Programski prevodioci

dr Miroslav Hajduković
mr Zorica Suvajdžin

Praktični uvod u programske prevodioce
s ciljem predstavljanja tipičnih problema
i načina njihovog rešavanja

Novi Sad 2004.

Uvod

- Zadatak programskih prevodilaca je da:
  - prevode programe koji su napisani jednim – **izvornim programskim jezikom** u programe istog značenja koji su napisani drugim – **ciljnim programskim jezikom**
- Primer izvornog jezika: *C*
- Primer ciljnog jezika: hipotetski asemblerski jezik
- Za prevođenje je potrebno:
  - uspostaviti korespondenciju iskaza izvornog i ciljnog programskog jezika
  - prepoznati iskaze izvornog programskog jezika i zameniti ih korespondentnim iskazima ciljnog programskog jezika

Hipotetski asemblerski jezik

- Podrazumeva se da registri i memorijske lokacije zauzimaju po 4 bajta
- Ukupno ima 16 registara
  - oznaka registra se sastoji od oznake `%` i rednog broja registra: `%0`, `%1`, ..., `%15`
  - registri `%0` do `%12` imaju opštu namenu i služe kao radni registri
  - registar `%13` rezervisan je za povratnu vrednost funkcije
  - registar `%14` služi kao pokazivač frejma
  - registar `%15` služi kao pokazivač steka
- Labele započinju malim slovom iza koga mogu da slede mala slova, cifre i podcrta '_' (alfabet je 7 bitni *ASCII*); iza labele se navodi dvotačka

Hipotetski asemblerski jezik

- Operandi
  - **neposredni operand** odgovara celom (označenom ili neoznačenom) broju:

    `$0` ili `$-152`

    a njegova vrednost vrednosti tog broja
  - **registarski operand** odgovara oznaci registra, a njegova vrednost sadržaju tog registra
  - **direktni operand** odgovara labeli. Njegova vrednost odgovara adresi labele, ako ona označava naredbu i koristi se kao operand naredbe skoka ili poziva potprograma. Ako direktni operand odgovara labeli koja označava direktivu i ne koristi se kao operand naredbe skoka ili poziva potprograma, njegova vrednost odgovara sadržaju adresirane lokacije.

Hipotetski asemblerski jezik

- **indirektni operand** odgovara oznaci registra navedenoj između malih zagrada:

  `(%0)`

  a njegova vrednost sadržaju memorijske lokacije koju adresira sadržaj registra
- **indeksni operand** započinje celim (označenim ili neoznačenim) brojem ili labelom iza čega sledi oznaka registra navedena između malih zagrada:

  `-8(%14)` ili **`4(%14)`** ili **`tabela(%0)`**

  Njegova vrednost odgovara sadržaju memorijske lokacije koju adresira zbir vrednosti broja i sadržaja registra, odnosno zbir adrese labele i sadržaja registra
- operandi se dele na ulazne (neposredni, registarski, direktni, indirektni i indeksni) i izlazne (registarski, direktni, indirektni i indeksni)

Hipotetski asemblerski jezik

- **naredba poređenja** brojeva postavlja bite status registra u skladu sa razlikom prvog i drugog ulaznog operanda

  `CMPx  ulazni operand, ulazni operand`

  `x:` **`S`** označeni

  `U` neoznačeni

  `F` realni (mašinska normalizovana forma)

Hipotetski asemblerski jezik

- **naredba bezuslovnog skoka** smešta u programski brojač vrednost ulaznog operanda (omogućujući tako nastavak izvršavanja od ciljne naredbe koju adresira ova vrednost)

```
JMP    ulazni operand
```

Hipotetski asemblerski jezik

- **naredbe uslovnog skoka** smeštaju u programski brojač vrednost ulaznog operanda samo ako je ispunjen uslov određen kodom naredbe (ispunjenost uslova zavisi od bita status registra)

```
JEQ    ulazni operand
JNE    ulazni operand
JGTx   ulazni operand
JLTx   ulazni operand
JGEx   ulazni operand
JLEx   ulazni operand
```

x: **S** označeni
U neoznačeni
F realni

Hipotetski asemblerski jezik

- **naredbe rukovanja stekom** omogućuju smeštanje na vrh steka vrednosti ulaznog operanda, odnosno preuzimanje vrednosti sa vrha steka i njeno smeštanje u izlazni operand (podrazumeva se da **%15** služi kao pokazivač steka, da se stek puni od viših lokacija ka nižim i da **%15** pokazuje vrh steka)

```
PUSH   ulazni operand
POP    izlazni operand
```

Hipotetski asemblerski jezik

- **naredba poziva potprograma** smešta na vrh steka zatečeni sadržaj programskog brojača, a u programski brojač smešta vrednost ulaznog operanda:

```
CALL    ulazni operand
```

- **naredba povratka iz potprograma** preuzima vrednost sa vrha steka i smešta je u programski brojač

```
RET
```

Hipotetski asemblerski jezik

- **naredba za sabiranje brojeva**

`ADDx  ulazni operand, ulazni operand, izlazni operand`

omogućuje sabiranje ulaznih operanada, uz izazivanje izuzetka ako se javi izlazak van opsega

- **naredba za oduzimanje brojeva**

`SUBx  ulazni operand, ulazni operand, izlazni operand`

omogućuje oduzimanje ulaznih operanada, uz izazivanje izuzetka ako se javi izlazak van opsega

`x:` **`S`** označeni

`U` neoznačeni

`F` realni (mašinska normalizovana forma)

Hipotetski asemblerski jezik

- **naredba za množenje brojeva**

`MULx  ulazni operand, ulazni operand, izlazni operand`

omogućuje množenje ulaznih operanada, uz izazivanje izuzetka ako proizvod ne može da stane u izlazni operand

- **naredba za delenje brojeva**

`DIVx  ulazni operand, ulazni operand, izlazni operand`

omogućuje delenje prvog ulaznog operanda drugim i smeštanje količnika u izlazni operand

`x:` **`S`** označeni

`U` neoznačeni

`F` realni (mašinska normalizovana forma)

Hipotetski asemblerski jezik

- **naredba za prebacivanje vrednosti**

```
MOV    ulazni operand, izlazni operand
```

- **naredba za prebacivanje adrese labele**

```
MOVA   labela, izlazni operand
```

Hipotetski asemblerski jezik

- **naredba konverzije celog broja u razlomljeni broj**

```
TOF   ulazni operand, izlazni operand
```

(vrednost ulaznog operanda je celi broj, a vrednost izlaznog operanda je ekvivalentni razlomljeni broj u mašinskoj normalizovanoj formi)

- **naredba konverzije razlomljenog broja u celi broj**

```
TOI   ulazni operand, izlazni operand
```

(vrednost ulaznog operanda je razlomljeni broj u mašinskoj normalizovanoj formi, a vrednost izlaznog operanda je ekvivalentni celi broj ako je konverzija moguća, inače se izaziva izuzetak)

Hipotetski asemblerski jezik

- **direktiva zauzimanja memorijskih lokacija** omogućuje zauzimanje broja uzastopnih memorijskih lokacija koji je naveden kao njen operand

```
WORD    broj
```

Primer prevođenja

- Segment programa za računanje najvećeg zajedničkog delioca dva prirodna broja, napisan programskim jezikom *C*:

```
a = 12;
b = 8;
while (a != b)
  if (a > b)
    a = a - b;
  else
    b = b - a;
```

- Prethodni segment programa treba prevesti u segment programa koji je napisan hipotetskim asemblerskim jezikom

Primer prevođenja

- Korespondencija iskaza programskog jezika *C* i naredbi hipotetskog asemblerskog jezika (podrazumeva se da neoznačenoj promenljivoj **a** odgovara memorijska lokacija sa labelom **a**, a neoznačenoj promenljivoj **b** odgovara memorijska lokacija sa labelom **b**)
  - Iskazi pridruživanja

```
a = 12;
b = 8;
a = a - b;
b = b – a;
```

```
MOV    $12,a
MOV    $8,b
SUBU   a,b,a
SUBU   b,a,b
```

Primer prevođenja

- **while** iskaz

```
while (a != b)
  while_telo
```

```
while0:
        CMPU    a,b
        JEQ     false0
true0:
        while_telo
        JMP while0
false0:
```

- **if** iskaz

```
if (a > b)
  then_telo
else
  else_telo
```

```
if1:
        CMPU    a,b
        JLEU    false1
true1:
        then_telo
        JMP exit1
false1:
        else_telo
exit1:
```

Primer prevođenja

```
a = 12;
b = 8;
while (a != b)
  if (a > b)
    a = a - b;
  else
    b = b - a;
```

```
MOV   $12,a
...
```

Primer prevođenja

```
a = 12;
b = 8;
while (a != b)
  if (a > b)
    a = a - b;
  else
    b = b - a;
```

```
MOV    $12,a
MOV    $8,b
...
```

Primer prevođenja

```
a = 12;
b = 8;
while (a != b)
  if (a > b)
    a = a - b;
  else
    b = b - a;
```

```
                MOV     $12,a
                MOV     $8,b
while0:
                CMPU    a,b
                JEQ     false0
true0:
                ...
                JMP     while0
false0:
```

Primer prevođenja

```
a = 12;
b = 8;
while (a != b)
  if (a > b)
    a = a - b;
  else
    b = b - a;
```

```
                MOV     $12,a
                MOV     $8,b
while0:
                CMPU    a,b
                JEQ     false0
true0:
if1:
                CMPU    a,b
                JLEU    false1
true1:
                ...
                JMP     exit1
false1:
                ...
exit1:
                JMP     while0
false0:
```

Primer prevođenja

```
a = 12;
b = 8;
while (a != b)
  if (a > b)
    a = a - b;
  else
    b = b - a;
```

```
                MOV     $12,a
                MOV     $8,b
while0:
                CMPU    a,b
                JEQ     false0
true0:
if1:
                CMPU    a,b
                JLEU    false1
true1:
                SUBU    a,b,a
                JMP     exit1
false1:
                ...
exit1:
                JMP     while0
false0:
```

Primer prevođenja

```
a = 12;
b = 8;
while (a != b)
  if (a > b)
    a = a - b;
  else
    b = b - a;
```

```
                MOV     $12,a
                MOV     $8,b
while0:
                CMPU    a,b
                JEQ     false0
true0:
if1:
                CMPU    a,b
                JLEU    false1
true1:
                SUBU    a,b,a
                JMP     exit1
false1:
                SUBU    b,a,b
exit1:
                JMP     while0
false0:
```

Analiza prevedenog programa

- suvišna druga naredba **`CMPU a,b`**
- naredbu **`JMP exit1`** treba zameniti naredbom **`JMP while0`** da bi se izbegla dva uzastopna skoka:

```
                MOV     $12,a
                MOV     $8,b
while0:
                CMPU    a,b
                JEQ     false0
true0:
if1:
                JLEU    false1
true1:
                SUBU    a,b,a
                JMP     while0
false1:
                SUBU    b,a,b
exit1:
                JMP     while0
false0:
```

- do neefikasnosti je došlo zbog parcijalnog pristupa prilikom zamene iskaza izvornog jezika iskazima ciljnog jezika i nesagledavanja šireg konteksta u kome se zamena obavlja

Osvrt na prevođenje

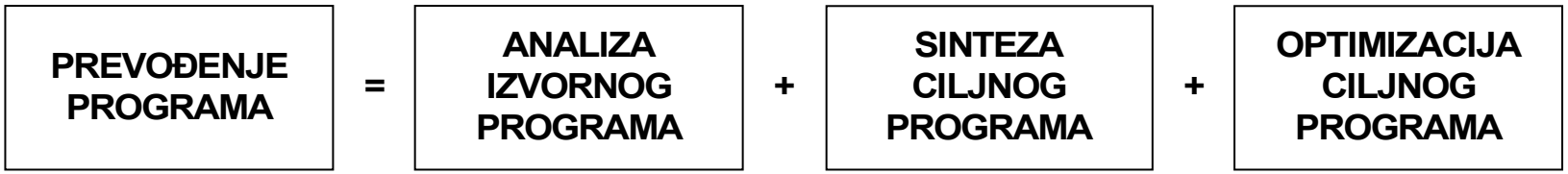
- Prepoznavanje iskaza izvornog programskog jezika – **analiza**
  - prepoznavanje reči ili **simbola** (*symbol*): **leksička analiza** (*lexical analisys*)

SIMBOLI												
a	=	12	;	b	8	while	(	!=	)	if	>	else

  - otkrivanje pogrešnih simbola: **leksičke greške** (samo ispravan izvorni program može biti preveden)
  - prepoznavanje rečenica ili iskaza: **sintaksna analiza** *(syntax analisys)*
  - otkrivanje (formalno) pogrešnih iskaza: **sintaksne greške** (primer: izostavljanje otvorene male zagrade iza `if`)
  - prepoznavanje značenja iskaza: **semantička analiza** *(semantic analisys)*
  - otkrivanje semantički pogrešnih iskaza: **semantičke greške** (primer: korišćenje nedefinisane promenljive)

