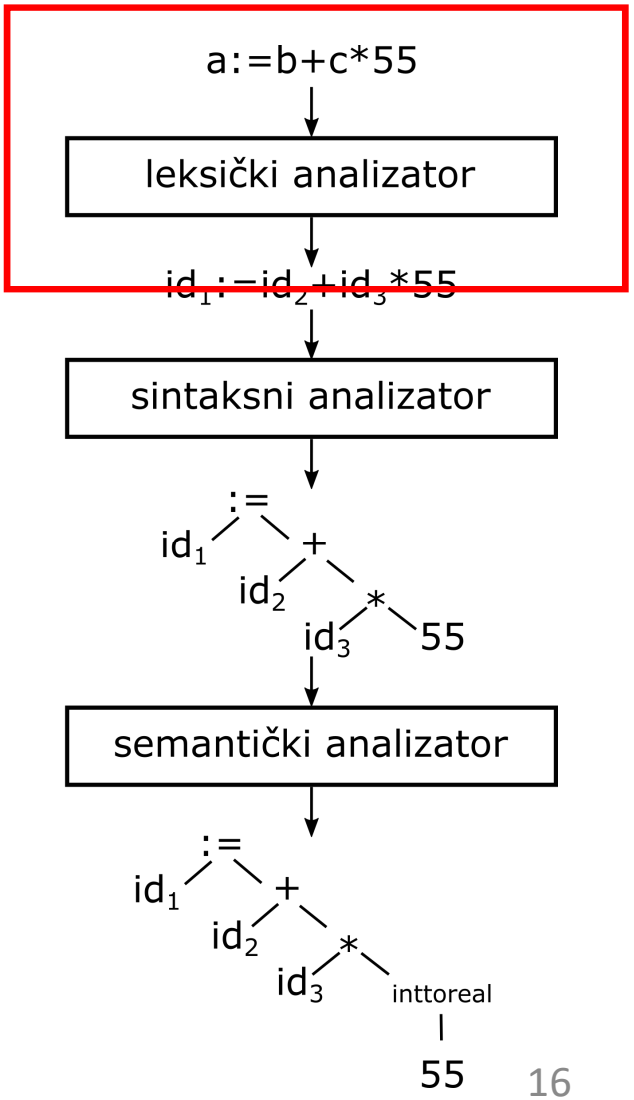
a := b + c*55

1. Identifikator a
2. Simbol dodeljivanja :=
3. Identifikator b
4. Znak plus
5. Identifikator c
6. Znak množenja
7. Broj 55

Blanko znaci kojima su odvojeni karakteri ovih tokena biće eliminisani u toku leksičke analize.

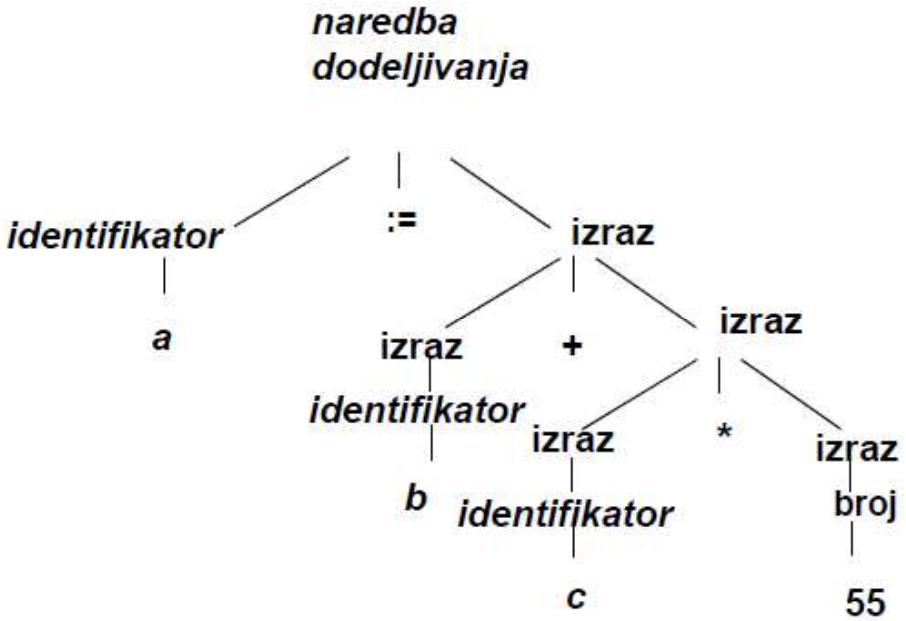
tabela simbola

a	...
b	...
c	...



Sintaksna analiza

- Sintaksna (hijerarhijska) analiza naziva se *raščlanjivanjem (parsing)*.
- Ona podrazumeva grupisanje tokena izvornog programa u gramatičke fraze koje se koriste od strane kompajlera za sintetizovanje izlaza.
- Obično se gramatiičke fraze izvornog programa predstavljaju parserskim stablom kao što je prikazano na sledećoj slici.
- a := b + c*55



Hijerarhijska struktura programa obično se izražava **rekurzivnim pravilima**. Na primer, mogli bismo da imamo sledeća pravila kao deo definicija izraza:

(1) Bilo koji *identifikator* je izraz.

