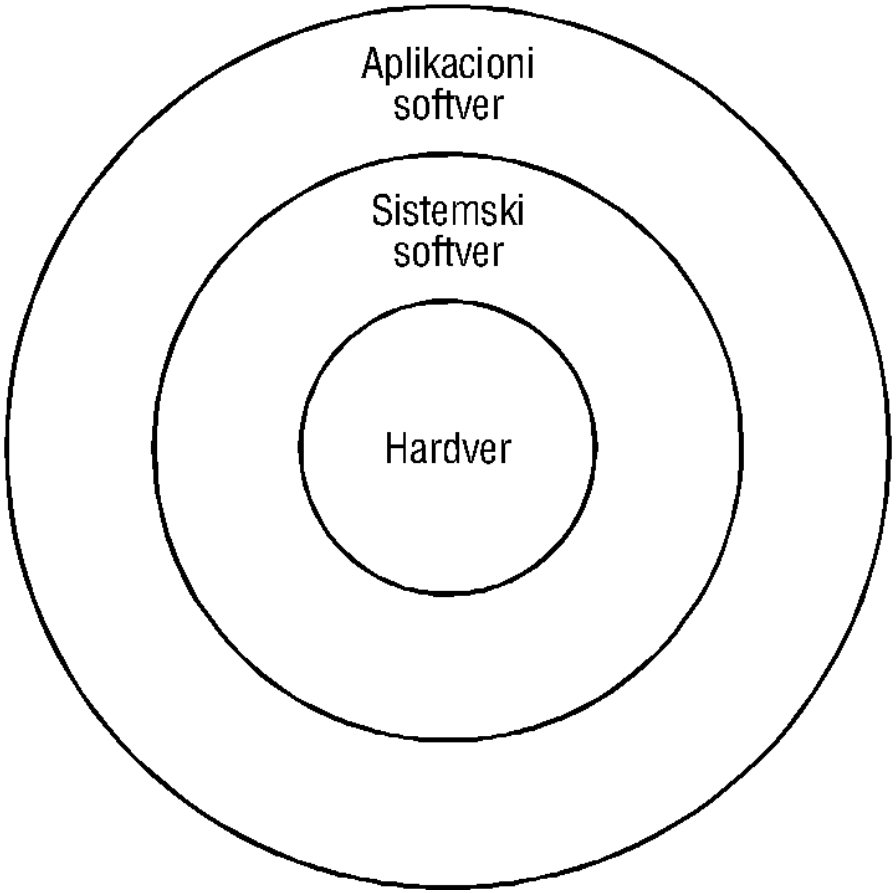
Compilers
-
Prevodioci

1

Odnos hardver, sistremski softver i aplikacioni softver kod računara:



2

Funkcija kompajlera:

3

Pascal iskaz preveden u asemlerski jezik, a nakon toga asembliran u mašinski jezik

Iskaz na višem programskom jeziku (Pascal)

```
C := A+B
```

Pascal kompilator

I korak

Programska sekvenca na asembler–skom jeziku za mašinu zasnovanu na mikroprocesoru MC68020

```
MOVE.W  B, D1
ADD.W   A, D1
MOVE.W  D1, C
```

Asembler

Program na binarnom mašinskom jeziku, za MC68020, u bitovima

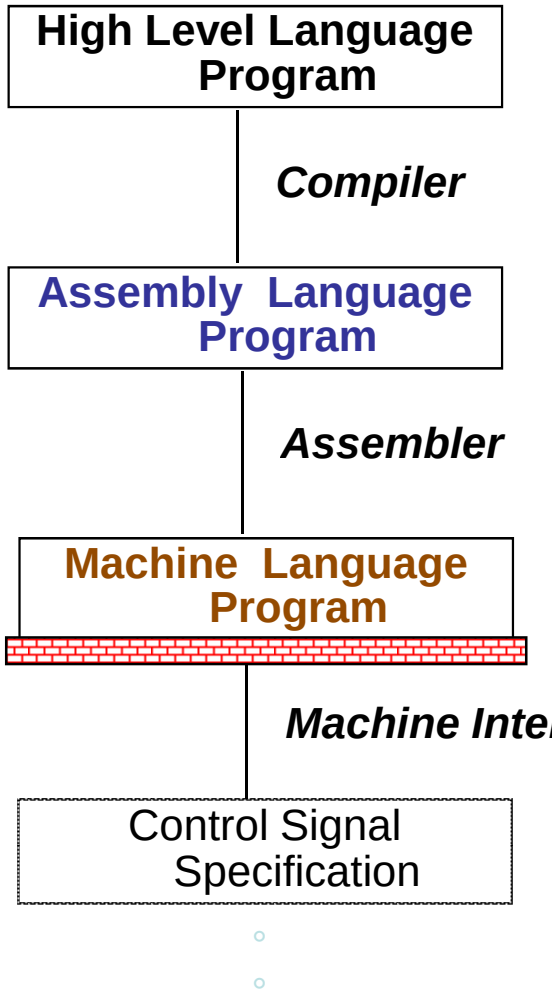
```
1000:   MOVE.W   (0X2002).W, D1
1004:   ADD.W    (0X2004).W, D1
1008:   MOVE.W   D1, (0X2000).W

2000:   C
2002:   B
2004:   A
```

```
0011 0010 0011 1000
0010 0000 0000 0010
1101 0010 0111 1000
0010 0000 0000 0100
0011 0001 1100 0001
0010 0000 0000 0000
```