Osvrt na prevođenje

- Generisanje iskaza ciljnog jezika – **sinteza**
- **Optimizacija izgenerisanog programa** (izraz optimizacija se koristi u smislu poboljšanja programa, a ne u smislu pravljenja optimalnog programa)



- Program prevodilac: **KOMPAJLER**
- Deo kompajlera zadužen za leksičku analizu: **SKENER**
- Deo kompajlera zadužen za sintaksnu analizu: **PARSER**
- Deo kompajlera zadužen za sintezu: **KOD GENERATOR**

Leksička i sintaksna analiza teksta

- Sintaksna analiza se oslanja na leksičku analizu
- Leksička i sintaksna analiza omogućuju prepoznavanje iskaza programskog ali i govornog jezika, pa se mogu primeniti ne samo na programski nego i na obični tekst
- Na primer, leksička i sintaksna analiza običnog teksta su potrebne da bi se odredio
  - ukupan broj reči u tekstu
  - ukupan broj rečenica u tekstu

Sintaksa jezika

- Za leksičku i sintaksnu analizu (programskog ili običnog) teksta potrebno je poznavati **sintaksu**, odnosno gramatiku (programskog ili govornog) jezika koja određuje pravila pisanja njegovih iskaza

Neformalna gramatika

- Gramatika može biti zadana neformalno:
  - reč je niz slova
  - rečenica je niz reči iza kojih dolazi tačka
  - tekst je niz rečenica
- Neformalna gramatika je neprecizna

Formalna gramatika

- Gramatika (programskog) jezika se izražava na formalan način u ***BNF* obliku**: Bakus-Naurova forma (*Backus-Naur form*)
- Ovako izražena gramatika se sastoji od **pravila** (*production*) koja određuju dozvoljene načineređanja (sintaksnih) **pojmova** (*nonterminal symbol*) i **simbola** (*terminal symbol*):

pojam → ***pojmovi*** **i/ili simboli**

- Leva strana pravila (pre znaka →) sadrži pojam koji može biti zamenjen sekvencom pojmova i/ili simbola koje sadrži desna strana pravila (iza znaka →). Jedan od pojmova predstavlja **polazni pojam**.

Formalna gramatika za običan tekst

text
→ ε
→ *text sentence*
sentence
→ capital_word *words* dot
words
→ ε
→ *words* word
→ *words* capital_word
word
→ small_letter
→ word small_letter
capital_word
→ capital_letter
→ capital_word small_letter
dot
→ “.”

Formalna gramatika za običan tekst

```
capital_letter
    → "A"
    → "B"
    → "C"
      ...
    → "Z"
small_letter
    → "a"
    → "b"
    → "c"
      ...
    → "z"
```

- Napomene
  - Pojmovi su napisani *italikom*
  - ε označava praznu desnu stranu pravila
  - Razmak i kraj linije imaju funkciju separatora simbola

Formalna gramatika

- **Proširena Bakus-Naurova forma** (*Extended Bacus-Naur Form*) uvodi sledeće elemente
  - na desnoj strani pravila mogu da se koriste:
    - [...] da označe da se sadržaj zagrada može pojaviti nijednom ili jednom
    - { ... } da označe da se sadržaj zagrada može pojaviti nijednom, jednom ili višestruko
    - (...) da označe grupisanje
    - | da označe alternative

Formalna gramatika za običan tekst

text
    → { *sentence* }
sentence
    → capital_word *words* dot
words
    → { (word | capital_word) }

word
   → small_letter { small_letter }
capital_word
   → capital_letter { small_letter }
dot
   → “.”

Formalna gramatika za običan tekst

capital_letter

→ **“A”** | **“B”** | **“C”** | **“D”** | **“E”** | **“F”** | **“G”** | **“H”**
| **“I”** | **“J”** | **“K”** | **“L”** | **“M”** | **“N”** | **“O”** | **“P”**
| **“Q”** | **“R”** | **“S”** | **“T”** | **“U”** | **“V”** | **“W”** | **“X”**
| **“Y”** | **“Z”**

small_letter

→ **“a”** | **“b”** | **“c”** | **“d”** | **“e”** | **“f”** | **“g”** | **“h”**
| **“i”** | **“j”** | **“k”** | **“l”** | **“m”** | **“n”** | **“o”** | **“p”**
| **“q”** | **“r”** | **“s”** | **“t”** | **“u”** | **“v”** | **“w”** | **“x”**
| **“y”** | **“z”**

Primer primene formalne gramatike

- Gramatika svakog (programskog) jezika obuhvata: simbole, pojmove, pravila i polazni pojam
- Gramatika (programskog) jezika može biti primenjena za proveru ispravnosti (programskog) teksta

Primer primene formalne gramatike

- Primer primene tekst gramatike za dokazivanje da je iskaz

  Ovo je tekst.

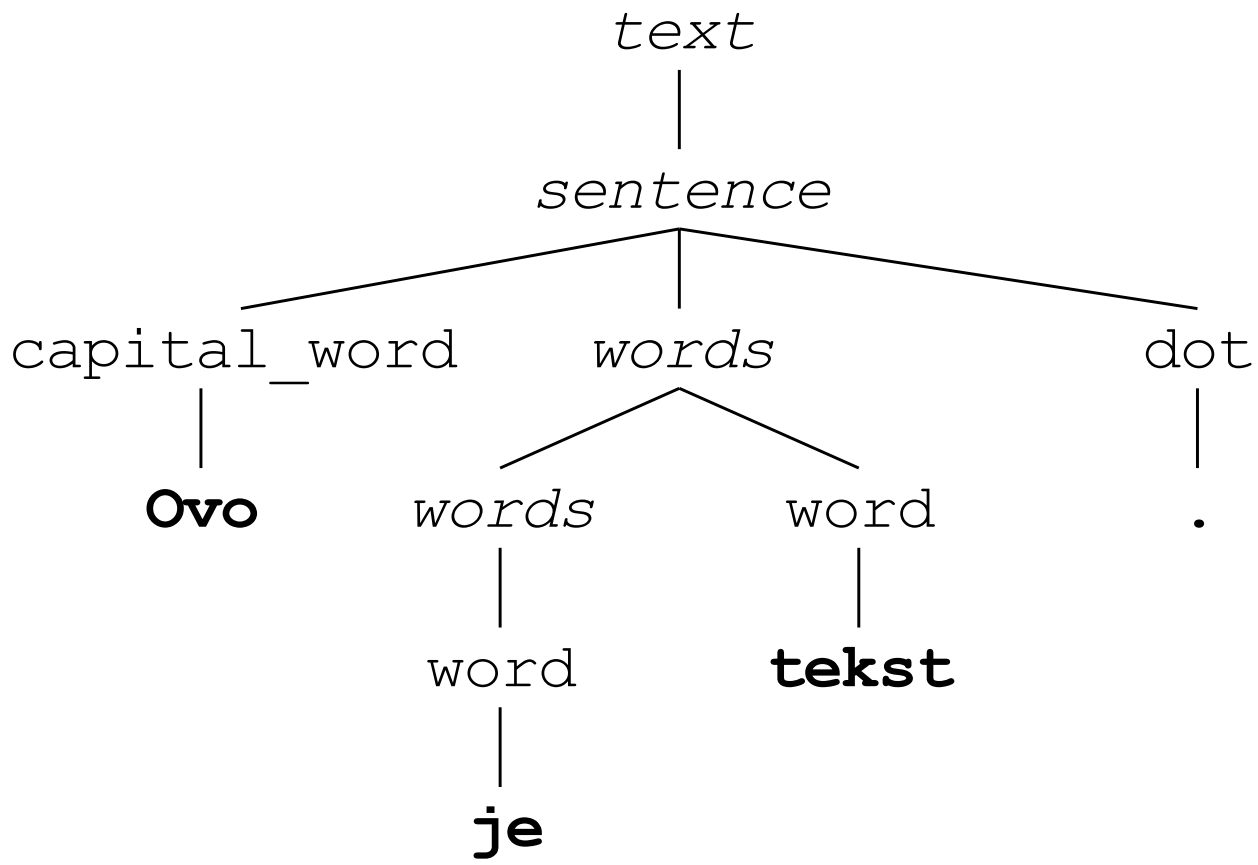
  ispravan tekst
  - izvođenje *(derivation)*

    text ⇒* *sentence* ⇒
    capital_word *words* dot ⇒
    Ovo *words* dot ⇒
    Ovo *words* word dot ⇒*
    Ovo word word dot ⇒
    Ovo je word dot ⇒
    Ovo je tekst dot ⇒
    Ovo je tekst .
  - znak ⇒ označava izvođenje u jednom koraku, a znak ⇒* označava izvođenje u više koraka

    text ⇒* **Ovo je tekst .**

Primer primene formalne gramatike

- prikaz izvođenja u obliku drveta



Skener

- Skener je zadužen za leksičku analizu
- Skener preuzima (skenira) znak po znak (programskog) teksta radi prepoznavanja simbola sastavljenih od zadanih znakova ili njihovih sekvenci. Pri tome ignoriše znakove koji razdvajaju simbole (delimiteri ili separatori) i reaguje na nedozvoljene znakove ili njihove sekvence.
- Skener je obično potprogram parsera i kada ga parser pozove on prepoznaje jedan simbol i isporučuje ga parseru (radi sintaksne analize)

Skener

- Za parser je zgodnije da mu skener umesto simbola (niza znakova) isporuči numeričku oznaku vrste simbola ili **token** (*token*)
- Za reči (na primer, `capital_word`) je potrebno da parser dobije uz token i pokazivač stringa reči jer string predstavlja vrednost simbola
- Za tačku (“.”) je potrebno da parser dobije samo token
- Znači, u opštem slučaju skener isporučuje parseru dve numeričke vrednosti – token i vrednost simbola

Skener – simboli tekst gramatike

SIMBOL	TOKEN	VREDNOST
“.”	_DOT	-
capital_letter {small_letter}	_CAPITAL_WORD	adresa stringa
small_letter {small_letter}	_WORD	adresa stringa

Skener – tokeni tekst gramatike

```
#define  _DOT            1
#define  _CAPITAL_WORD   2
#define  _WORD           3
```

Skener – dijagram prelaza

- Potreban sistematičan način prepoznavanja simbola
- Generalni – opšti skener (preuzme opis simbola i na osnovu njega prepoznaje i klasifikuje simbole)
- Prepoznavanje simbola je zasnovano na **gramatici simbola** (*patterns*): `word`, `capital_word`, `dot` (formalna gramatika za običan tekst)
- Ona se može prikazati u grafičkom obliku pomoću **dijagrama prelaza** *(transition diagram)* koji daje osnovu za sistematičan način prepoznavanja simbola

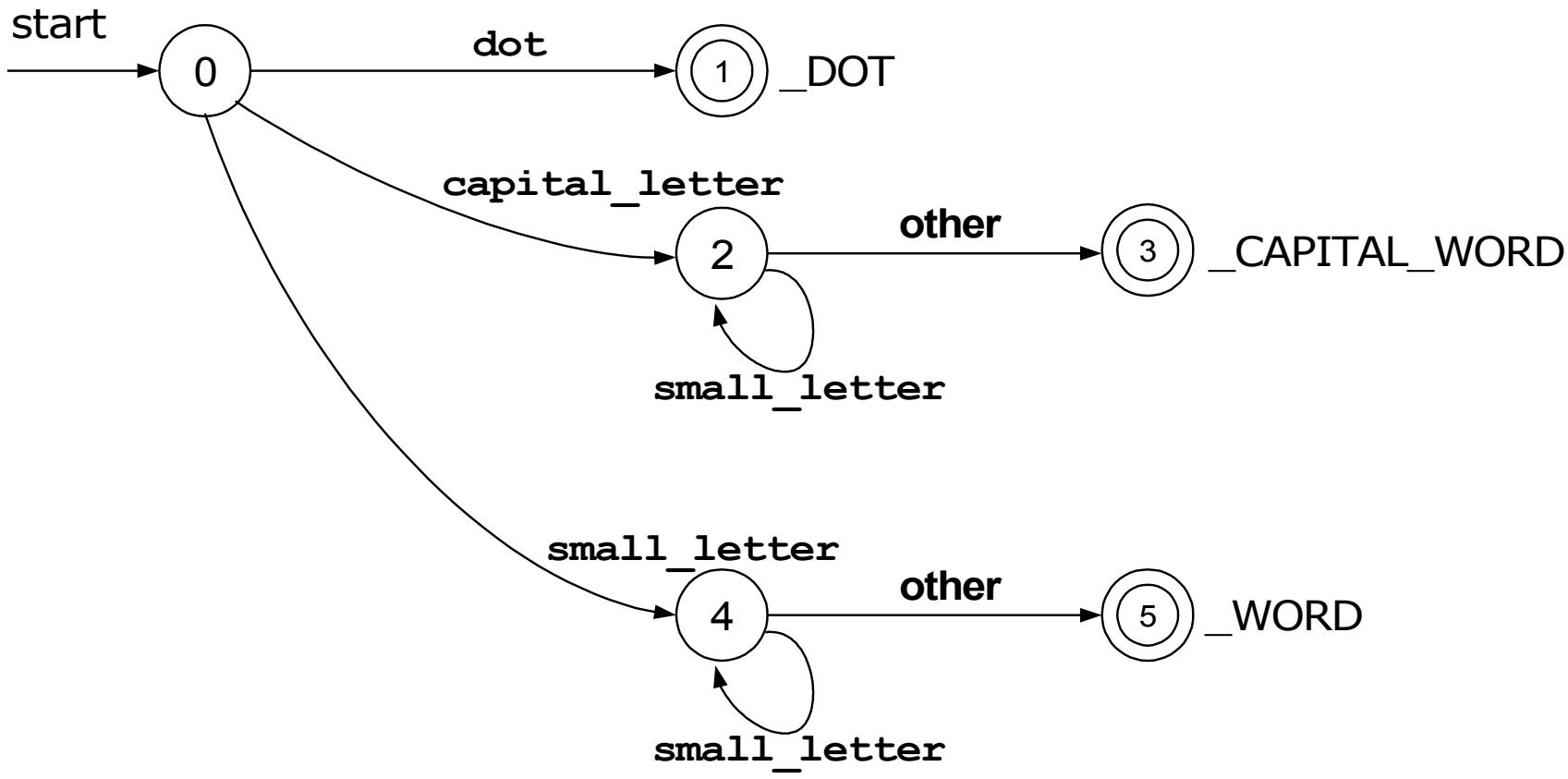
Skener – dijagram prelaza

- Za formiranje dijagrama prelaza potrebno je uvesti stanja skenera
  - polazno stanje
  - po jedno završno stanje za svaki prepoznati simbol
  - za simbole sastavljene od jednog znaka, pojava tog znaka prevodi skener iz polaznog u završno stanje
  - za simbole sastavljene od više znakova, pojava prvog znaka prevodi skener iz polaznog u prvo međustanje, a pojava svakog narednog znaka prevodi skener u sledeće međustanje, kada ima više međustanja. Ako pravilo formiranja simbola dozvoljava ponavljanje pojedinih znakova, tada nakon njihove pojave skener ostaje u zatečenom međustanju.