(2) Bilo koji *broj* je izraz.

(3) Ako su *izraz1* i *izraz2* izrazi, onda su izrazi takodje

izraz1 + *izraz2;*

izraz1 * *izraz2*

(*izraz1*)

Pravila (1) i (2) su osnovna pravila, dok pravilo (3) definiše izraze pomoću operatora primenjenih na druge izraze. Dakle, po pravilu (1) promenljive b i c su izrazi.
Po pravilu (2) , broj 55 je izraz, dok po pravilu (3) prvo zaključujemo da c*55 predstavlja izraz, i konačno, da b+c*55 predstavlja izraz.

Slično, većina programskih jezika definiše naredbe rekurzivno pomoću pravila, kao što su recimo:

1. Ako je *identifikator1* je identifikator, a *izraz2* izraz, onda je naredba
   Identifikator:=izraz2
2. Ako je *izraz1* izraz, a *naredba2* naredba onda su naredbe:
   While (izraz1) do naredba2
   If (izraz1) then naredba2

Semantička analiza

- Semantičkom analizom se proverava da li izvorni program ima semantičkih grešaka i prikupljaju se informacije o tipovima podataka za kasniju fazu generisanja koda.
- Koristi se hijerarhijska struktura odredjena sintaksnom analizom za identifikovanje operatora, operanada, izraza i naredbi.
- Važna komponenta semantičke analize je **provera tipa**.
- Ovde kompajler proverava da li svaki operator ima operande koji su dopušteni specifikacijom izvornog jezika.

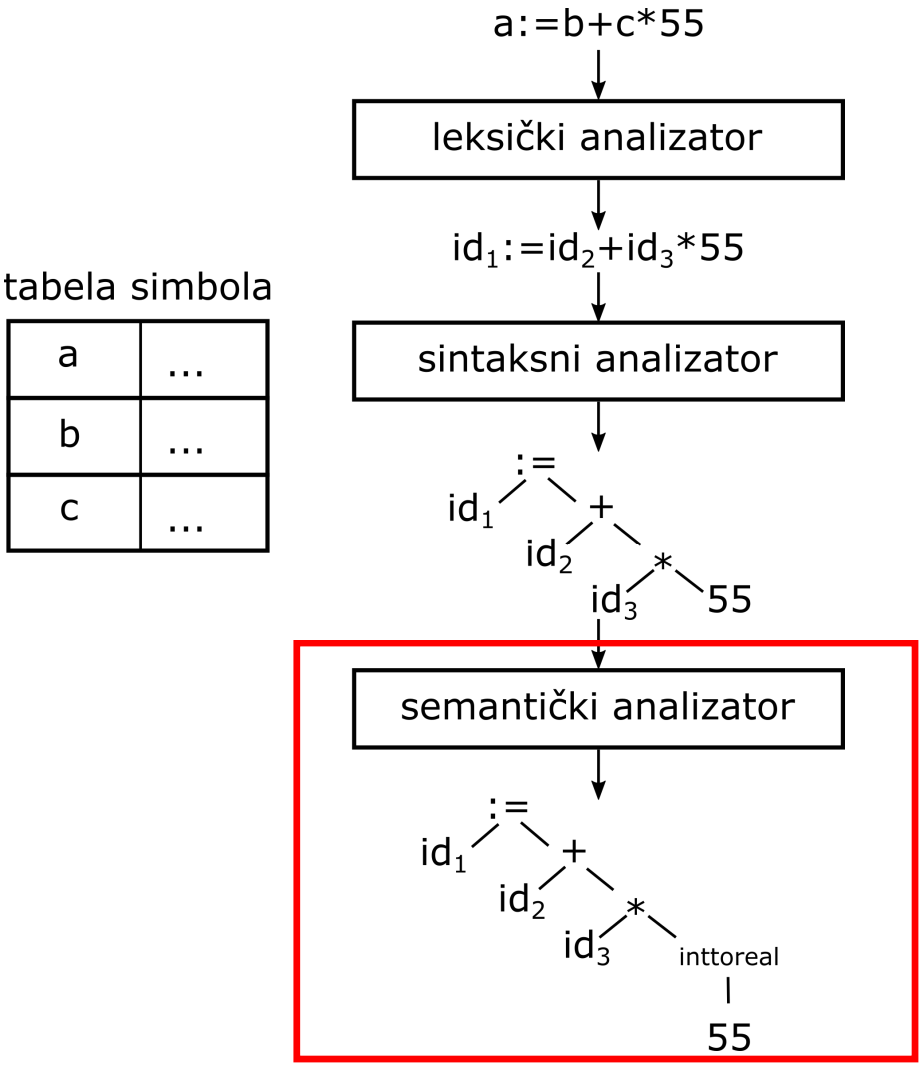
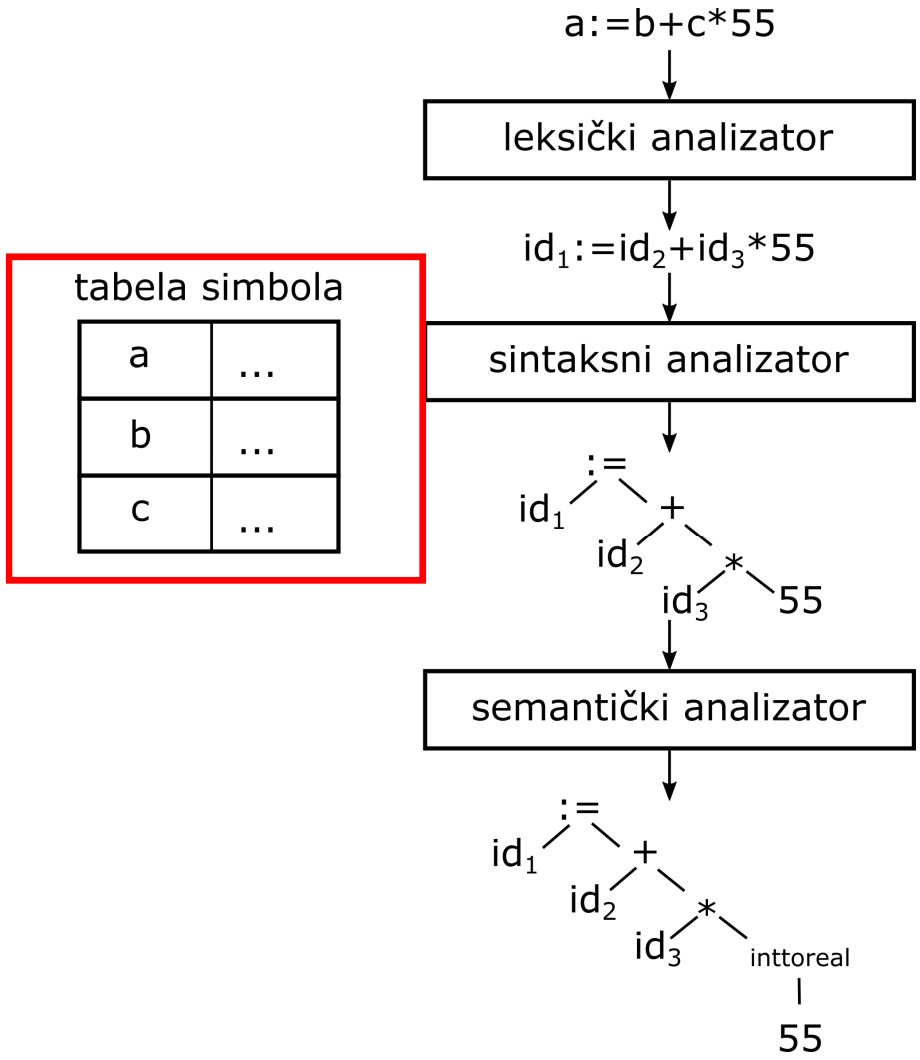