II korak

Levels of Representation



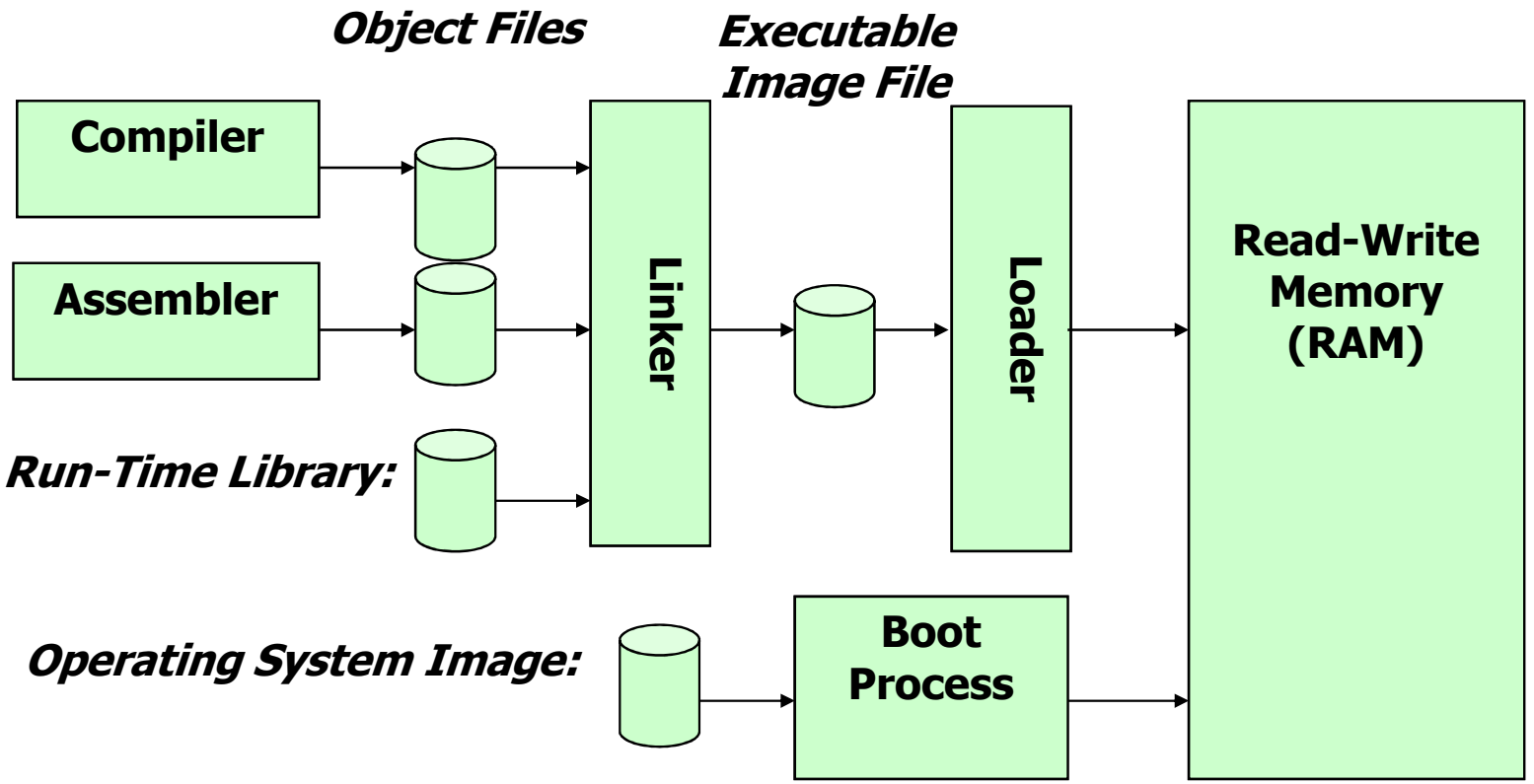

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $15,   0($2)
lw $16,   4($2)
sw $16,   0($2)
sw $15,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

ALUOP[0:3] <= InstReg[9:11] & MASK

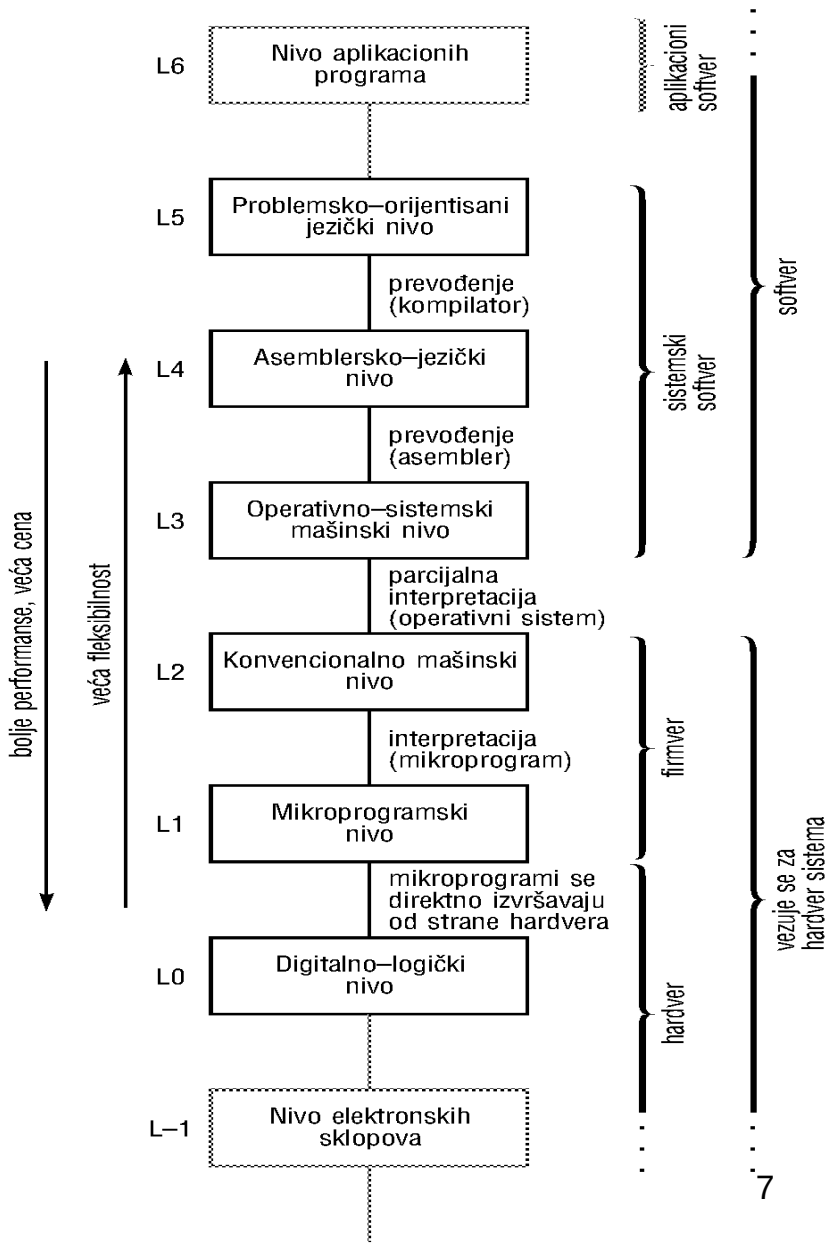
The build and load process for desktop application programs:



6

Hijerarhijska organizacija računara – koncept nivoa:

- Nivo aplikacionog programa
- Nivo višeg programskog jezika
- Nivo mašinskog kôda (asemblerski jezik)
- Upravljački nivo
- Nivo funkcionalne jedinice
- Logička kola, tranzistori i veze



Generacije računarskih jezika

- Globalno razlikujemo četiri klase računarskih jezika

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

8

Prva generacija – mašinski jezik

Karakteristike:

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

9

Druga generacija – asemblerski jezik

Karakteristike:

- svaka instrukcija se predstavlja mnemonikom, kao na primer ADD
- 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 – HLL

Karakteristike:

- **kompajler prevodi programske iskaze u odgovarajuće 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 programi**

Četvrta generacija – novi jezici

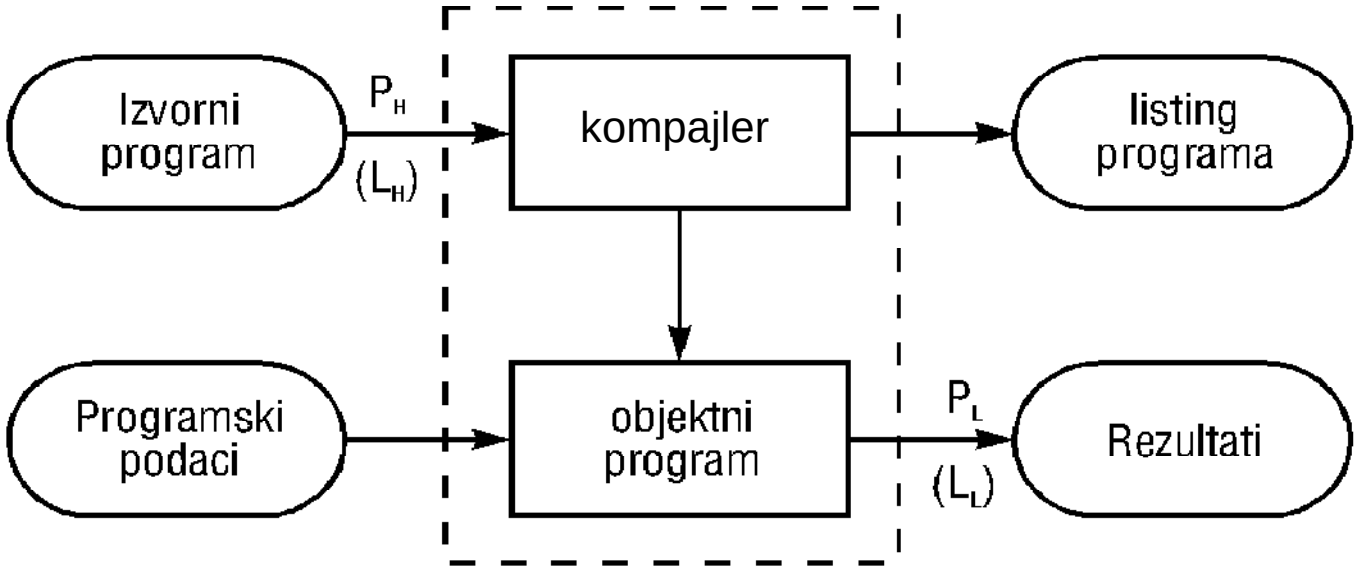
Novi tipovi računarskih jezika se karakterišu sledećim osobinama:

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

12

Mehanizmi za prevodjenje sa HLL-a na mašinski kôd:

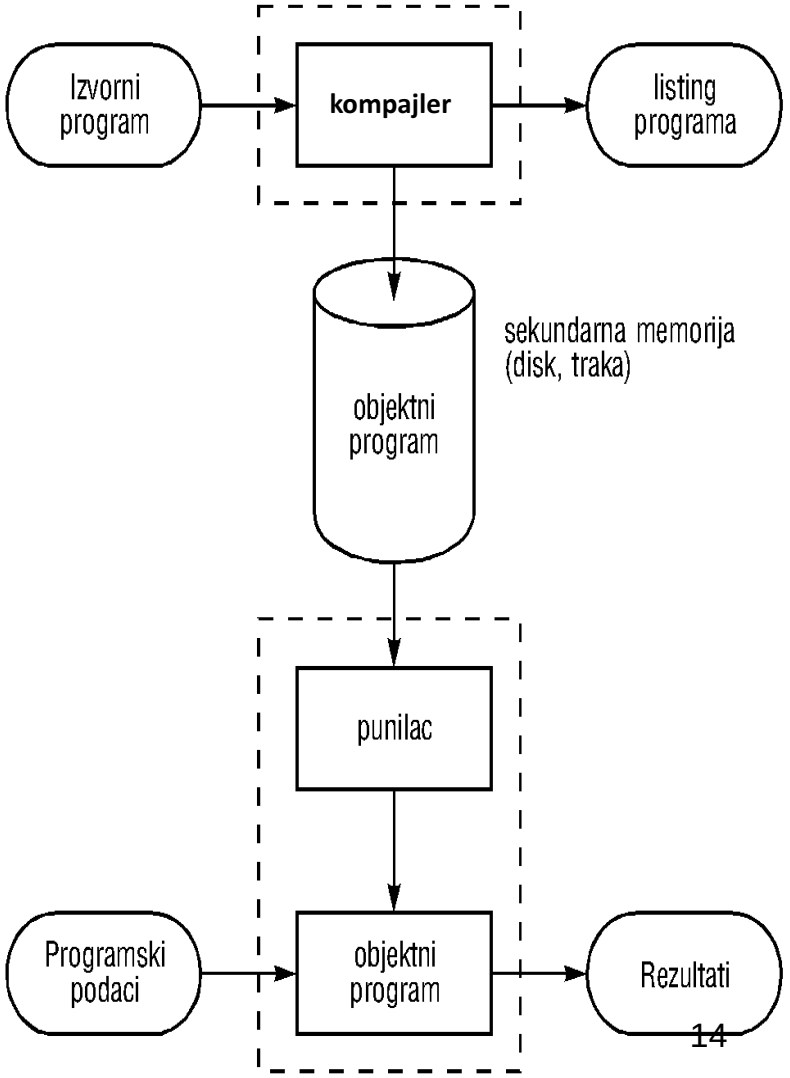
Kompajler je program koji na svom ulazu prihvata program napisan na nekom od programskih jezika nazvan izvorni program, a na svom izlazu generiše ekvivalentan program na mašinskom kôdu nazvan objektni program, tj. program u formi mašinskog kôda koji se može direktno izvršavati od strane hardvera računara.



13

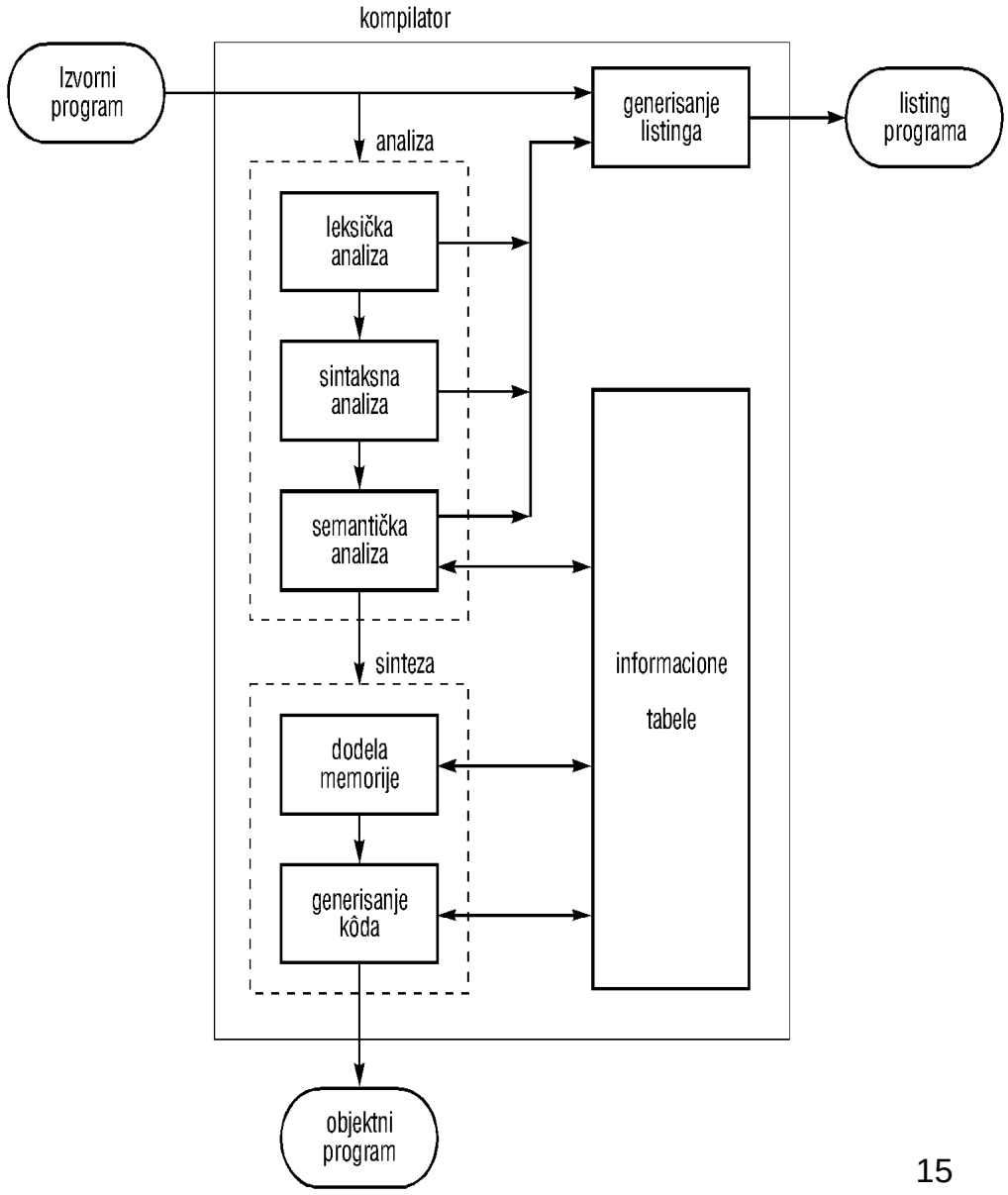
Ciklus kompilacije:

- **Objektni program koji se generiše na izlazu kompajlera upisuje se u sekundarnu memoriju.**
- **Korisnički program poznat kao punilac (*loader*) smešta program iz sekundarne memorije u glavnu čime je objektni program spreman za izvršenje**



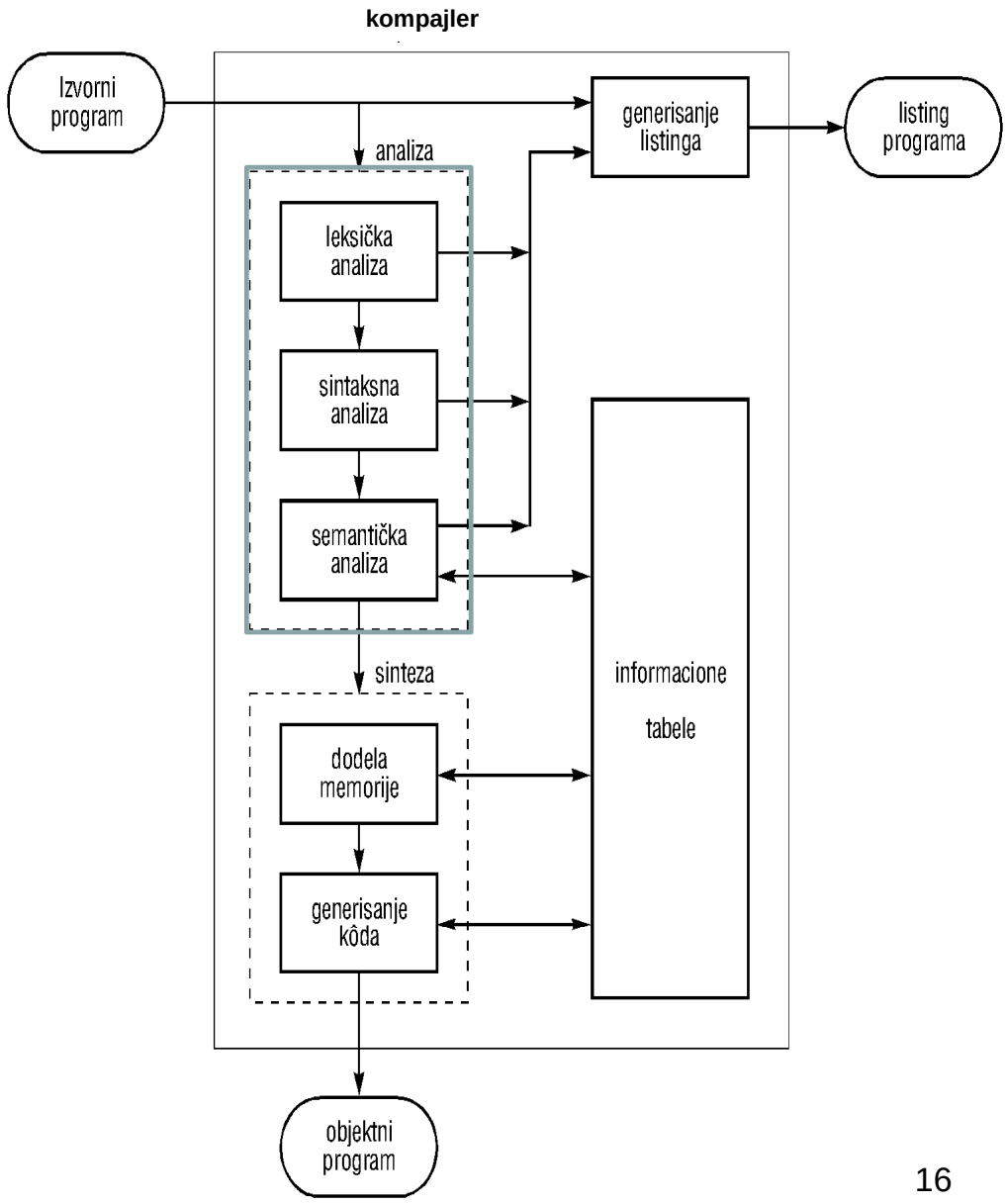
Proces kompilacije:

- **Proces kompilacije može se podeliti na dva glavna dela:**
  - **analiza izvornog programa**
  - **sinteza objektnog programa**



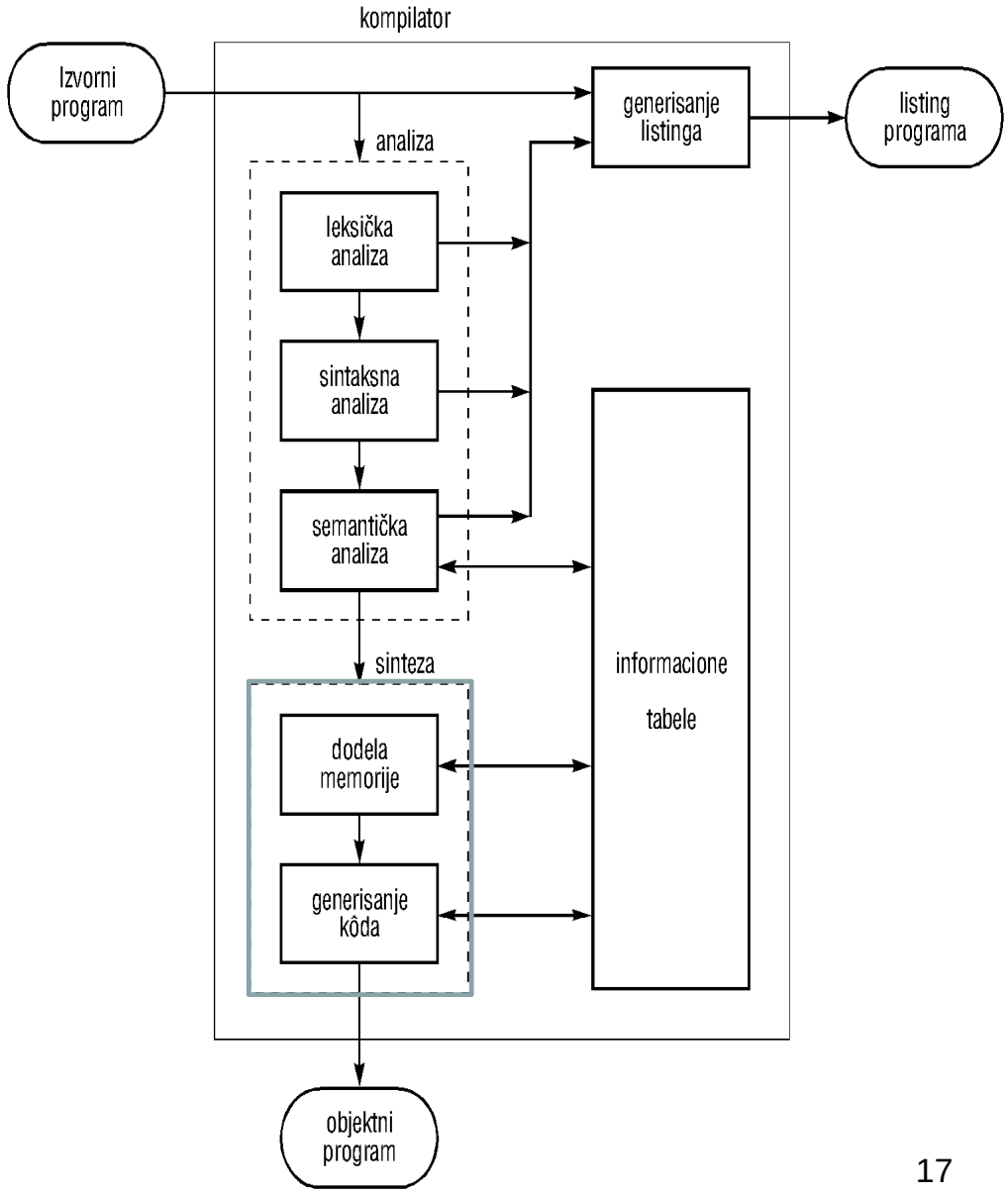
Proces kompilacije:

- **Proces kompilacije može se podeliti na dva glavna dela:**
  - **analiza izvornog programa**
  - **sinteza objektnog programa**

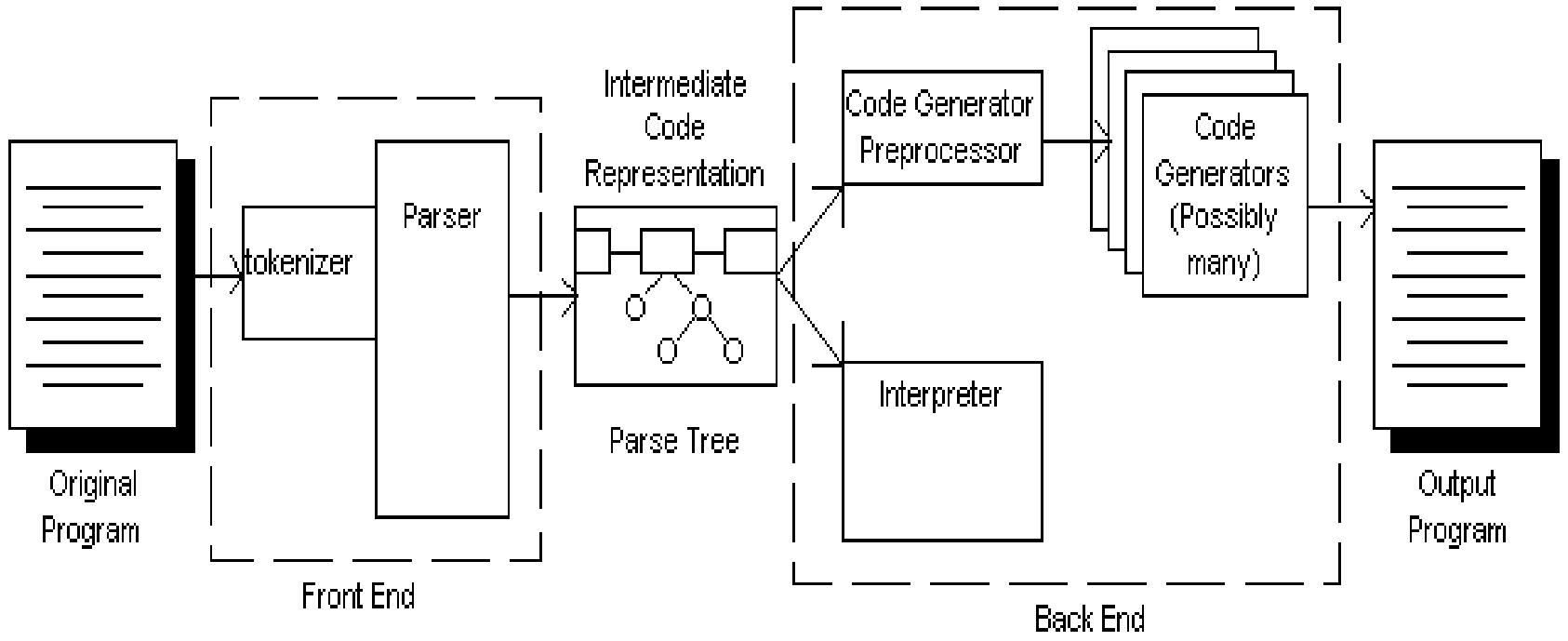


Proces kompilacije:

- **Proces kompilacije može se podeliti na dva glavna dela:**
  - **analiza izvornog programa**
  - **sinteza objektnog programa**



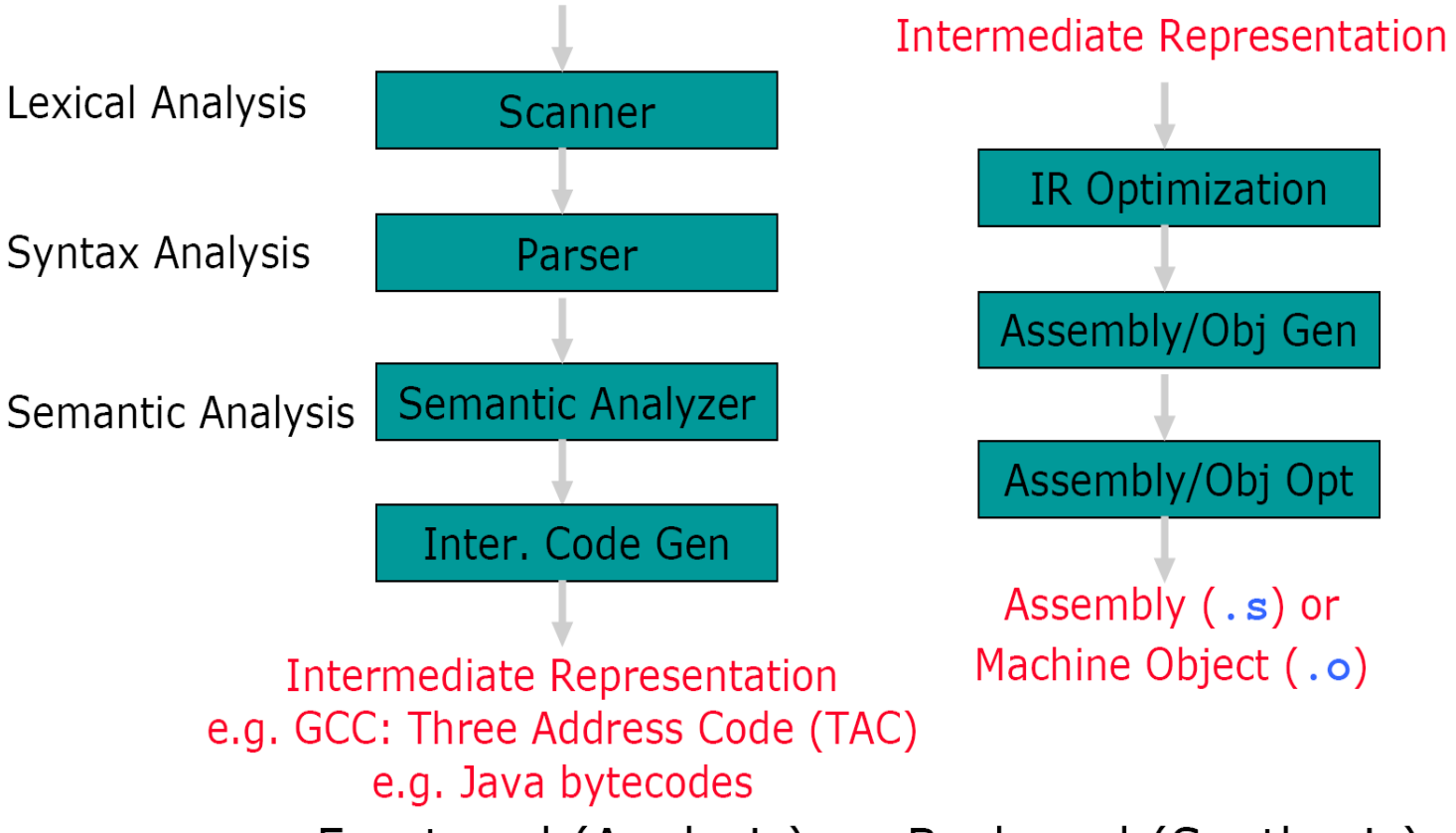
Struktura kompajlera:



18

Faze kompajlera:

- Compiler phases



20

Analiza izvornog programa

- **Kod kompajliranja, analiza se sastoji od tri faze:**
- ***Linearnna 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 karakteri ili tokeni grupišu hijerarhijski u ugnježdene skupove sa zajedničkim značenjem.**
- ***Semantička analiza*, kod koje se izvode neke provere da bi se osiguralo da komponente programa zajedno imaju značenje.**

21

Leksička analiza:

U kompajleru, linearna analiza se naziva *leksičkom analizom* ili *skaniranjem*. Na primer, kod leksičke analize karakteri u naredbi dodeljivanja

a := b + c*55

biće grupisani u naredne tokene:

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 normalno eliminisani u toku leksičke analize.