Skener – dijagram prelaza

- Dijagram prelaza je usmereni graf u čiji sastav ulaze:
  - čvorovi koji predstavljaju stanja skenera – jedan od ovih čvorova je polazni, a više njih mogu biti završni čvorovi
  - usmerene spojnice između čvorova koje ukazuju na moguće prelaske iz jednog u drugo stanje
  - labele usmerenih spojnica. Svaka labela odgovara skupu znakova. Podrazumeva se da spojnice koje kreću iz istog čvora imaju različite labele, odnosno da je presek njihovih labela prazan skup (ovo ograničenje obezbeđuje jednoznačnost prelazaka).
  - znakovi od kojih se mogu obrazovati labele

Skener – dijagram prelaza



Skener – dijagram prelaza

- Podrazumeva se da početno stanje skenera odgovara početnom čvoru
- Kada se ne nalazi u završnom stanju, skener preuzima znak i proverava da li on pripada skupu znakova neke od labela spojnica izlazećih iz čvora koji odgovara trenutnom stanju skenera. Ako preuzeti znak pripada skupu znakova jedne od labela pomenutih spojnica, skener prelazi u stanje određeno čvorom koga ukazuje dotična spojnica. Ako ne pripada, tada je otkrivena leksička greška.

Skener – dijagram prelaza

- Kada dospe u stanje koje odgovara završnom čvoru, skener je prepoznao simbol i može da ponovo pređe u početno stanje. U nekim od završnih stanja skener mora da **vrati poslednje preuzeti znak**, ako on ne pripada prepoznatom simbolu, jer tada on pripada sledećem simbolu. Na primer, ako iza znaka < ne sledi znak =, reč je o relacionom operatoru manje, pa znak iza znaka < ne pripada tom nego sledećem simbolu.
- Skener – konačan automat (unapred zadan konačan broj stanja i prelaza između njih)

Skener – tabela prelaza

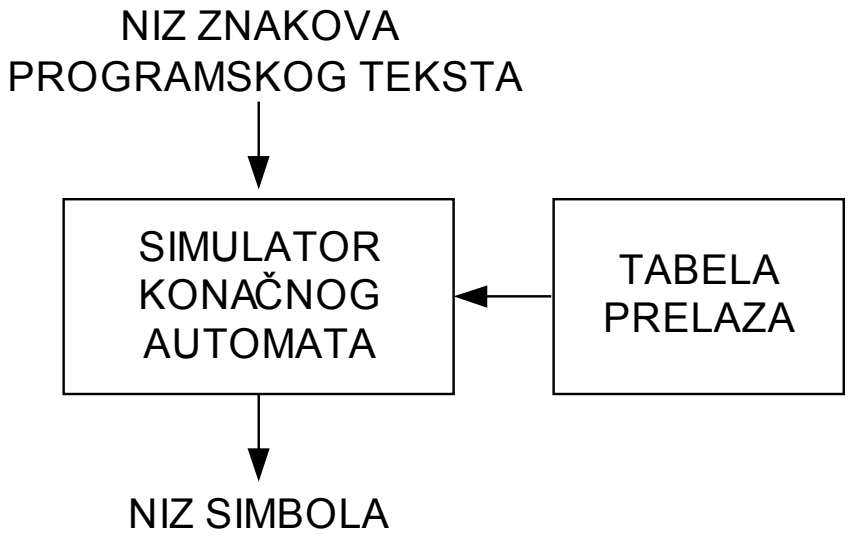
- Dijagram prelaza može biti prikazan i u obliku **tabele prelaza**
- Tabela prelaza opisuje akciju skenera na pojavu nekog znaka u zadanom stanju. Redove tabele prelaza označavaju redni brojevi polaznog stanja i međustanja, a njene kolone označavaju znakovi
- Svaki element tabele prelaza sadrži ili redni broj stanja u koje skener prelazi nakon pojave odgovarajućeg znaka ili crticu kao oznaku da u datom stanju nije predviđena pojava odgovarajućeg znaka

Skener – tabela prelaza

STANJE	ZNAK		
	.	`capital_letter`	`small_letter`
0	1	2	4
2	3	3	2
4	5	5	4

Skener - deterministički konačni automat

- Dijagram/tabela prelaza definiše **deterministički konačni automat** (*deterministic finite automata*) koji određuje ponašanje skenera
- Skener može imati oblik funkcije koja simulira takav automat. Da bi ovakav skener prepoznao simbole konkretne gramatike simbola on mora imati na raspolaganju odgovarajuću tabelu prelaza:



Skener – regularni izrazi

- Tabelu prelaza u potpunosti definiše odgovarajući opis gramatike simbola. Za ovu svrhu se koriste **regularne gramatike**. Njihova pravila imaju oblik **regularnih definicija** koje sadrže ime za određenu vrstu simbola i njen opis u formi **regularnog izraza** (*regular expression*):

  ```
  ime -> regularni izraz
  ```

- Oblikovanje regularnih izraza
  - pojedinačni znakovi predstavljaju regularne izraze
    - regularnom izrazu `a` odgovara string “a”
  - spajanjem pojedinačnih znakova nastaju novi regularni izrazi
    - regularnom izrazu `abc` odgovara string “abc”

Skener – regularni izrazi

- znak `*` označava da se njemu prethodeći regularni izraz navodi nijednom ili više puta
  - regularnom izrazu `abc*` odgovaraju stringovi “ab”, “abc”, “abcc”, “abccc”, ...
- znak `+` označava da se njemu prethodeći regularni izraz navodi jednom ili više puta
  - regularnom izrazu `abc+` odgovaraju stringovi “abc”, “abcc”, “abccc”, ...
- znak `?` označava da se njemu prethodeći regularni izraz navodi nijednom ili jednom
  - regularnom izrazu `abc?` odgovaraju stringovi “ab” i “abc”

Skener – regularni izrazi

- znak `|` označava alternative
  - regularnom izrazu `ab|cd` odgovaraju stringovi “ab” i “cd”
- male zagrade omogućuju grupisanje
  - regularnom izrazu `a(b|c)d` odgovaraju stringovi “abd” i “acd”
- uglaste zagrade omogućuju navođenje klase znakova
  - regularni izraz `[0123456789]` odgovara regularnom izrazu `(0|1|2|3|4|5|6|7|8|9)`, znači svakoj od pojedinačnih cifara
- znak `-` omogućuje skraćeno navođenje klase znakova
  - regularni izraz `[0-9]` odgovara regularnom izrazu `[0123456789]`

Skener – regularni izrazi

- znak `.` označava bilo koji znak osim znaka za novi red
  - regularnom izrazu `.*` odgovaraju svi mogući stringovi koji mogu da se pojave u jednom redu, uključujući i prazan red
- znak `\` omogućuje da se posebni znakovi tretiraju kao obični
  - regularnom izrazu `a\.b` odgovara string “a.b”
- vitičaste zagrade omogućuju definisanje broja ponavljanja
  - regularni izraz `a{2,4}` opisuje stringove “aa”, “aaa” i “aaaa”

- Primeri regularnih izraza
  - regularni izraz `[0-9]+` opisuje prirodne brojeve i nulu
  - regularni izraz `-?[0-9]+` opisuje cele brojeve

Skener – generator skenera

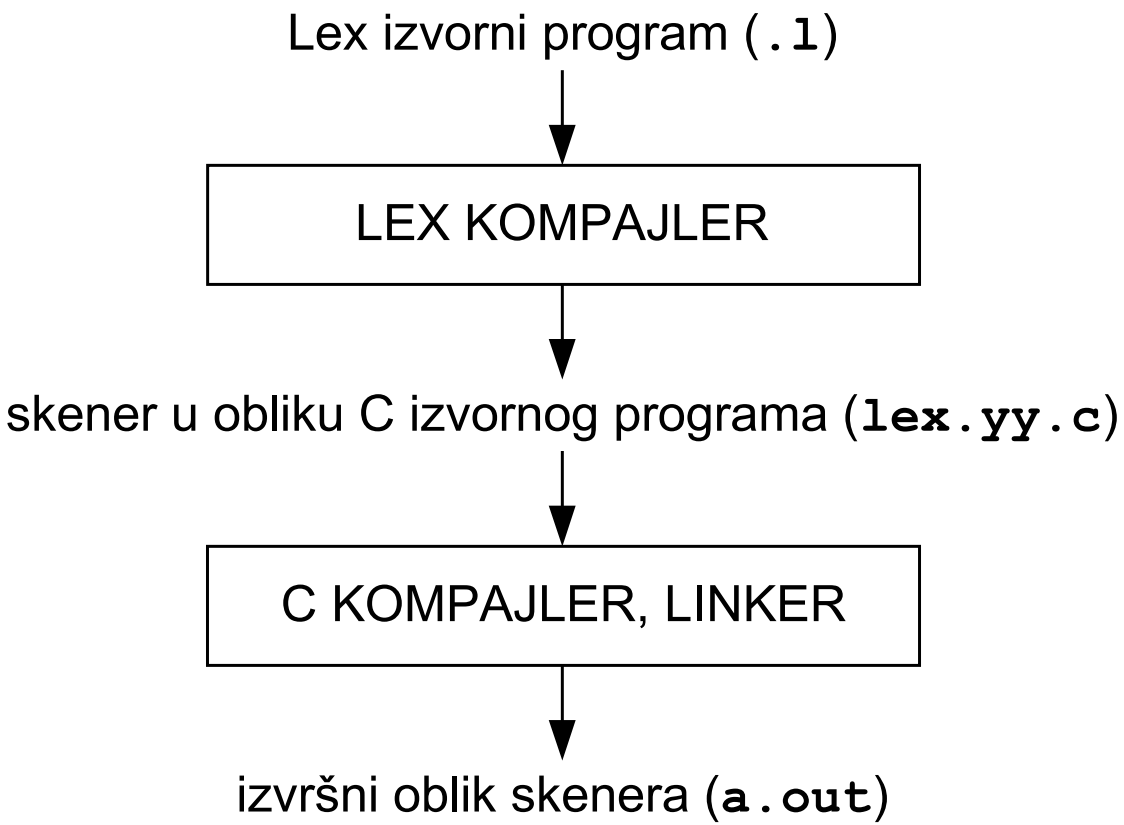
- Opis simbola u formi regularnog izraza je dovoljan za automatsko generisanje tabele prelaza namenjene za skener koji ima oblik simulatora konačnog automata, pa može da prepoznaje opisane simbole. Generator tabele prelaza određuje ponašanje ovakvog skenera, pa se naziva i **generator skenera**.
- Domet generisanog skenera je prepoznavanje simbola. Međutim, nakon dolaska skenera u završno stanje, treba da usledi njegova akcija nakon prepoznavanja simbola. Nju zna i opisuje korisnik u obliku segmenta programa. Takvi korisnički opisi akcija skenera moraju biti saopšteni generatoru skenera, da bi izgenerisani skener mogao na željeni način da reaguje na prepoznavanje simbola. Opisi takvih akcija se uparuju sa opisom odgovarajućih simbola u obliku:

```
regularni izraz      opis akcije
```