Tabela simbola

Važna funkcija kompajlera je da prvo zapiše sve identifikatore korišćene u izvornom programu i onda prikupi informacije:

- o atributima svakog identifikatora (o memoriji – koliki prostor zauzima neka promenljiva, kog je tipa, opseg gde je u programu važeća),
- u slučaju imena procedura, broj i tipove argumenata, način prenošenja argumenta da li po vrednosti ili po referenci, koji je vraćeni tip, itd.
- Nastaje **Tabela simbola**. To je struktura podataka koja sadrži zapis za svaki identifikator, sa poljima za njegove atribute.



- Kada se leksičkom analizom, u programu detektuje identifikator, on se unosi u tabelu simbola.
- Tokom leksičke analize se ne mogu se odrediti atributi identifikatora.

Na primeru narebe u C-u:

double a, b, c;

Tip *double* nije poznat kada leksički analizator identifikuje promenljive a, b, c

- U narednim fazama se unose informacije o identifikatorima u tabelu simbola; tokom semantičke analize utvrđuju se prvo tipovi identifikatora zatim, da li ih program koristi na ispravan način i da li se ispravno izvršavaju sve operacije nad njima.

Detektovanje grešaka i obaveštavanje o njima

- Svaka faza može da otkrije greške.
- Medjutim, posle detektovanja neke greške, faza mora nekako da tretira grešku, tako da kompilacija može da se nastavi, da bi se omogućilo detektovanje ostalih grešaka u izvornom programu.
- Kompajler koji se zaustavlja kada otkrije prvu grešku nije naročito koristan

- Faze sintaksne i semantičke analize obično rukuju velikim delom grešaka koje kompajler detektuje.
- Leksička faza može da detektuje greške kada karakteri na ulazu ne formiraju ni jedan token jezika.
- Greške gde niz tokena krši strukturna pravila (sintaksu) jezika definišu se u fazi sintaksne analize.
- U toku semantičke analize, kompajler pokušava da detektuje konstrukcije koje imaju pravu sintaktičku strukturu bez značenja operacija, na primer, ako pokušamo da saberemo dva identifikatora, od kojih jedan predstavlja ime polja, a drugi ime procedure.