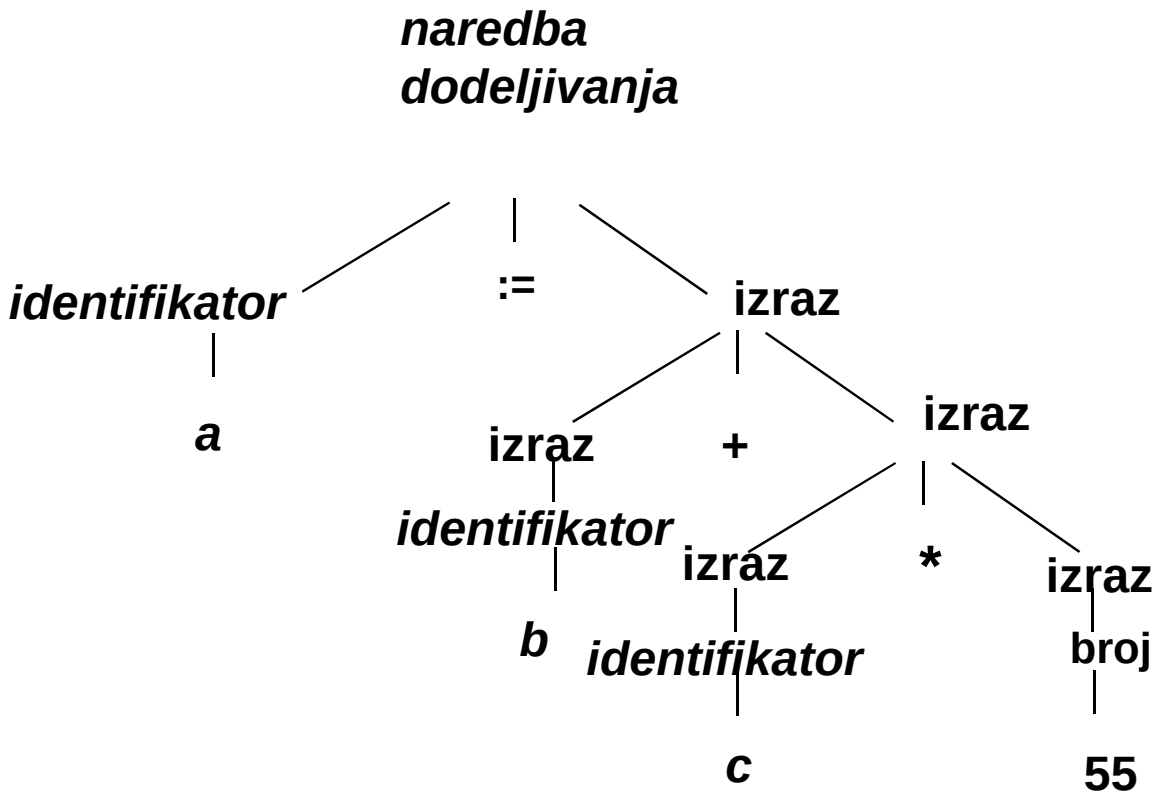
22

Sintaksna analiza

Hijerarhijska analiza naziva se *raščlanjivanjem (parsing)* ili *sintaksnom analizom*. Ona podrazumeva grupisanje tokena izvornog programa u gramatičke fraze koje se koriste od strane kompajlera za sintetizovanje izlaza. Obično se gramatičke fraze izvornog programa predstavljaju parserskim stablom kao što je prikazano na sledećoj slici.

23

Parsersko stablo za a := b + c*55



Rekurzivna pravila:

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

- Bilo koji *identifikator* je izraz.
- Bilo koji *broj* je izraz.
- Ako su *izraz1* i *izraz2* izrazi, onda je izraz takodje i

  izraz1 + izraz2

  *izraz1 * izraz2*

  (*izraz1*)

25

Rekurzivna pravila:

Slično, mnogo jezika definiše naredbe rekurzivno pomoću pravila kao što su:

- **Ako *identifikator1* je identifikator, a *izraz2* je izraz, onda**

  identifikator1* := *izraz2

  je naredba.
- **Ako *izraz1* je izraz, a *naredba2* je naredba, onda**

  while (*izraz1*) do *naredba2*

  if (*izraz1*) then *naredba2*

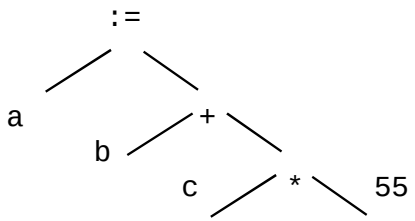
  su naredbe.

Semantička analiza

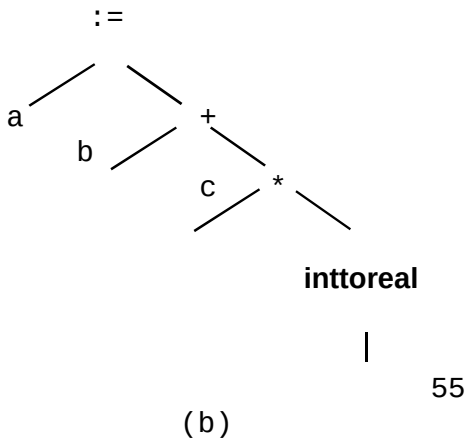
- **Semantičkom analizom proverava se da li izvorni program ima semantičkih grešaka i prikuplja se informacija o tipu za kasniju fazu generisanja koda. Koristi se hijerarhijska struktura odredjena sintaksnom analizom za identifikovanje operatora i operanada izraza i naredbi.**
- ***Važna komponenta semantičke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

27

Semantička analiza:



(a)



(b)

Tabela simbola:

- ***Tabela simbola*** **je struktura podataka koja sadrži zapis za svaki identifikator, sa poljima za atribute identifikatora.**
- **Ovi atributi mogu da obezbede informaciju o memoriji dodeljenoj za identifikator, njegov tip, njegov opseg (gde u programu je on važeći), i, u slučaju imena procedura, stvari kao broj i tipove argumenata, način prenošenja svakog argumenta (na pr. pomoću reference), i vraćeni tip, ako postoji.**

29

Detektovanje grešaka i obaveštavanje o njima, 1#:

- **Svaka faza može da otkrije greške. Medjutim posle detektovanja 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 tako koristan kao što bi mogao da bude.**

30

Detektovanje grešaka i obaveštavanje o njima, 2#:

- **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.**

31

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.