Skener – generator skenera

- Pošto su opisi akcija segmenti programa koji koriste konstante, promenljive i funkcije, pored opisa akcija, generatoru skenera moraju biti saopštene odgovarajuće definicije konstanti, promenljivih i funkcija, korišćenih u opisu akcija
- Radi preglednosti regularnih izraza u njima se koriste imena navedena u regularnim definicijama, koje prethode regularnim izrazima
- Primer generatora skenera je **Lex kompajler** (*Lex compiler*)

Skener - Lex


Lex izvorni program (.l)
LEX KOMPAJLER
skener u obliku C izvornog programa (lex.yy.c)
C KOMPAJLER, LINKER
izvršni oblik skenera (a.out)

Skener - Lex

- Izgled Lex izvornog programa
  (/* i */ označavaju početak i kraj komentara)

```
/* deklaracije */
%{
   /* deklaracije i definicije (konstante, promenljive) */
%}
/* regularne definicije u obliku:
   ime_regularnog_izraza      regularni_izraz */
%%
/* pravila prelaza u obliku:
   regularni_izraz            { segment C programa
                                koji opisuje akciju } */
%%
/* definicije pomoćnih C funkcija */
```

Skener - Lex

- Lex kompajler generiše skener u obliku *C* funkcije **`yylex`**:

```
int yylex(void);
```

- Lex kompajler oblikuje funkciju **`yylex`** na osnovu regularnih definicija, regularnih izraza i njima pridruženih segmenata *C* programa koji opisuju željene akcije
- Funkciju **`yylex`** Lex kompajler smešta u izlaznu datoteku **`lex.yy.c`** zajedno sa neizmenjenim definicijama *C* konstanti, *C* promenljivih i pomoćnih *C* funkcija
- Imena regularnih izraza se mogu navoditi u drugim regularnim izrazima samo između velikih zagrada
- Za Lex kompajler znakovi **<** i **>** imaju posebno značenje (označavaju stanja skenera), pa se moraju koristiti između navodnika (“<” i “>”) kada se koriste kao obični znakovi
- Lex kompajler zahteva definiciju funkcije **`yywrap()`** koja opisuje ponašanje skenera kada naiđe na EOF znak

Skener - Lex

- Nakon poziva funkcije **`yylex`**, u toku njenog izvršavanja, ponavlja se prepoznavanje simbola i izvršavanje zadanih akcija sve dok se u okviru zadanih akcija ne izvrši **`return`** iskaz. Tek tada sledi povratak iz ove funkcije. Automatski povratak iz ove funkcije se dešava kada se u toku preuzimanja znakova naiđe na kraj datoteke, a u tom slučaju povratna vrednost funkcije je **`0`**.

Skener - Lex

- Uz **`yylex`** funkciju Lex kompajler definiše i globalne promenljive:

`char yytext[] (char * ytext)`	omogućuje pristup stringu poslednje prepoznatog simbola
`int yyleng`	sadrži dužinu stringa poslednje prepoznatog simbola

- Ove promenljive omogućuju preuzimanje znakova iz stringa poslednje prepoznatog simbola.
- Znakove iz stringa simbola treba preuzeti odmah po prepoznavanju, da ne bi bili izgubljeni!

Skener - Lex

- Podrazumeva se da se token prepoznatog simbola vraća kao vrednost funkcije **`yylex`**, a da se vrednost prepoznatog simbola vraća posredstvom globalne promenljive **`yylval`** koja je posebno definisana:

```
int yylval
```

- Funkcija **`yylex`** pronalazi 1) najduži string koji odgovara nekom regularnom izrazu i 2) pripisuje ga prvonađenom regularnom izrazu. Ako pronađeni string odgovara nekolicini regularnih izraza, prvo pravilo omogućuje, na primer, da se **`<=`** prepozna kao manje ili jednako, a ne kao manje. U istom slučaju drugo pravilo omogućuje, na primer, da se rezervisana reč **`if`** odmah prepozna kao rezervisana reč, a ne kao identifikator, jasno pod uslovom da regularni izraz **`if`** prethodi regularnom izrazu **`[A-Za-z]([A-Za-z]|[0-9])*`**

Skener – primer izlaza skenera

```
Ovo je tekst.
Za skeniranje.

_____________________________________________________________

START

                Ovo   TOKEN:   _CAPITAL_WORD    value:   Ovo
                 je   TOKEN:   _WORD            value:   je
              tekst   TOKEN:   _WORD            value:   tekst
                  .   TOKEN:   _DOT
                 Za   TOKEN:   _CAPITAL_WORD    value:   Za
         skeniranje   TOKEN:   _WORD            value:   skeniranje
                  .   TOKEN:   _DOT

STOP
```

- Lex specifikacija (`examples/text/scanner/scanner.l`)

Parser

- Zadatak parsera je da proveri da li je ulazni niz simbola (tokena), dobijen od skenera, u skladu sa gramatikom
- Ovakva provera se svodi ili na (1) pokušaj da se iz polaznog pojma gramatike po njenim pravilima izvede niz simbola identičan ulaznom nizu simbola ili na (2) pokušaj da se ulazni niz simbola redukuje (sažme) po pravilima gramatike u njen polazni pojam
- Prvi pristup odgovara **silaznom** (*top-down*) **parsiranju**, a drugi pristup odgovara **uzlaznom** (*bottom-up*) **parsiranju**
- Kod silaznog parsiranja parser u toku izvođenja formira **niz izvođenja** (*derivation*) koji se može prikazati u obliku **drveta parsiranja** (*parse tree*)
- Izvođenja mogu biti (1) **s leva** (*leftmost*): u svakom koraku se zamenjuje krajnje levi pojam ili (2) **s desna** (*rightmost*): u svakom koraku se zamenjuje krajnje desni pojam

Parser

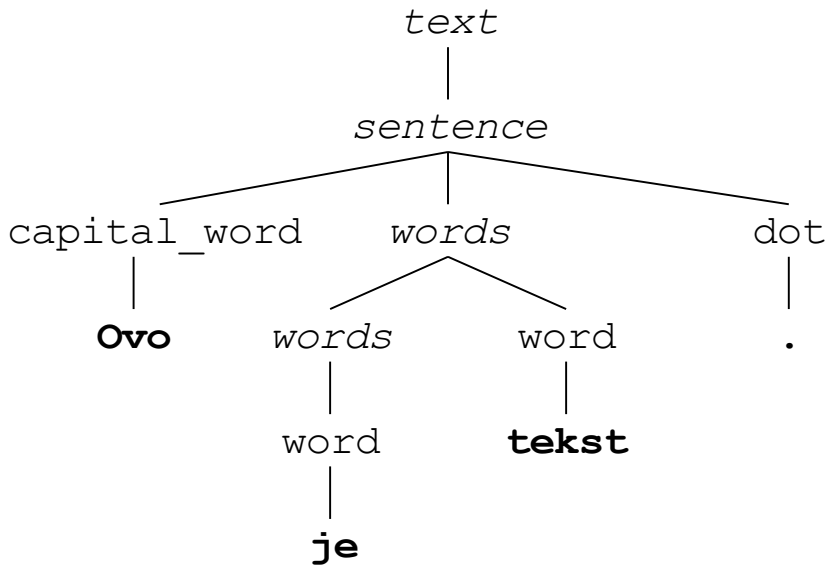
- Primeri nizova izvođenja i korespondentnog drveća parsiranja za iskaz

  Ovo je tekst.

- izvođenje s leva

text =>
text sentence =>
sentence =>
capital_word *words* dot =>
Ovo *words* dot =>
Ovo *words* word dot =>
Ovo *words* word word dot =>
Ovo word word dot =>
Ovo je word dot =>
Ovo je tekst dot
Ovo je tekst .

- drvo parsiranja za izvođenje s leva

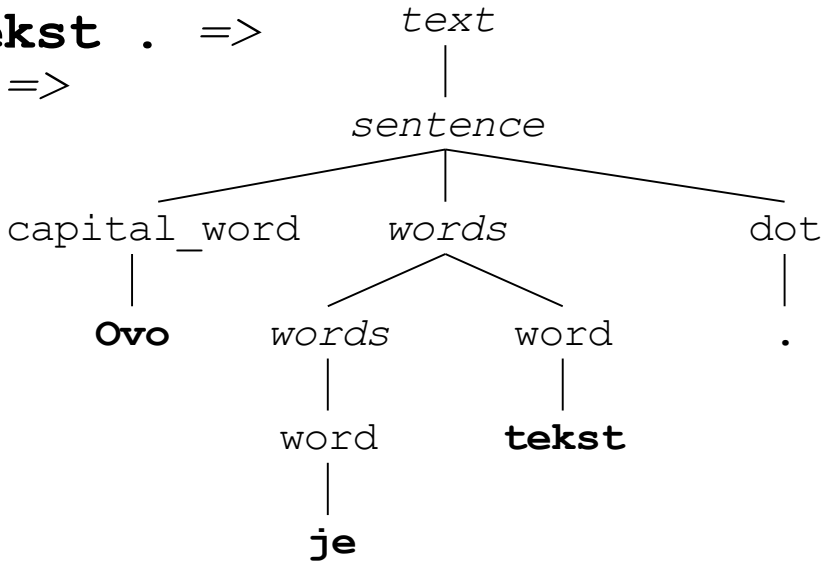


Parser

- izvođenje s desna

text =>
text sentence =>
text capital_word *words* dot =>
text capital_word *words* . =>
text capital_word *words* word . =>
text capital_word *words* **tekst** . =>
text capital_word *words* word **tekst** . =>
text capital_word *words* **je tekst** . =>
text capital_word **je tekst** . =>
text **Ovo je tekst** . =>
Ovo je tekst .

- drvo parsiranja za izvođenje s desna



Parser

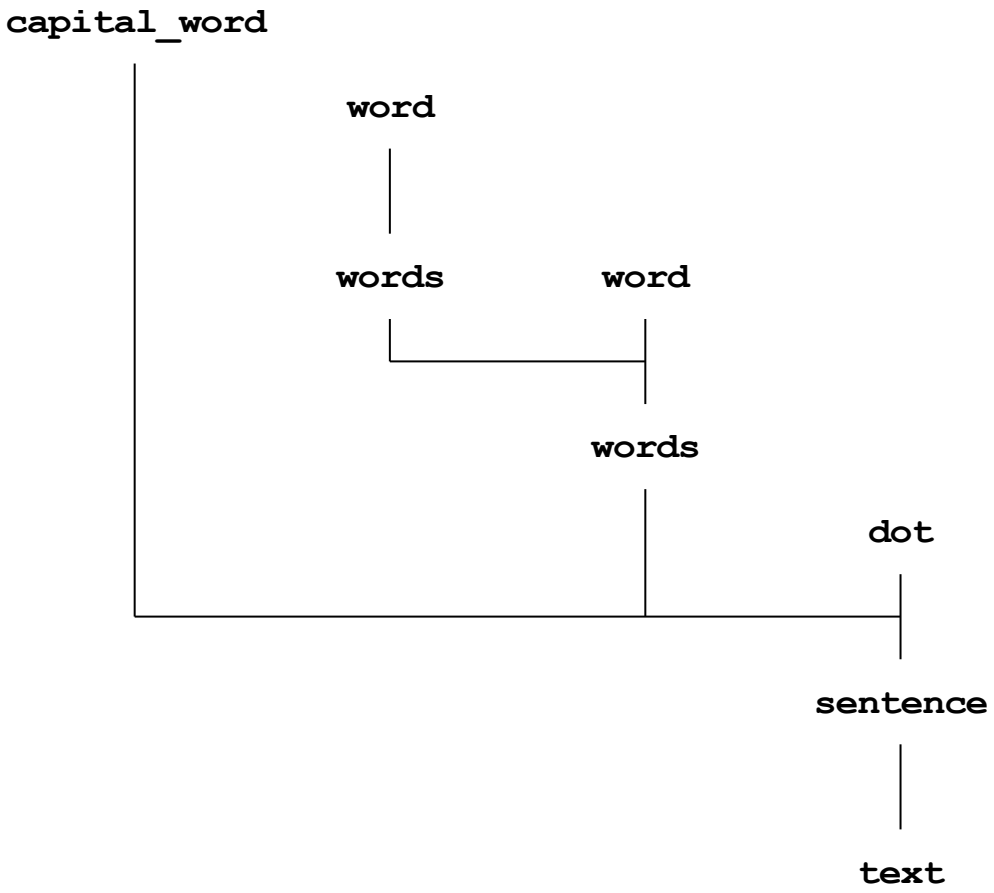
- Kod silaznog parsiranja parser na svakom koraku preuzima s leva jedan simbol iz ulaznog niza simbola i proverava da li gramatika predviđa pojavu preuzetog simbola u dotičnom koraku
- Na primer, parser može preuzeti kao prvi simbol `capital_word` (**Ovo**) i ustanoviti da pravila *`text`* i *`sentence`* omogućuju njegovu pojavu. Sada parser može kao drugi simbol preuzeti `word` (**je**) koji se uklapa u pravilo *`words`*. U isto pravilo uklapa se i treći simbol `word` (**tekst**). Dalje parser može preuzeti simbol `dot` (.), koji se uklapa u pravilo *`sentence`*, odnosno *`text`*.
- U toku silaznog parsiranja parser praktično konstruiše drvo parsiranja prikazano na prethodnim slajdovima
- Za silazno parsiranje je zgodno primeniti izvođenje s leva, jer se tada simboli sa ulaza preuzimaju u prirodnom redosledu

Parser

- Postupak redukcije kod uzlaznog parsiranja je inverzan (suprotan) postupku izvođenja kod silaznog parsiranja. U praksi je prihvaćen postupak redukcije koji je inverzan postupku izvođenja s desna (da bi se simboli iz ulaznog niza simbola mogli preuzimati u prirodnom redosledu)
- Kod uzlaznog parsiranja parser na svakom koraku s leva preuzima jedan simbol iz ulaznog niza simbola, dodaje ga na kraj prethodno formirane sekvence pojmova i/ili simbola i proverava da li je novoformirana sekvenca identična desnoj strani nekog pravila gramatike. Ako jeste, parser redukuje (zamenjuje) novoformiranu sekvencu pojmom sa leve strane pomenutog pravila.
- Na primer, parser može preuzeti kao prvi simbol `capital_word` (**`Ovo`**). Zatim on može kao drugi simbol preuzeti `word` (**`je`**) i redukovati ga pojmom *`words`*. Dalje on može kao treći simbol preuzeti `word` (**`tekst`**), a zatim redukovati sekvencu `words word` pojmom *`words`*. Na kraju on može kao četvrti simbol preuzeti `dot` (`.`) i sekvencu `capital_word` *`words`* `dot` redukovati pojmom *`sentence`*, a ovaj pojam, zatim redukovati pojmom *`text`*.