Generisanje medjukoda

- Posle sintaksne i semantičke analize, neki kompajleri generišu eksplicitnu medjupredstavu izvornog programa.
- Medjupredstavu možemo da zamislimo kao program za apstraktnu mašinu.
- Ova medjupredstava treba da ima dva važna svojstva: mora lako da se pravi i lako da se prevodi u ciljni program.

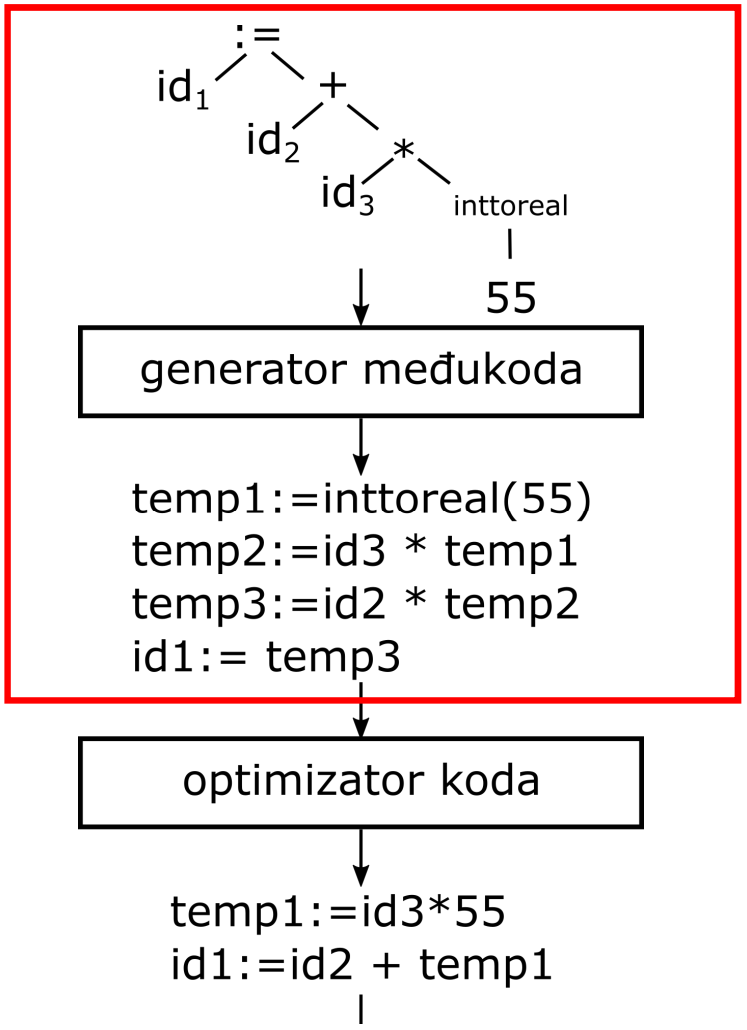
- Medjupredstava može imati različite forme.
- Razmotrićemo jednu medjuformu koja je nazvana "**troadresni kod**", koja je kao asemblerski jezik za mašinu kod koje svaka memorijska lokacija može da funkcioniše kao registar.
- Troadresni kod sastoji se od niza instrukcija od kojih svaka ima tri operanda.
- Izvorni program u troadresnom kodu:

temp1 := inttoreal(55)

temp2 := id3 * temp1

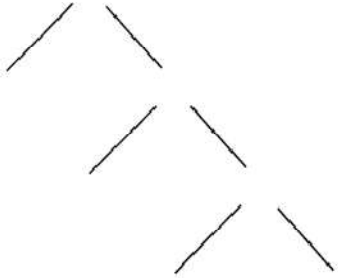
temp3 := id2 + temp2

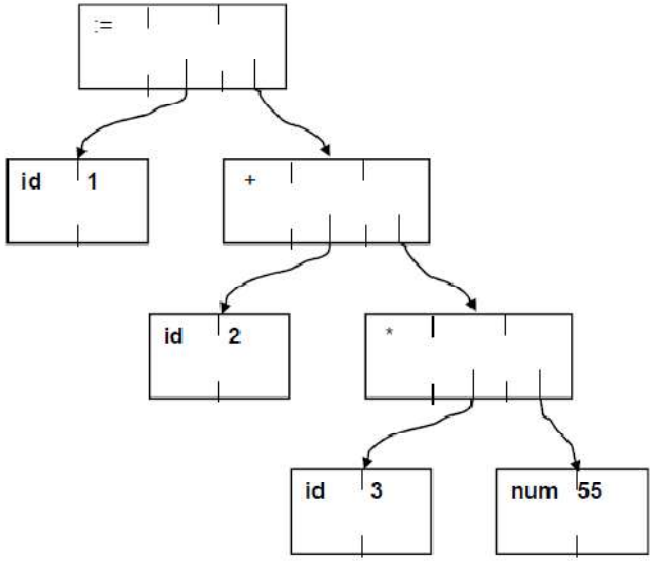
tid1 := temp3



- Pri generisanju ovih instrukcija kompajler odlučuje o redosledu izvršavanja operacija, npr. množenje prethodi sabiranju u izvornom programu.
- Drugo, kompajler mora da generiše privremeno ime da bi sačuvao vrednost izračunatu svakom instrukcijiom
- Neke troadresne instrukcije imaju manje od 3 operanda