32

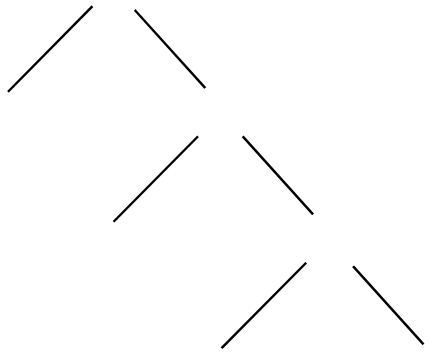
Medjukod:

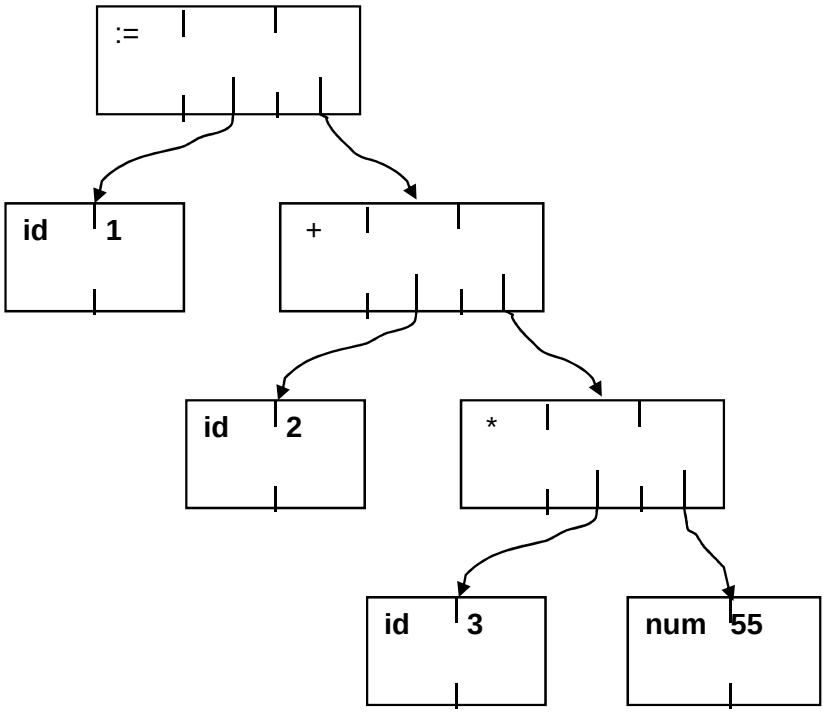
Medjupredstava može imati različite forme.

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 (*.1) može se pojaviti u troadresnom kodu kao

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

Generisanje medjukoda:





34

Optimizacija koda:

Faza optimizacije koda pokušava da poboljša medjukod, tako da rezultuje u bržem mašinskom kodu. 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

- **temp1 := id3 * 55**
- **id1 := id2 + temp1**

35

Još jedan primer generisanja međukoda:

- Three-address code (TAC)

```
j = 2 * i + 1;
if (j >= n)
   j = 2 * i + 3;
return a[j];
```

```
        t1 = 2 * i
        t2 = t1 + 1
        j = t2
        t3 = j < n
        if t3 goto L0
        t4 = 2 * i
        t5 = t4 + 3
        j = t5
L0:     t6 = a[j]
        return t6
```

Optimizacija koda:

- **Cilj: popraviti karakteristike pomoću:**

- Uklanjanja redundantnog rada

- nedostižnog koda
- eliminisanja zajedničkih podizraza
- eliminisanja indukcionih promenljivih

- Kreiranja jednostavnijih operacija

- razmatranja konstanti u kompajleru
- redukovanja množenja (konvertovanja u šiftovanje)

- Dobrog upravljanja (dodeljivanja) registrima

37

Optimizacija koda

- Example

```
        t1 = 2 * i
        t2 = t1 + 1
        j = t2
        t3 = j < n
        if t3 goto L0
        t4 = 2 * i
        t5 = t4 + 3
        j = t5
L0:     t6 = a[j]
        return t6
```

```
        t1 = 2 * i

        j = t1 + 1
        t3 = j < n
        if t3 goto L0

        j = t1 + 3

L0:     t6 = a[j]
        return t6
```

- Copy propagation
- Common subexpression elimination

38

Optimizacija kompajlera:

- **Omogućiti efikasno preslikavanje programa na mašinu**
  - selekcija i uređenje koda
  - eliminacija minornih neefikasnosti
  - dodela registara
- **Ne popravljati asimptotsku efikasnost**
  - na programeru je selekcija najboljeg ukupnog algoritma
  - često je bolje postići krajnju efikasnost nego konstantne faktore
- **Tipovi optimizacije**
  - lokalni: unutar osnovnih blokova
  - globalni: u petljama

39

Limitations of Optimizing Compilers

- Operate Under Fundamental Constraints
  1. Improved code has same output
  2. Improved code has same side effects
  3. Improved code is as correct as original code

- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs
  - Profile driven compilation becoming more prevalent (JIT)

- **When in doubt, the compiler must be conservative**

40

Common Sub-Expression Elimination

```
main() {
   int x, y, z;
   // They get
   // initial values
   x = (1+20)* -x;
   y = (x*x)+(x/y);
   y = z = (x/y)/(x*x);
}
```

```
tmp1 = 1 + 20 ;
tmp2 = -x ;
x = tmp1 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
tmp6 = x * x ;
z = tmp5 / tmp6 ;
y = z ;
```

46

Common Sub-Expression Elimination – cont.

```
t1 = 1 + 20 ;
t2 = -x ;
x = t1 * t2 ;
t3 = x * x ;
t4 = x / y ;
y = t3 + t4 ;
t5 = x / y ;
t6 = x * x ;
z = t5 / t6 ;
y = z ;
```

```
t2 = -x ;
x = 21 * t2 ;
t3 = x * x ;
t4 = x / y ;
y = t3 + t4 ;
t5 = x / y ;
z = t5 / t3 ;
y = z ;
```

47

Induction Variable Elimination

```
while (i<100) arr[i++] = 0;
```

- Remember replacing multiply with add in loop?
- Why not get rid of induction variable i altogether?

```
   t4 = arr ;
L0:   t2 = i < 100;
   IfZ t2 Goto _L1 ;
   *t4 =  0;
   t4 = t4 + 4;
   i = i + 1 ;
L1:
```

```
   t4 = arr ;
L0:   t2 = t4 < 400;
   IfZ t2 Goto _L1 ;
   *t4 =  0;
   t4 = t4 + 4;
L1:
```

48

50

Generisanje koda:

- **Finalna faza kompajlera je generisanje ciljnog koda koji se normalno sastoji od prenosivog mašinskog koda ili asemblerskog koda. Memorijske lokacije selektuju se za svaku promenljivu korišćenu od strane programa. Zatim, svaka medjuinstrukcija prevodi se u niz mašinskih instrukcija koje vrše isti zadatak. Najvažniji aspekt je dodeljivanje promenljivih registrima.**

51

Generisanje koda:

Na primer, koristeći registre 1 i 2, prevodjenje koda (*.4) može postati

- **MOVF id3, R2**
- **MULF #55.0, R2**
- **MOVF id2, R1** **(*.5)**
- **ADDF R2, R1**
- **MOVF R1, id1**

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.

52

Assembly Code Generation

- Example: a in $a1, i in $a0, n in $a2

```
        t1 = 2 * i

        j = t1 + 1
        t3 = j < n
        if t3 goto L0

        j = t1 + 3

L0:     t6 = a[j]
        return t6
```

```
        slli  $t0, $a0, 1

        addiu $t2, $t0, 1
        slt   $t2, $t2, $a2
        bne   $t2, $zero, L0
        addiu $t2, $t0, 3

L0:     slli  $t2, $t2, 2
        addu  $t2, $t2, $a1
        lw    $v0, 0($t2)
        jr    $ra
```

- Register allocation

Asembler

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**

Ovaj kod pomera sadržaj adrese a u registar 1, zatim dodaje konstantu 2 tome sadržaju, tretirajući sadržaj registra 1 kao broj u fiksnom zarezu, i konačno memoriše rezultat u lokaciji nazvanoj b. Dakle, izračunato je b := a+2.