Parser

- I u toku uzlaznog parsiranja parser konstruiše (izvrnuto) drvo parsiranja



Parser

- Sekvenca pojmova i/ili simbola, koja je kandidat za redukciju, može da se formira na steku tako što se na stek smeštaju jedan za drugim njeni elementi. Tako se na vrhu steka uvek nalazi završni element dela sekvence koja može biti redukovana. U okviru redukcije redukovani deo sekvence se skida sa steka, a umesto njega se na stek smešta pojam u koga se pomenuti deo sekvence može redukovati

Parser

- Primer korišćenja steka (stek se puni s desna u levo)

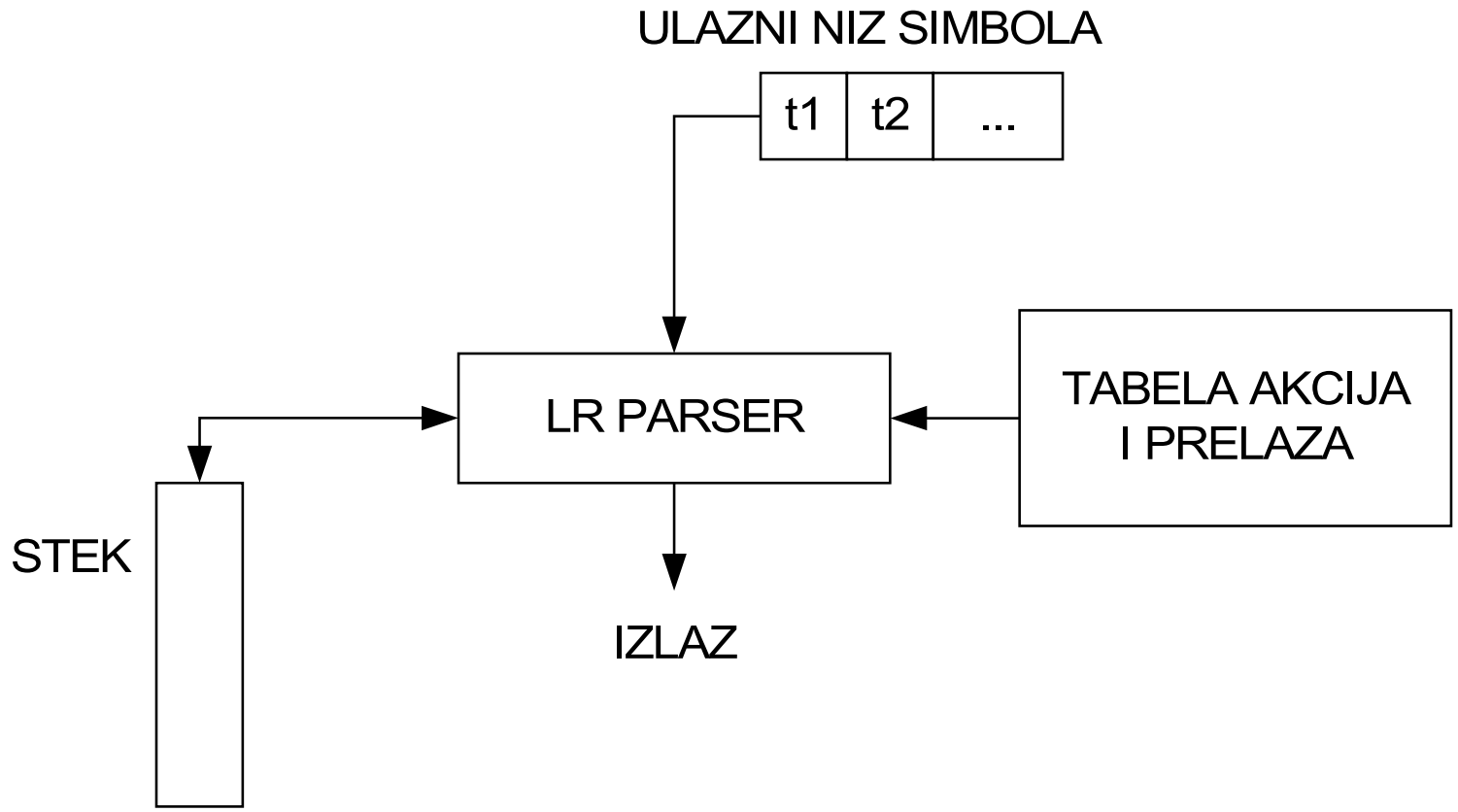
`capital_word`	sadržaj steka nakon smeštanja tokena `capital_word` (**Ovo**)
`capital_word word`	sadržaj steka nakon smeštanja tokena `word` (**je**)
`capital_word` *`words`*	sadržaj steka nakon redukcije
`capital_word` *`words`* `word`	sadržaj steka nakon smeštanja tokena `word` (**tekst**)
`capital_word` *`words`*	sadržaj steka nakon redukcije
`capital_word` *`words`* `dot`	sadržaj steka nakon smeštanja tokena `dot` (**.**)
`sentence`	sadržaj steka nakon redukcije
`text`	sadržaj steka nakon redukcije

Parser

- Praktičan značaj uzlaznog parsiranja proizlazi iz činjenice da taj postupak koriste LR parseri (***L****eft-to-right scanning of the input, constructing a* ***R****ightmost derivation in reverse*)
- LR parseri imaju svojstvo opštosti i implementiraju se kao konačni automati

Parser

- Model LR parsera



Parser – LR parser

- Na svakom koraku svoje aktivnosti LR parser se nalazi u jednom od stanja iz skupa mogućih stanja. Podrazumeva se da se LR parser nalazi u početnom stanju na početku svoje aktivnosti.
- Stanja su odabrana tako da svako stanje registruje napredak, ka nekoj od mogućih redukcija, koji je LR parser napravio na datom koraku. Znači postepeno napredovanje koje parser napravi ka nekoj redukciji se registruje u obliku niza stanja kroz koja on prolazi u toku pomenutog napredovanja.
- Svoja stanja LR parser čuva na steku umesto odgovarajućih simbola i/ili pojmova. Na vrhu steka se nalazi važeće stanje, ispod njega prethodno stanje, itd. za preostala stanja iz niza stanja kroz koja je LR parser prošao.

Parser – LR parser

- Tabela akcija i prelaza upravlja aktivnošću LR parsera. Broj njenih redova je određen brojem različitih stanja, a broj njenih kolona je određen brojem različitih simbola. U preseku reda stanja S_i i kolone simbola t_j nalazi se element tabele koji određuje akciju LR parsera kada on u stanju S_i na početku ulaznog niza simbola pronađe simbol t_j
- Svaki od elemenata tabele akcija i prelaza sadrži oznaku jedne od moguće četiri akcije LR parsera. Pod pretpostavkom da je LR parser u stanju S_i i da se na početku ulaznog niza simbola nalazi simbol t_j, moguće akcije LR parsera su:

Parser – LR parser

1. LR parser prelazi u stanje navedeno u elementu $[S_i,t_j]$, smešta oznaku tog stanja na stek i pomera (*shift*) simbol t_j sa početka ulaznog niza simbola
2. LR parser redukuje (*reduce*) niz od n stanja sa vrha steka novim stanjem i proizvede odgovarajući izlaz. Dužina n niza redukovanih stanja je određena sadržajem elementa $[S_i, t_j]$, a novo stanje LR parsera zavisi od stanja koje na steku prethodi redukovanom nizu stanja i od vrste redukcije. Simbol t_j ostaje na početku ulaznog niza simbola.
3. LR parser objavljuje uspešno prepoznavanje ulaznog niza simbola
4. LR parser objavljuje da je ulazni niz simbola pogrešan

- Zbog prve dve akcije, parsiranja koja obavljaju LR parseri se nazivaju *shift-reduce* parsiranja

Parser – LR parser

- LR parseri se međusobno razlikuju po tabeli akcija i prelaza. Ova tabela je zavisna od gramatike. Ona se oblikuje tako da se za polazno stanje LR parsera ustanovi koji simboli su prihvatljivi u tom stanju. Za njih se odrede nova stanja u koja LR parser prelazi iz početnog stanja pri pojavi pomenutih simbola. Za ostale simbole se konstatuje da njihova pojava u polaznom stanju predstavlja grešku. Za svako od novih stanja se ponavlja prethodni postupak, uz napomenu da pojava određenih simbola u pomenutim stanjima može da dovede i do redukcije ili do uspešnog prepoznavanja ulaznog niza simbola.
- LR parseri se mogu automatski izgenerisati ako se na osnovu zadate gramatike automatski proizvede tabela akcija i prelaza i tako definiše ponašanje generisanog LR parsera. To je zadatak **generatora LR parsera**.

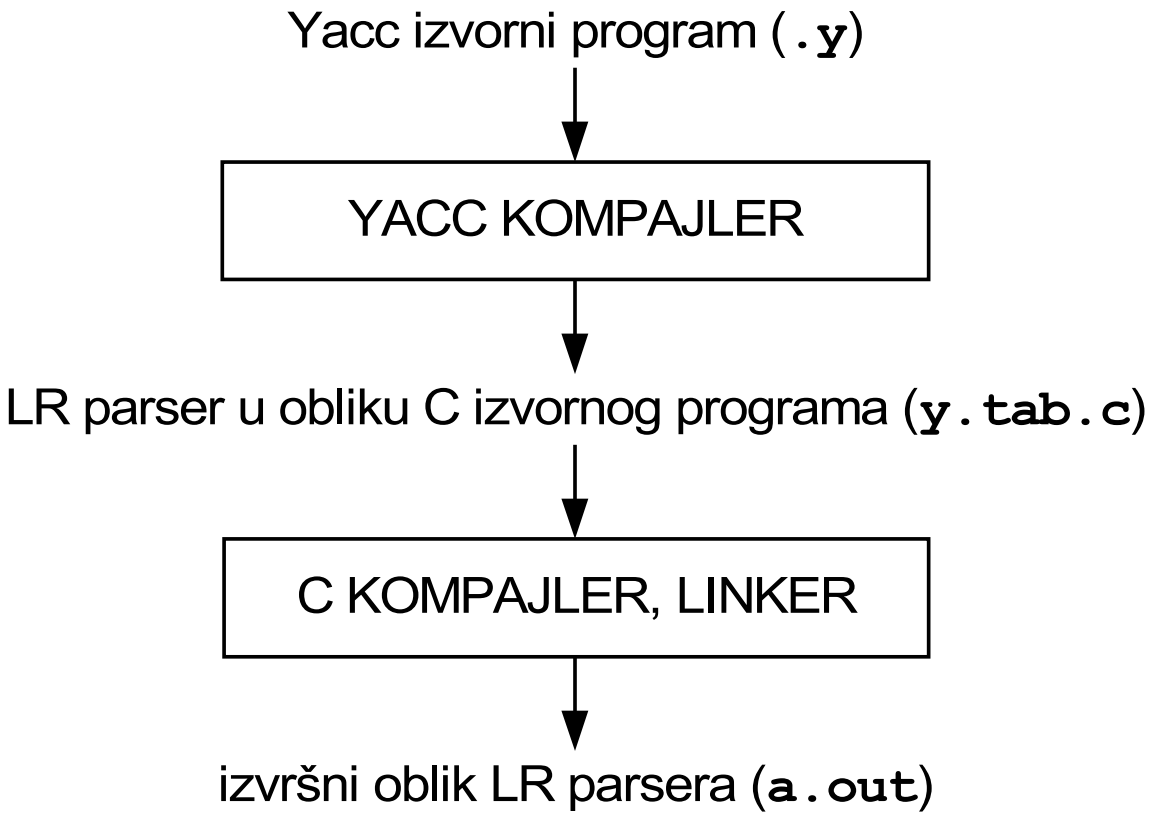
Parser – generator LR parsera

- Domet generisanog LR parsera je prepoznavanje ispravnih ili pogrešnih nizova simbola. Akciju parsera nakon prepoznavanja ispravnih ili pogrešnih nizova simbola, odnosno njegov izlaz u toj situaciji treba da definiše korisnik. On to može da uradi tako što će generatoru LR parsera uz pravila navoditi i segmente programa koji određuju akcije LR parsera nakon prepoznavanja ispravnih ili pogrešnih nizova simbola primenom pomenutih pravila:

```
pravilo          opis akcije
```