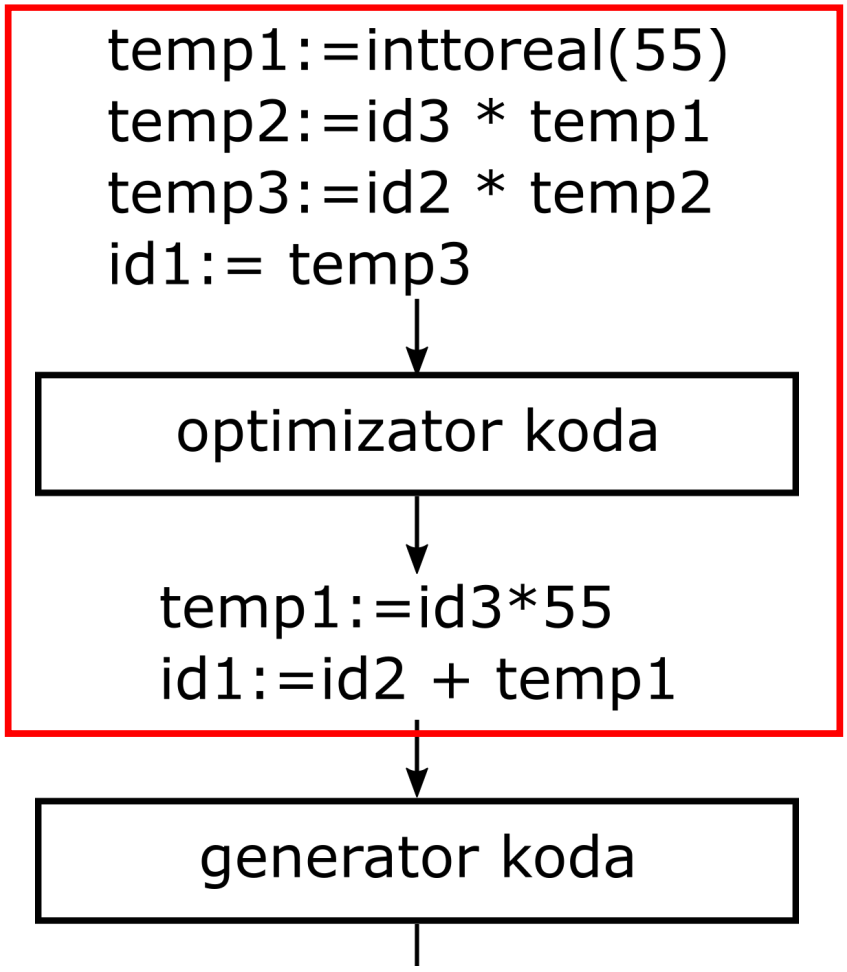
```
temp1 := inttoreal(55)
temp2 := id3 * temp1
temp3 := id2 + temp2
tid1 := temp3
```