55

Mašinski kod

Hipotetički mašinski kod u koji može biti prevedena asemblerska instrukcija može biti:

0001 01 00 00000000 *

0011 01 10 00000010

0010 01 00 00000100 *

Uočavamo malu instrukcionu reč, kod koje prva četiri bita predstavljaju kod instrukcije, gde su 0001, 0010 i 0011 predstavljaju učitati, memorisati i sabrati, respektivno. Pod učitati i memorisati podrazumevamo pomeranje iz memorije u registar i obrnuto. Sledeća dva bita imenuju registar, a 01 odnosi se na registar 1 u svakoj od gornjih instrukcija. Naredna dva bita predstavljaju "tag" ("značku"), gde 00 stoji za obični način adresiranja, a zadnjih osam bitova odnose se na memorijsku adresu. Tag 10 znači "neposredni" način i tada zadnjih osam bitova uzimaju se literalno kao operand. Ovaj način javlja se u drugoj instrukciji

Pseudoinstructions

- Assembler expands *pseudoinstructions*

```
move $t0, $t1                # Copy $t1 to $t0

addu $t0, $zero, $t1         # Actual instruction
```

- Some pseudoinstructions need a temporary register
  - Cannot use `$t`, `$s`, etc. since they may be in use
  - The `$at` register is reserved for the assembler

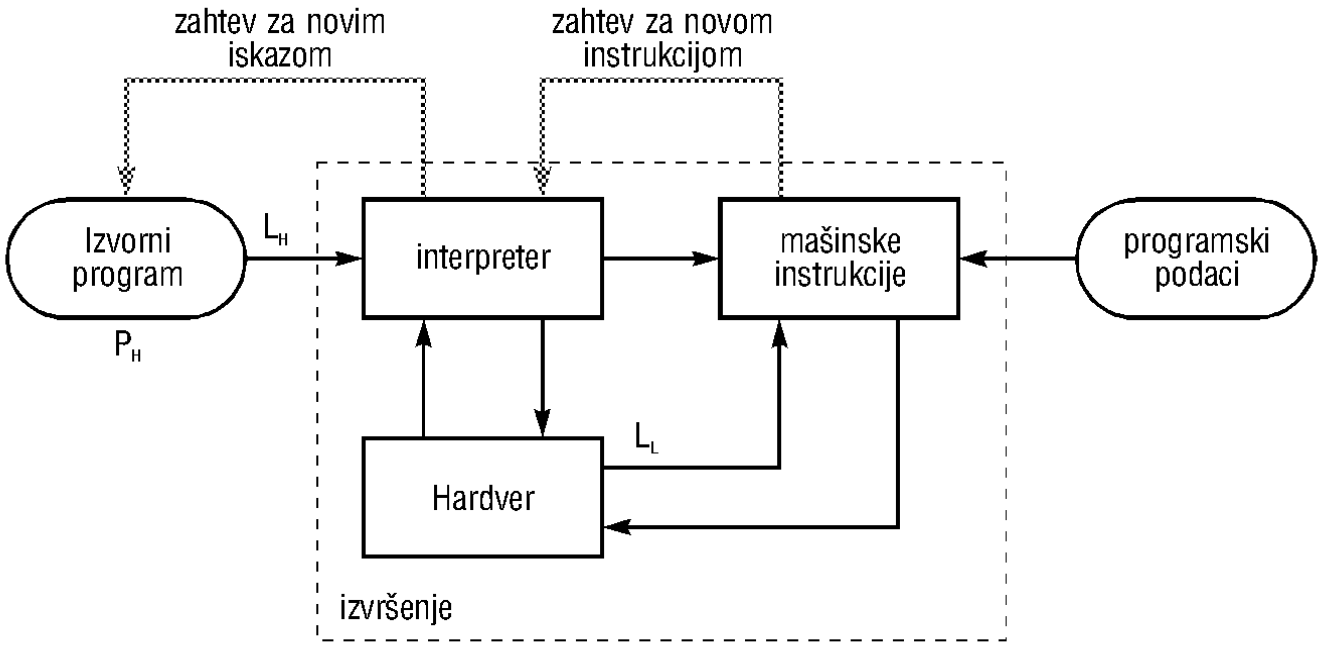
```
blt $t0, $t1, L1     # Goto L1 if $t0 < $t1

slt $at, $t0, $t1    # Set $at = 1 if $t0 < $t1
bne $at, $zero, L1   # Goto L1 if $at != 0
```

57

Interpretacija

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



61

Interpreteri

- **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 kada ga želimo izvršiti mora ponovo prevesti interpreterom.**

- **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 izvedbenom programu, nego ga je potrebno ponovno kompajlirati.**

- **Prednosti kompajlera: brži rad od interpretera i zaštićen izvorni program.**

- **Nedostaci kompajlera: odvojenost prevedenog i izvornog programa.**

62

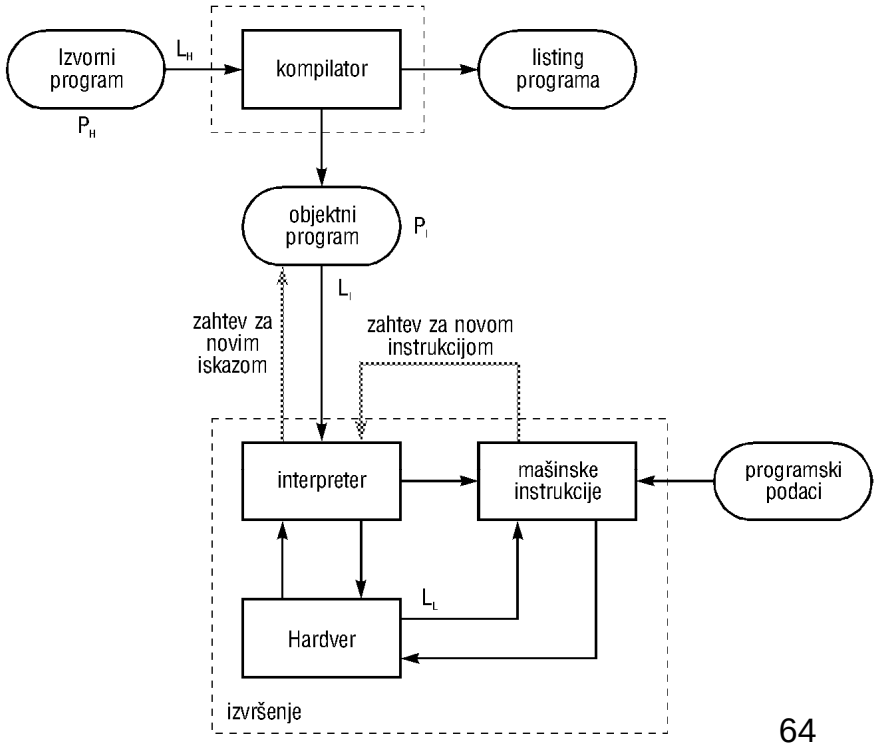
Interpretacija - prednosti i nedostaci

- Prednost interpeterskog mehanizma konverzije je ta što je interpreter relativno mali i što se u odnosu na kompajler lakše imlementira na nivou mašine.
- Nedostatak je taj što je izvršenje izvornih programa postupkom interpretacije sporije kod kompilovanih programa.
- Razlika u brzini izvršenja je reda 10.

63

Kombinovanje interpetacije i kompilacije

Kombinovani mehanizam kompilacija-interpretacija koristi među-jezik L_I, koji se nalazi između nivoa L_L i L_H. Program P_H, napisan na jeziku L_H, kompajlira se u program P_I na jeziku L_I. Program P_I se zatim intepretira na nivou L_L pomoću interpretera .



Prednost ove metode ogleda se u boljoj prenosivosti programa, kao i u tome što se zadržavaju ostale dobre osobine tehnika kompilacije i interpretacije.

64

66