- Pošto su opisi akcija segmenti programa u kojima se koriste konstante, promenljive i funkcije, pored opisa akcija, generatoru LR parsera moraju biti saopštene odgovarajuće definicije konstanti, promenljivih i funkcija, korišćenih u opisu akcija
- U okviru pravila se navode tokeni, pa se i njihove definicije moraju saopštiti generatoru LR parsera
- Primer generatora LR parsera je **Yacc** *(**Y**et **A**nother **C**ompiler **C**ompiler*) **kompajler**

Parser - Yacc


Yacc izvorni program (.y)
YACC KOMPAJLER
LR parser u obliku C izvornog programa (y.tab.c)
C KOMPAJLER, LINKER
izvršni oblik LR parsera (a.out)

Parser - Yacc

- Izgled Yacc izvornog programa (specifikacije)

```
/* deklaracije */
%{ /* deklaracije i definicije (konstante, promenljive);
      globalni scope - vide se u akcijama) */
%}
/* definicije tokena */
/* prioriteti i redosled primena operatora */
%%
/* pravila u obliku:
   pravilo  { segment C programa
              koji opisuje akciju } */
%%
/* definicije pomoćnih C funkcija */
```

Parser - Yacc

- Yacc kompajler generiše LR parser u obliku *C* funkcije **`yyparse`**:

```
int yyparse(void);
```

- Povratna vrednost funkcije **`yyparse`** različita od nule ukazuje na grešku u parsiranju
- Tokeni se zadaju u obliku

```
%token   ime tokena
```

- Podrazumeva se da je prvo pravilo polazno (redukcijom po ovom pravilu parser završava rad)

Parser - Yacc

- Podrazumeva se da su pravila navedena u *BNF* notaciji
- Pravilo:

pojam -> alternativa$_1$ | alternativa$_2$ | ... | alternativa$_n$

Yacc prihvata u obliku:

```
pojam :  alternativa1      { /* C opis akcije */ }
      |  alternativa2      { /* C opis akcije */ }
         ...
      |  alternativan      { /* C opis akcije */ }
      ;
```

- Uputno je da se ponavljanja označavaju levom rekurzijom

Parser - Yacc

- Yacc podrazumeva da svaki pojam i simbol iz pravila poseduje vrednost. Vrednost simbola određuje skener, a vrednost pojmova određuje parser. Radi čuvanja vrednosti pojmova i simbola, čija stanja se nalaze na steku, Yacc predviđa poseban **stek za vrednosti**. Relativne pozicije stanja koja odgovaraju pojmovima/simbolima na **steku stanja** i relativne pozicije odgovarajućih vrednosti na steku vrednosti su identične. Radi rukovanja ovim vrednostima Yacc predviđa posebnu notaciju. Vrednost pojma s leve strane pravila Yacc označava sa **$$**, a vrednost pojmova i simbola sa desne strane pravila Yacc označava sa $\$_i$ (**i** je redni broj pojma ili simbola na desnoj strani pravila, posmatrano s leva u desno).
- $\$_i$ označava lokacije ispod vrha steka, a nakon njihove redukcije **$$** označava vrh steka

Parser - Yacc

- Za pravilo

sentence → _CAPITAL_WORD words _DOT

$$ označava vrednost pojma **sentence**, **$1** vrednost simbola **_CAPITAL_WORD**, **$2** vrednost pojma **words**, a **$3** vrednost simbola **_DOT**. Podrazumeva se da su vrednosti pojmova/simbola s desne strane pravila definisane i da se pomoću njih definiše vrednost pojma s leve strane pravila. Ako desna strana pravila sadrži samo jedan pojam ili simbol, njegova vrednost automatski postaje vrednost pojma sa leve strane pravila, ako se ne naznači drugačije.

Parser - broj reči i rečenica

- Lex specifikacija (**examples/text/count/count.l**)
- Yacc specifikacija (**examples/text/count/count.y**)

- Funkcija **yyparse** automatski poziva funkciju **yylex**
- Poziv funkcije **free** je potreban za dealokaciju memorije koja je zauzeta u skeneru posredstvom poziva funkcije **strdup**, a čija adresa je prosleđena parseru posredstvom globalne promenljive **yylval** (i kojoj se pristupa preko metapromenljive **$$_i$$**)

Parser – Yacc

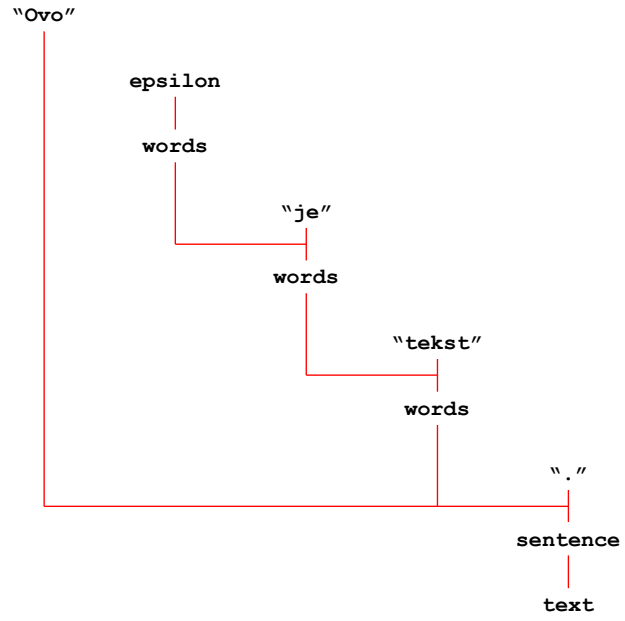
- Kada se Yacc pozove sa opcijom **`-v`**, generiše se datoteka **`y.output`** koja sadrži čitljiv opis tabele akcija i prelaza parsera, a kada se pozove sa opcijom **`-d`**, u datoteku **`y.tab.h`** se kopiraju deklaracija unije koja predstavlja tip promenljive **`yylval`** i definicije tokena.

- `y.output` datoteka za primer brojanja reči i rečenica
- `y.tab.h` datoteka za primer brojanja reči i rečenica

Parser - broj reči i rečenica

- U toku brojanja reči i rečenica parser konstruiše (izvrnuto) drvo parsiranja pa može i da ga prikaže
- Lex specifikacija (**examples/text/count-tree/count.l**)
- Yacc specifikacija (**examples/text/count-tree/count.y**)
- Za ulaz: "Ovo je tekst.", izlaz je:

```
START
```




```
Total words: 3.
Total sentences: 1.

STOP
```

Parser – primer kalkulatora

- Primer gramatike za jednostavni kalkulator

```
lines ->   ε | lines NEWLINE | lines e NEWLINE
e     ->   e + NUMBER | e – NUMBER | NUMBER
```

(podrazumeva se da je **NUMBER** simbol)

- Ulaz: Izlaz:

```
5 + 2          7
1 – 3          -2
```

- Kalkulator definisan prethodnom gramatikom može biti u obliku parsera čije ponašanje opisuje odgovarajuća Yacc specifikacija

- Lex specifikacija

(**examples/calculator/simple/calc.l**)

- Yacc specifikacija

(**examples/calculator/simple/calc.y**)

Parser – primer kalkulatora

- U Yacc specifikaciji je navedeno da kalkulator prihvata jedan izraz u jednoj liniji (prvo pravilo) i kada prepozna izraz odštampa njegovu vrednost (**`printf`** štampa vrednost drugog pojma e). U drugoj grupi pravila je prikazano računanje izraza. Podrazumeva se da vrednost simbola **`NUMBER`** vraća skener u globalnoj promenljivoj **`yylval`**.

Parser – Yacc

- Kada otkrije grešku, podrazumeva se da LR parser pozove funkciju **`yyerror`** radi ispisivanja poruke "*syntax error*" i da zatim završi parsiranje. Međutim Yacc dozvoljava da se umesto podrazumevajućeg obavi korisničko tretiranje greške. To omogućuje korisniku da pokuša oporavak od greške.
- Yacc podržava oporavak od grešaka nastalih za vreme pokušaja redukcije po nekom pravilu tako što dozvoljava da se u takvo pravilo uvede alternativa koja omogućuje redukciju nakon pojave greške. Takva alternativa sadrži Yacc ključnu reč **`error`** i niz pojmova i/ili simbola koji treba da budu otkriveni nakon pojave greške da bi u slučaju greške redukcija bila moguća (neposredno iza **`error`** obavezno sledi simbol). Na ovaj način se može ignorisati niz ulaznih simbola koji sadrže grešku i nastaviti sintaksna analiza iza takvog niza simbola.

Parser – Yacc

- Na primer, da bi se ignorisali svi simboli iz pogrešne linije kod jednostavnog kalkulatora potrebno je u prvo pravilo dodati alternativu

```
| lines error NEWLINE { yyerror("reenter last line:");
                        yyerrok; }
```

- Kada LR parser naiđe na grešku u liniji on primeni prethodno pravilo (pre toga izmeni stanje na steku tako da može da prepozna alternativu pravila koja omogućuje oporavak od greške). Podrazumeva se da je **`yyerror`** funkcija koja prikazuje poruku u slučaju greške, a **`yyerrok`** je makro koji signalizira LR parseru da je greška obrađena i da može da nastavi sa parsiranjem.

- Lex specifikacija (**`examples/calculator/error/calc.l`**)
- Yacc specifikacija (**`examples/calculator/error/calc.y`**)

Parser - dvosmislene gramatike

- Gramatika je dvosmislena (*ambiguous*) ako razna izvođenja dovode do istog niza simbola
- Kod dvosmislenih gramatika za isti niz simbola postoje bar dva različita drveta parsiranja
- Dvosmislene gramatike su problematične, jer dozvoljavaju različita tumačenja jednog-istog niza simbola

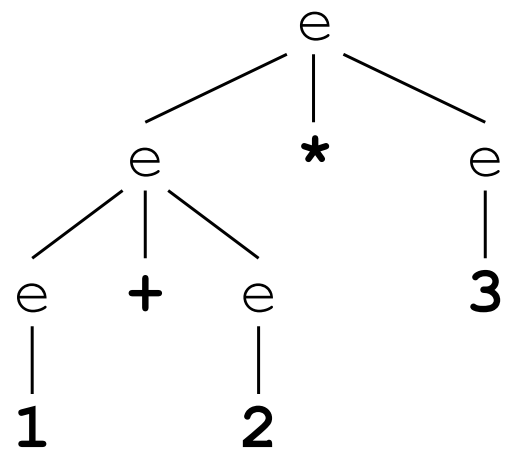
Parser - dvosmislene gramatike

- Primer dvosmislene gramatike
- Gramatika za kalkulator koji podržava i `*` i `/` i unarni minus

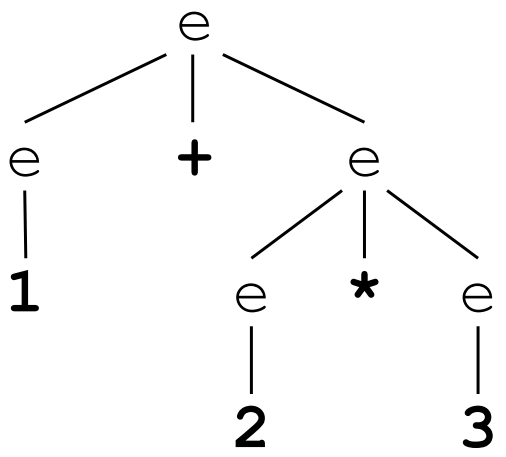
```
e  ->  e + e | e - e | e * e | e / e | -e | NUMBER
```

- Primer dvosmislenosti (dva različita drveta parsiranja za isti niz simbola):

```
1 + 2 * 3
```



Izbor jednog od prethodna dva drveta zavisi od prioriteta operatora



Parser - Yacc

- Dvosmislenosti redosleda primene operatora, zahtevaju da se parseru saopšti redosled njihove primene *(associativity* i *precedence)*
- Redosled primene operatora s leva u desno se zadaje u obliku

```
%left    '+' '-'
```

(primer se odnosi na operatore sabiranja i oduzimanja), a s desna u levo u obliku

```
%right   UMINUS
```

(primer se odnosi na unarni minus)

- Prvo se navode manje prioritetni operatori

```
%left    '+' '-'
%left    '*' '/'
%right    UMINUS
```

(u prethodnom primeru najprioritetniji operator je unarni minus)

- Prioritet operatora se pridružuje pravilu ako se iza pravila navede oznaka **`%prec`** i zatim dotični operator.