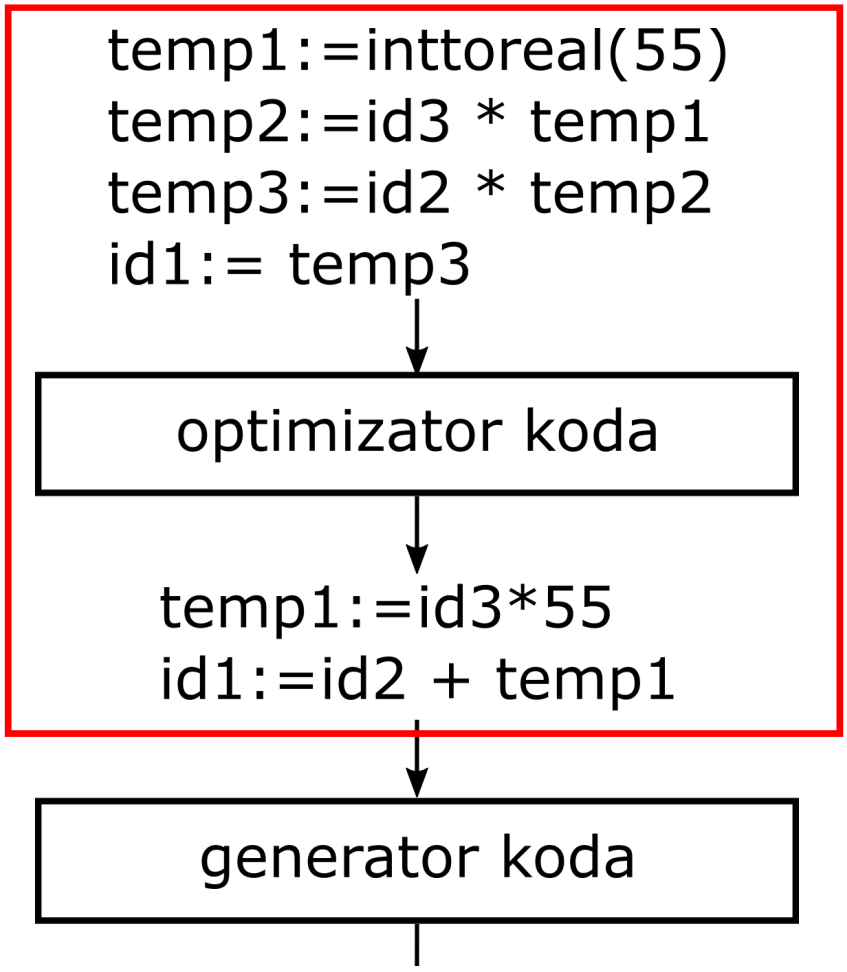


Optimizacija koda

- Faza optimizacije koda pokušava da poboljša medjukod tako da rezultuje bržim mašinskim kodom.
- Neke optimizacije su trivijalne. Na primer, prirodni algoritam generiše prethodni medjukod koristeći jednu instrukciju za svaki operator u predstavi stabla posle semantičke analize, čak i ako postoji bolji način da se ostvari isto izračunavanje, koristeći dve instrukcije.



- Nema ničeg pogrešnog u ovom algoritmu, s obzirom da se problem može otkriti tokom faze optimizacije koda.
- Dakle, kompajler može da zaključi da konverziju broja 55 iz *integer* tipa u *real* može da izvrši samo jednom, tako da operacija *inttoreal* može da bude eliminisana
- Pored toga *temp3* se koristi samo jednom, za prenošenje njegove vrednosti identifikatoru *id1*.
- Onda postaje bezbedno zameniti *id1* sa *temp3*.

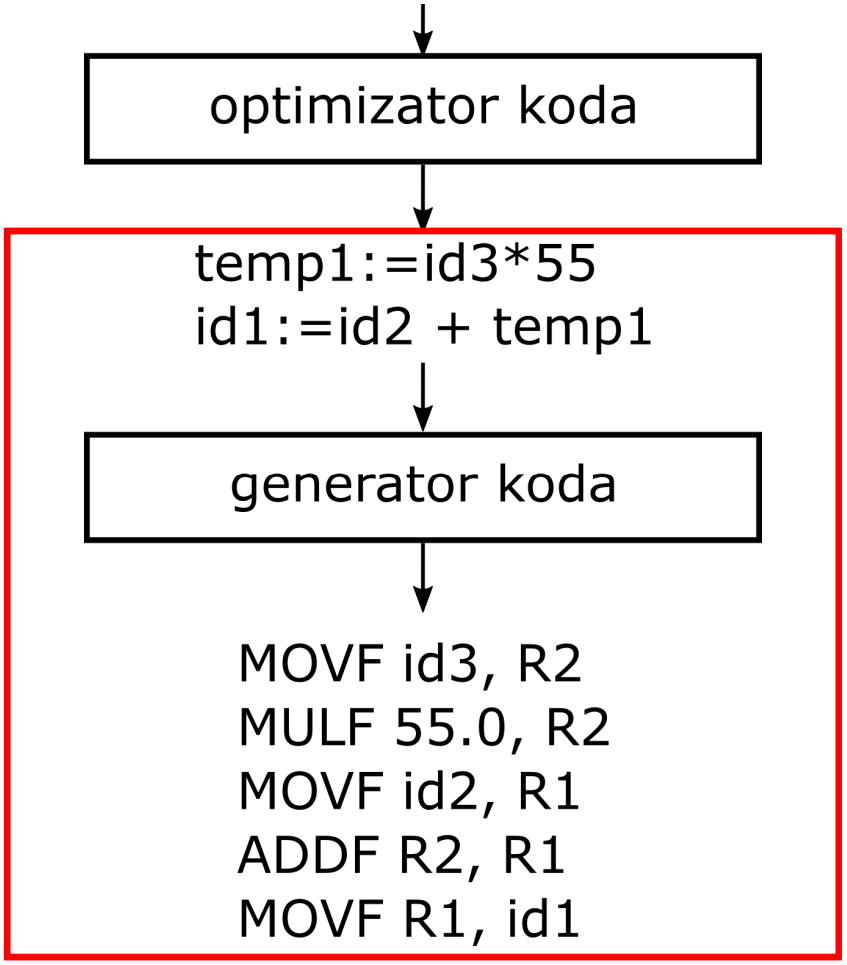


Ciljevi optimizacije koda:

- Uklanjanje redundantnog rada
  - nedostižnog koda
  - eliminisanja zajedničkih podizraza
  - eliminisanja indukcionih promenljivih
- Kreiranja jednostavnijih operacija
  - razmatranja konstanti u kompajleru
  - redukovanje množenja (konvertovanje u šiftovanje)
- Dobro upravljanje (dodeljivanje) registara

Generisanje koda

- Finalna faza kompajlera je generisanje ciljnog koda, koji se sastoji od prenosivog mašinskog koda ili asemblerskog koda.
- Memorijske lokacije se selektuju za svaku promenljivu korišćenu od strane programa
- Svaka medjuinstrukcija se prevodi u niz mašinskih instrukcija koje vrše isti zadatak.
- Najvažniji aspekt je dodeljivanje promenljivih registrima
- Na primer, koristeći registre 1 i 2, prevodjenje koda može postati:



- Prvi i drugi operand svake instrukcije specificira izvor i destinaciju, respektivno.
- U svakoj instrukciji, F nam kaže da instrukcija radi sa brojevima u pokretnom zarezu (floating point).
- Ovaj kod pomera sadržaj adrese id3 u registar 2, zatim ga množi sa realnom konstantom 55.0.
- Znak # znači da 55.0 treba tretirati kao konstantu.
- Treća instrukcija premešta id2 u registar 1 i dodaje mu veličinu prethodno izračunatu u registru 2.
- Konačno, veličina u registru 1 premešta se u adresu id1, tako da kod implementira dodeljivanje.

```
MOVF id3, R2
MULF #55.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Preprocesiranje

Preprocesori proizvode ulaz za kompajlere.

Obavljaju sledeće funkcije:

- **Makroprocesiranje.** Korisnik može da definiše makroe koji predstavljaju skraćenice za duže konstrukcije.
- **Uključivanje fajlova**. Preprocesor može da uključi *header* fajlove u tekst programa. Na primer, C preprocesor menja naredbu #include <global.h> sadržajem fajla global.h.
- **Proširenje jezika.** Dodaju nove jezičke sposobnosti. Na primer jezik Equel je jezik za obradu baza podataka koji je ugrađen u jezik C. Naredbe koje počinju ## sa preprocesor tretira kao naredbe za pristupanje bazi podataka, prevode se specijalnim programskim procedurama koje ostvaruju pristup bazama podataka.

Asembleri

- Asemblerski kod (**assembly code**) je mnemonička verzija mašinskog koda, u kojoj se imena koriste umesto binarnih kodova za operacije, a takodje se daju imena i memorijskim adresama.
- Tipičan niz asemblerskih instrukcija mogao bi da bude

MOV a, R1

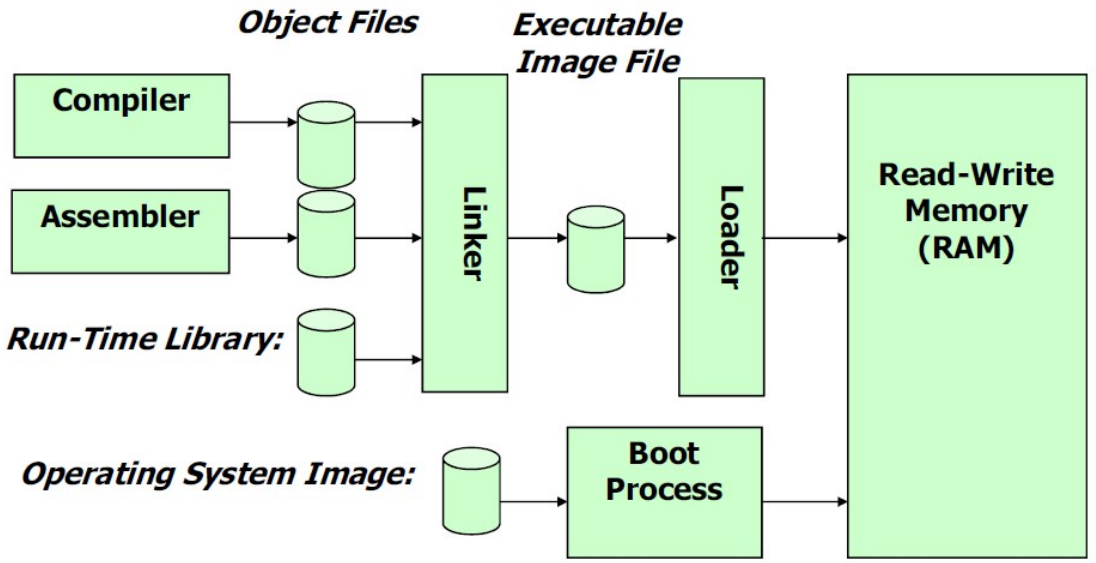
ADD #2, R1

MOV R1, b

- Najprostija forma asemblera vrši dva prolaza nad ulaznim kodom, gde se prolaz sastoji iz jednog čitanja ulaznog fajla.
- U prvom prolazu, pronalaze se svi identifikatori koji označavaju memorijske lokacije. Oni se memorišu u tabeli simbola.
- U drugom prolazu asembler ponovo skanira ulaz. Ovog puta, on prevodi svaki kod operacije u sekvencu bitova koja predstavlja tu operaciju u mašinskom jeziku.