Parser - Yacc

- Lex specifikacija potpunog kalkulatora (`examples/calculator/full/calc.l`)
- Yacc specifikacija potpunog kalkulatora (`examples/calculator/full/calc.y`)

Gramatika μC

- Primer gramatike za programski jezik μC (podskup programskog jezika *C*) izražene u *BNF* notaciji

- μC gramatika

  (**grammars/MicroCGrammar.pdf**)

  (analiza svih simbola μC gramatike)

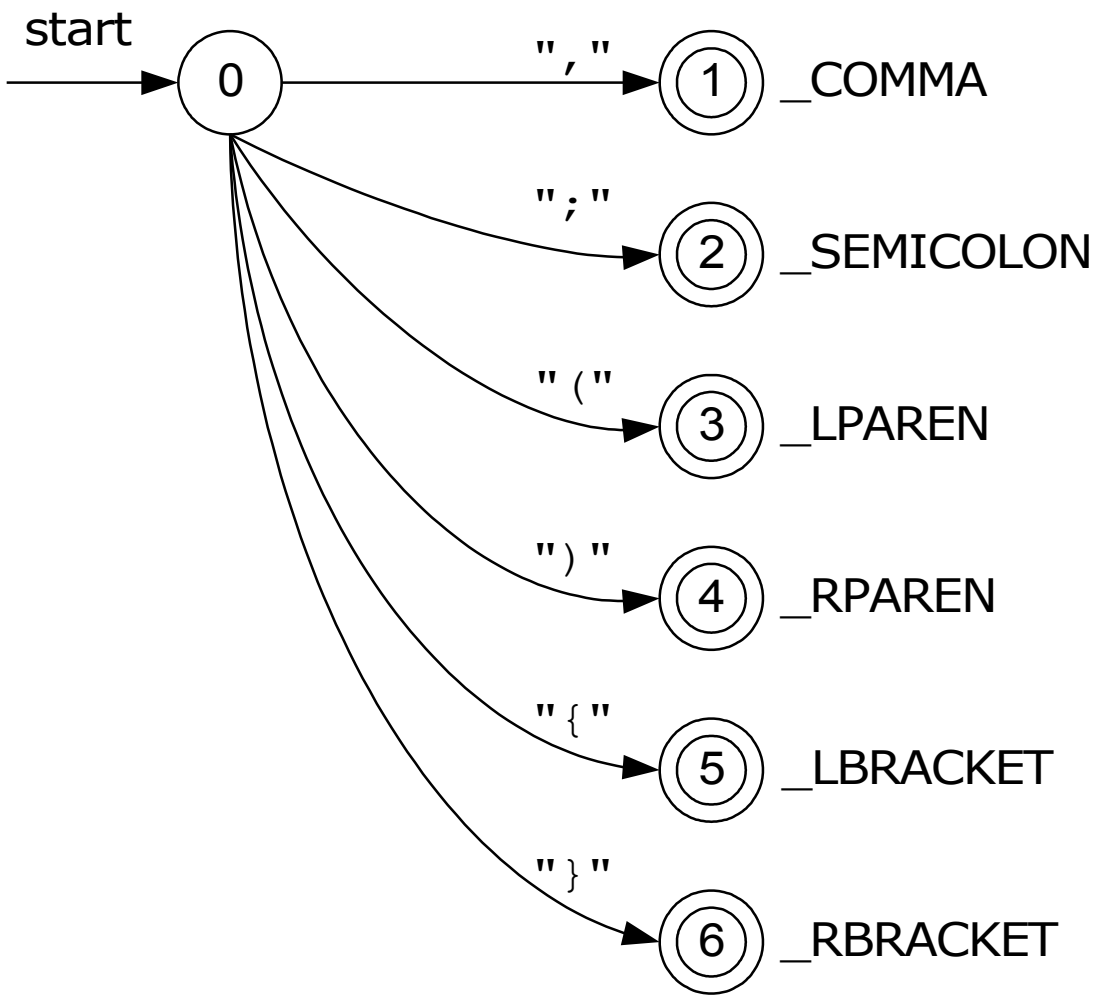
Skener za μC gramatiku

- Za rezervisane reči (**`if`**, **`while`**, **`...`**) je dovoljno da parser dobije token
- Za relacione operatore je potrebno da parser dobije token i redni broj operatora koji predstavlja vrednost simbola
- Za aritmetičke operatore **`*`** i **`/`** je potrebno da parser dobije token i redni broj operatora koji predstavlja vrednost simbola
- Za numeričke konstante je potrebno da parser dobije token i pokazivač stringa konstante koji predstavlja vrednost simbola
- Za identifikatore je potrebno da parser dobije token i pokazivač stringa identifikatora koji predstavlja vrednost simbola

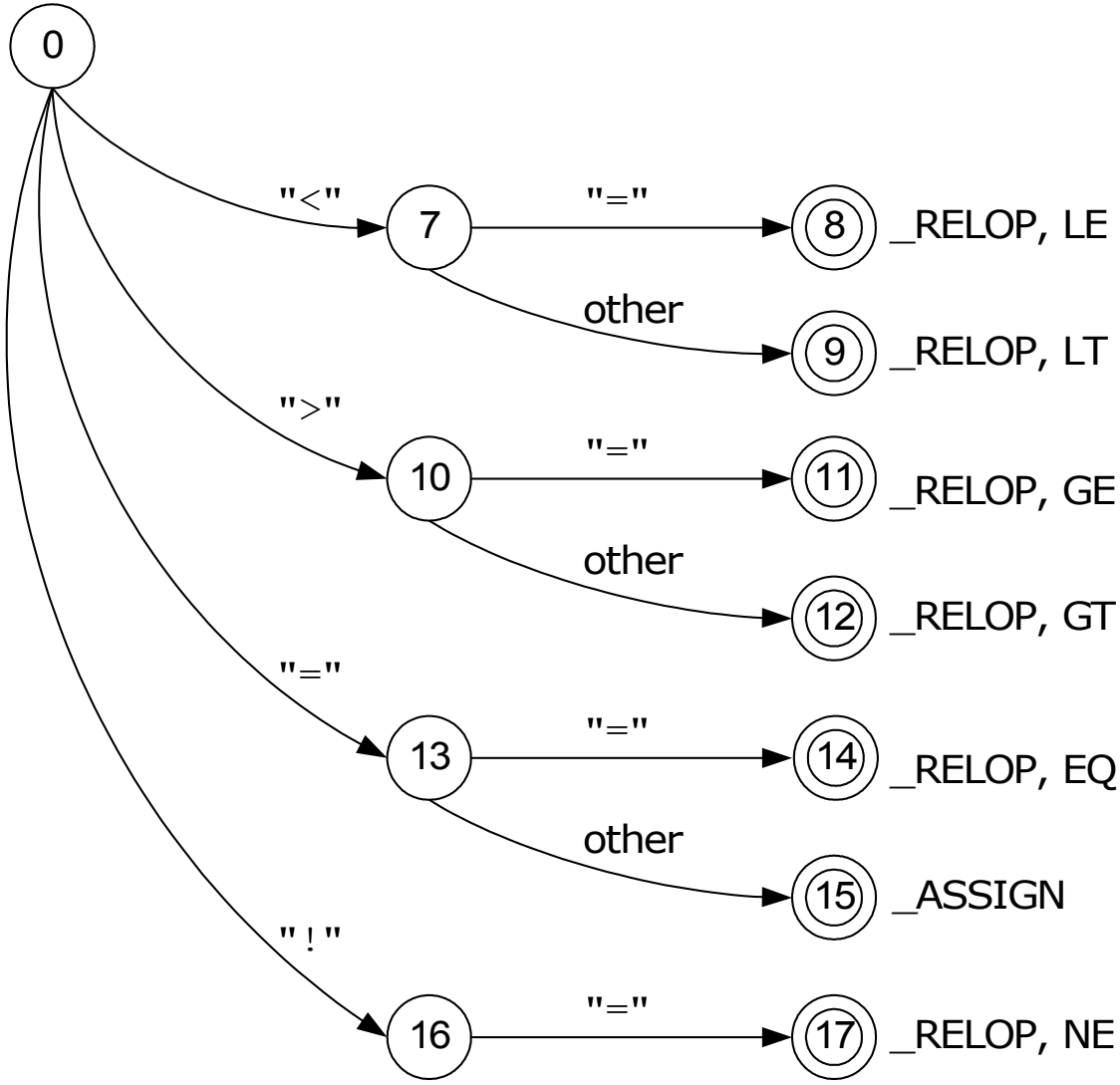
Skener - tokeni

```
#define _TYPE                1
#define _IF                  2
#define _ELSE                3
#define _WHILE               4
#define _RETURN              5
#define _ID                  6
#define _INT_NUMBER          7
#define _UNSIGNED_NUMBER     8
#define _LPAREN              9
#define _RPAREN              10
#define _COMMA               11
#define _LBRACKET            12
#define _RBRACKET            13
#define _ASSIGN              14
#define _SEMICOLON           15
#define _PLUS                16
#define _MINUS               17
#define _MULOP               18
#define _OR                  19
#define _AND                 20
#define _RELOP               21

#define TIMES                1
#define DIV                  2

#define LT                   0
#define GT                   1
#define LE                   2
#define GE                   3
#define EQ                   4
#define NE                   5
```

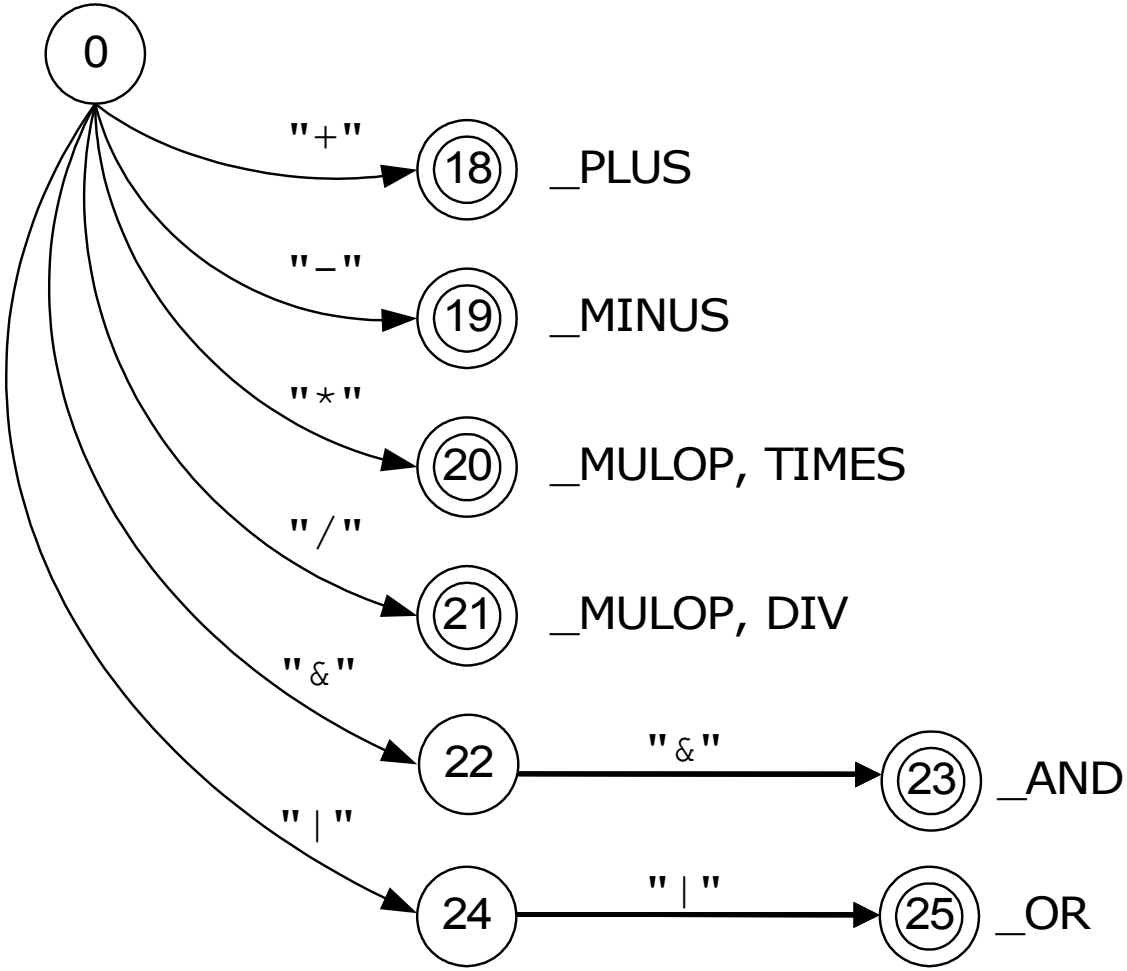
Skener – deterministički konačni automat