- Takođe, u drugom prolazu svaki identifikator se prevodi u adresu memorijske lokacije koja odgovara tom identifikarou u tabeli simbola.
- Izlaz drugog prolaza je premestivi (*relocatable*) mašinski kod.

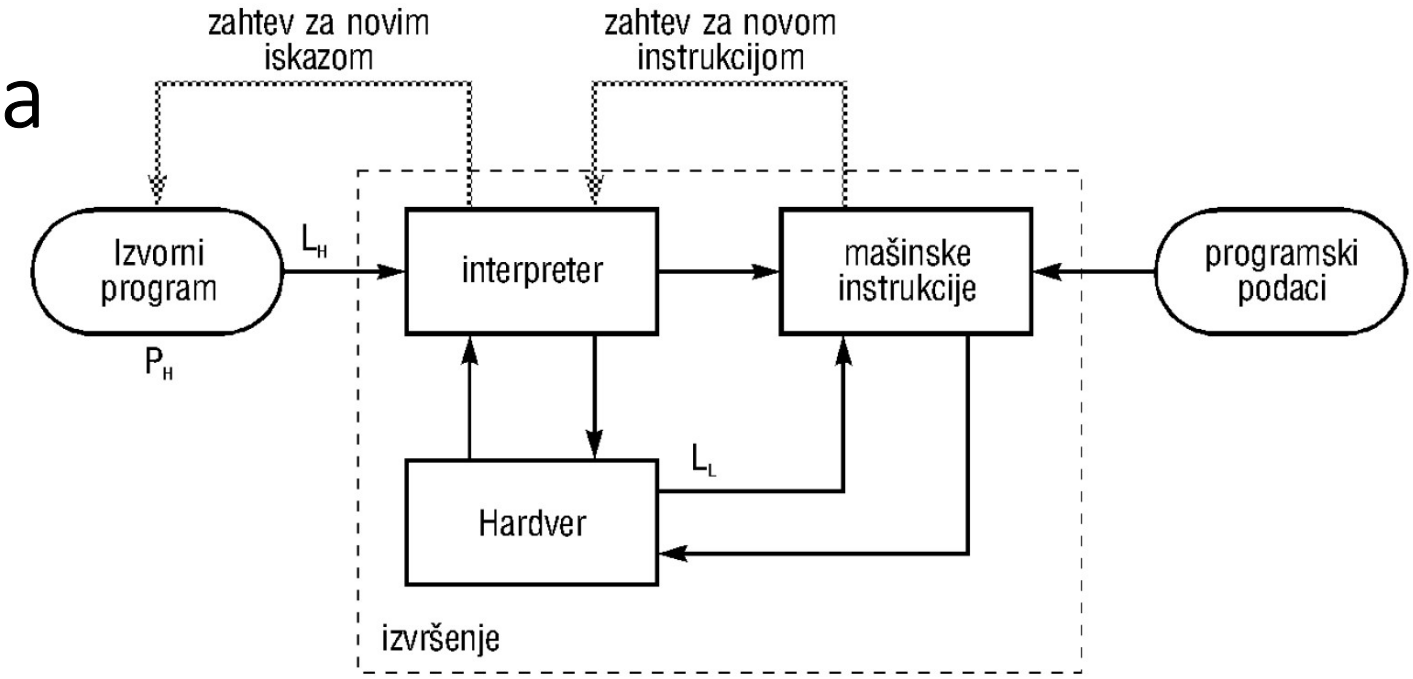
Linker/loader

- Obično, program koji se naziva linker obavlja dve funkcije: **učitavanje (load) i linkovanje**.
- Proces učitavanja sastoji se od uzimanja premestivog mašinskog koda, promene premestivih adresa, i smeštanja instrukcija sa promenjenim adresama u RAM memoriju na pogodnoj adresi.



- Linker nam **dopušta da napravimo jedan program** od više fajlova sa premestivim mašinskim kodom.
- Ovi fajlovi mogu da budu rezultat odvojenih procesa kompilacije.

Interpretacija



- Interpreter uzima jednu instrukciju iz programa, analizira je i uslovljava da se sa istim efektom izvrši niz instrukcija sa drugog nivoa.
- Ovaj proces se nastavlja sve dok se ne izvrši kompletan program.

- INTERPRETERI – programi prevodioci, koji za razliku od kompajlera **prevode i odmah izvršavaju svaku naredbu višeg programskog jezika**.
- Pomoću interpretera ne možemo dobiti program u mašinskom jeziku, nego se program svaki put mora ponovo prevesti interpreterom kada ga želimo izvršiti.
- Za razliku od interpretera, kod kompajlera su izvorni program i prevedeni program potpuno odvojeni i pri izvođenju nezavisni.
- Ako se izmeni izvorni program, to se neće automatski odraziti na izvršni program. Izvršni program je potrebno ponovno kompajlirati.
- Prednosti kompajlera: brži rad od interpretera i zaštićen izvorni program.
- Nedostaci kompajlera: odvojenost izvršnog i izvornog programa.

- GNU Compiler Collection ili GCC je, bez ikakve sumnje, **najmoćniji kompajler**.
- To je kamen temeljac **open-source** GNU platforme i korišten je za izgradnju gotovo svake moderne Linux mašine.
- GCC nudi paket kompajlera za standardne kompajlirane jezike, uključujući C, C++, Objective C, Ada, Pascal, Fortran i mnoge druge.
- Za većinu Linux distribucija podrazumevano je već instaliran GCC.
- Ako ste programer početnik, koristite GCC!.


That moment when its says error's On Line 86 but is actually on line 72!