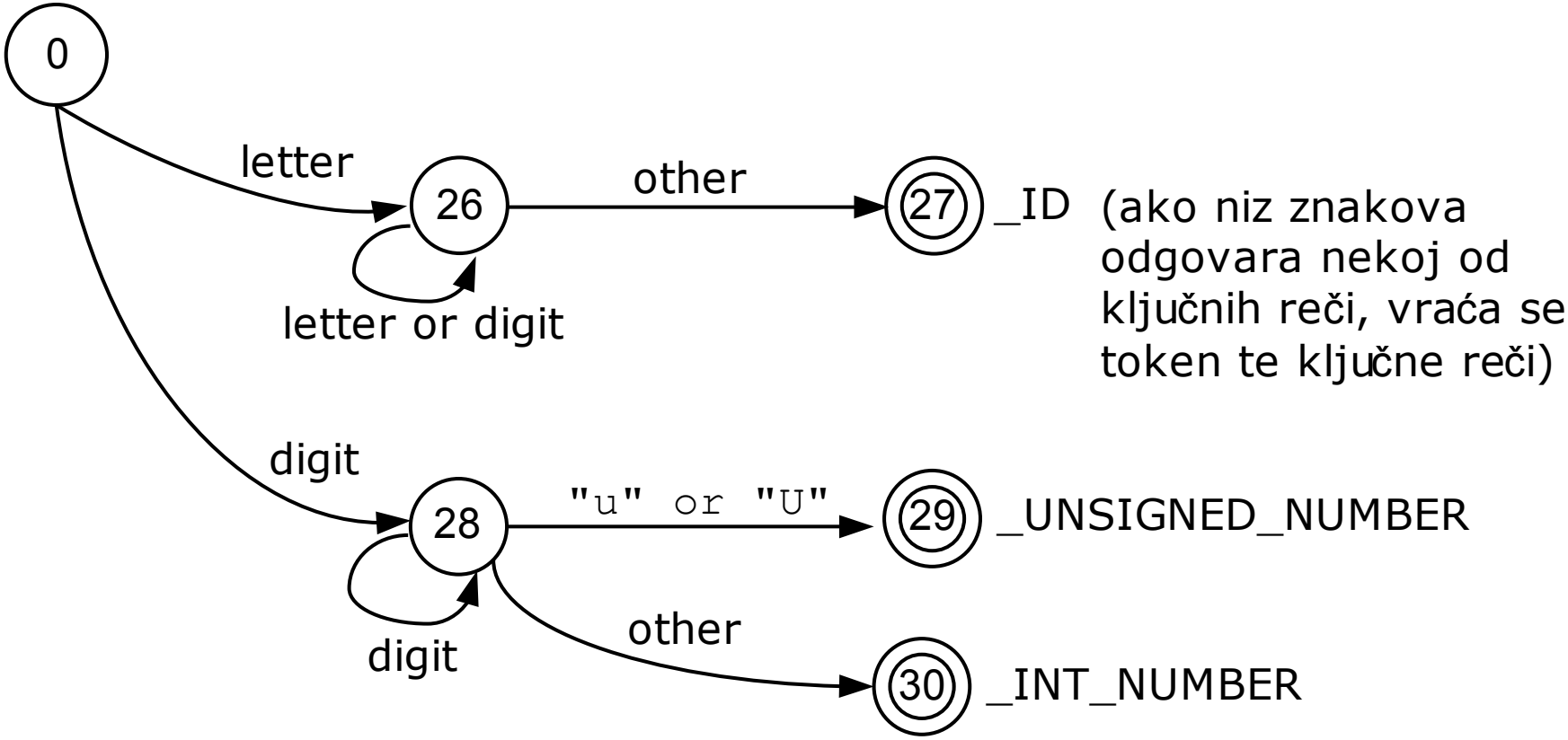
Skener – deterministički konačni automat



Skener – deterministički konačni automat



Skener – deterministički konačni automat



Skener – primer izlaza skenera

```
int abs(int x) {
    if(x < 0)
        return -x;
    else
        return x;
}
```

```
START

   int  TOKEN:          _TYPE  value: INT_TYPE
   abs  TOKEN:            _ID  value: abs
     (  TOKEN:        _LPAREN
   int  TOKEN:          _TYPE  value: INT_TYPE
     x  TOKEN:            _ID  value: x
     )  TOKEN:        _RPAREN
     {  TOKEN:     _LBRACKET
    if  TOKEN:            _IF
     (  TOKEN:        _LPAREN
     x  TOKEN:            _ID  value: x
     <  TOKEN:         _RELOP  value: LT
     0  TOKEN:   _INT_NUMBER  value: 0
     )  TOKEN:        _RPAREN
return  TOKEN:        _RETURN
     -  TOKEN:         _MINUS
     x  TOKEN:            _ID  value: x
     ;  TOKEN:    _SEMICOLON
  else  TOKEN:          _ELSE
return  TOKEN:        _RETURN
     x  TOKEN:            _ID  value: x
     ;  TOKEN:    _SEMICOLON
     }  TOKEN:     _RBRACKET

STOP
```

μC Skener

- definicije konstanti
  (**`examples/microC/scanner/defs.h`**)
- Lex specifikacija
  (**`examples/microC/scanner/scanner.l`**)

Parser za μC gramatiku

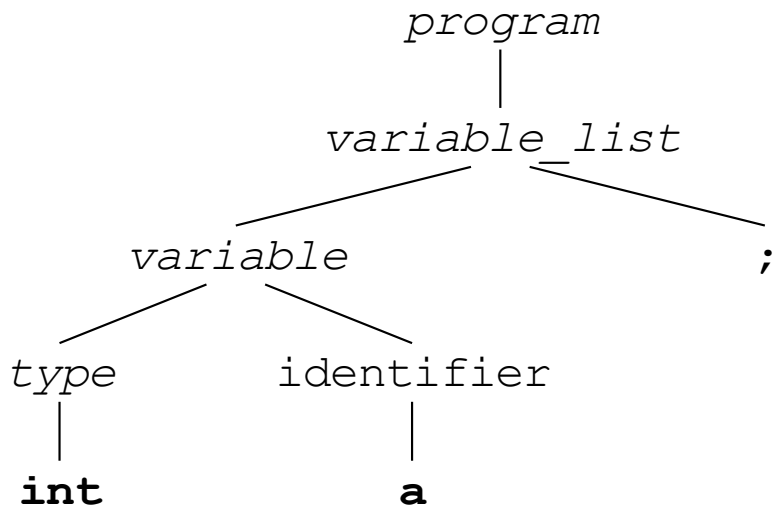
- Primeri nizova izvođenja i korespondentnog drveća parsiranja za iskaz

```
int a;
```

- izvođenje s leva

program ⇒ *variable_list* ⇒
variable ; ⇒
type identifier ; ⇒
int identifier ; ⇒
int a ;

- drvo parsiranja za izvođenje s leva



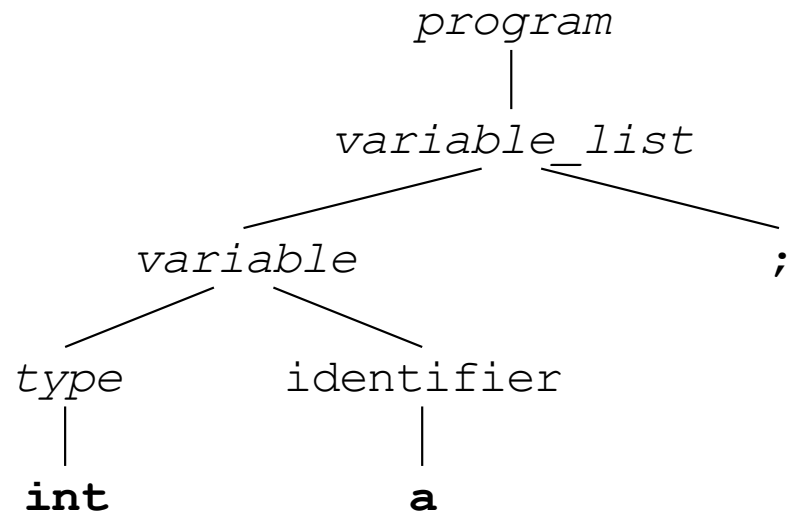
Parser za μC gramatiku

```
int a;
```

- izvođenje s desna

program ⇒ *variable_list* ⇒
variable **;** ⇒
type identifier **;** ⇒
type **a ;** ⇒
int a ;

- drvo parsiranja za izvođenje s desna



Parser - dvosmislene gramatike

- Primer dvosmislene gramatike za iskaz

  `if (a != b) if (a > b) b = a; else a = b;`

  koji opisuje pridruživanje vrednosti veće promenljive manjoj promenljivoj

  - netačno izvođenje s leva

statement ⇒ *if_statement* ⇒

if (*log_exp***)** *statement* **else** *statement* ⇒*

if (a != b) *statement* **else** *statement* ⇒*

if (a != b) *if_statement* **else** *statement* ⇒

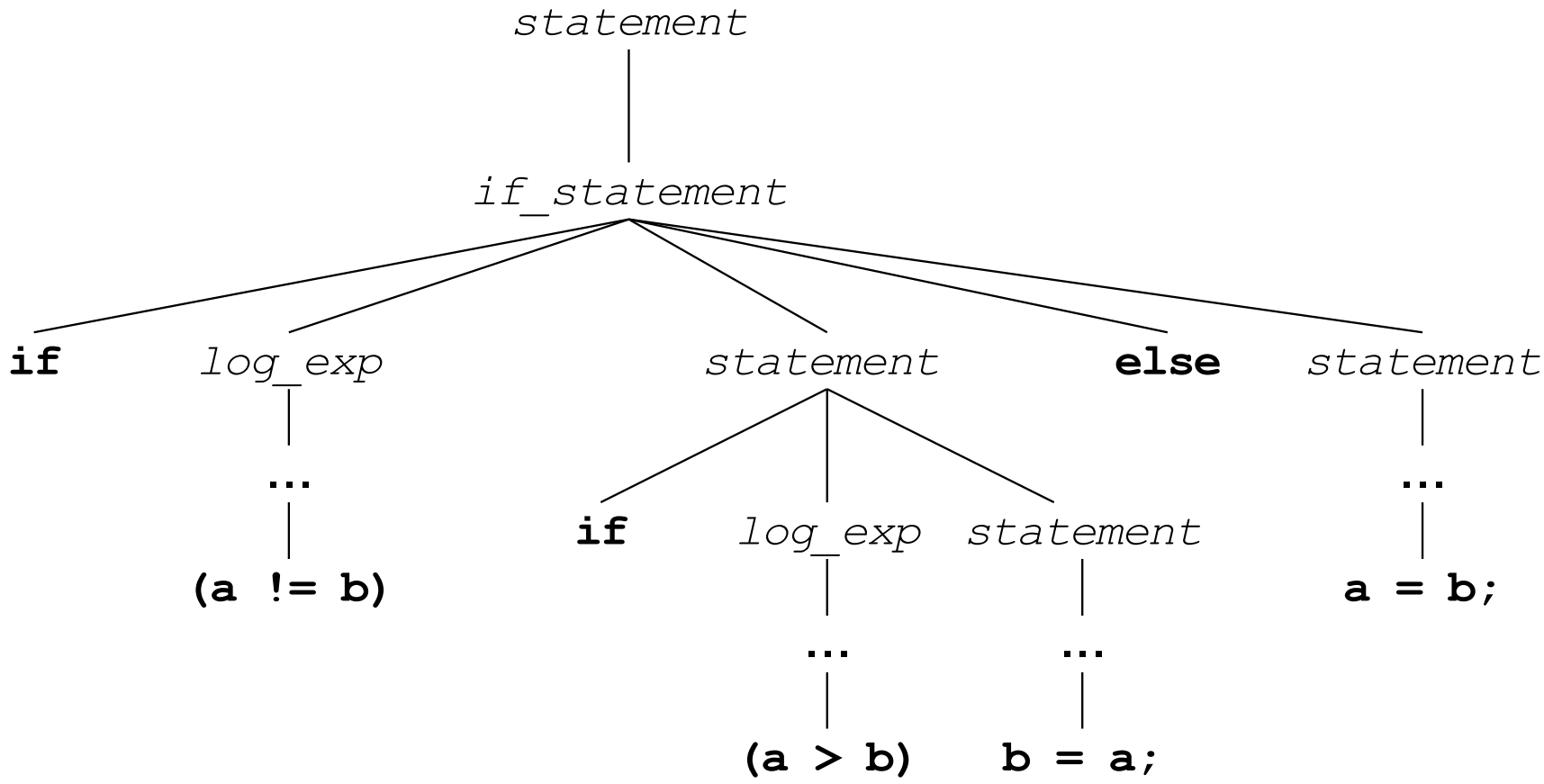
if (a != b) if (*log_exp***)** *statement* **else** *statement* ⇒*

if (a != b) if (a > b) *statement* **else** *statement* ⇒*

if (a != b) if (a > b) b = a; else a = b;

Parser - dvosmislene gramatike

- drvo parsiranja za netačno izvođenje s leva



Parser - dvosmislene gramatike

- tačno izvođenje s leva

statement ⇒ *if_statement* ⇒

if (*log_exp***)** *statement* ⇒*

if (a != b) *statement* ⇒

if (a != b) *if_statement* ⇒

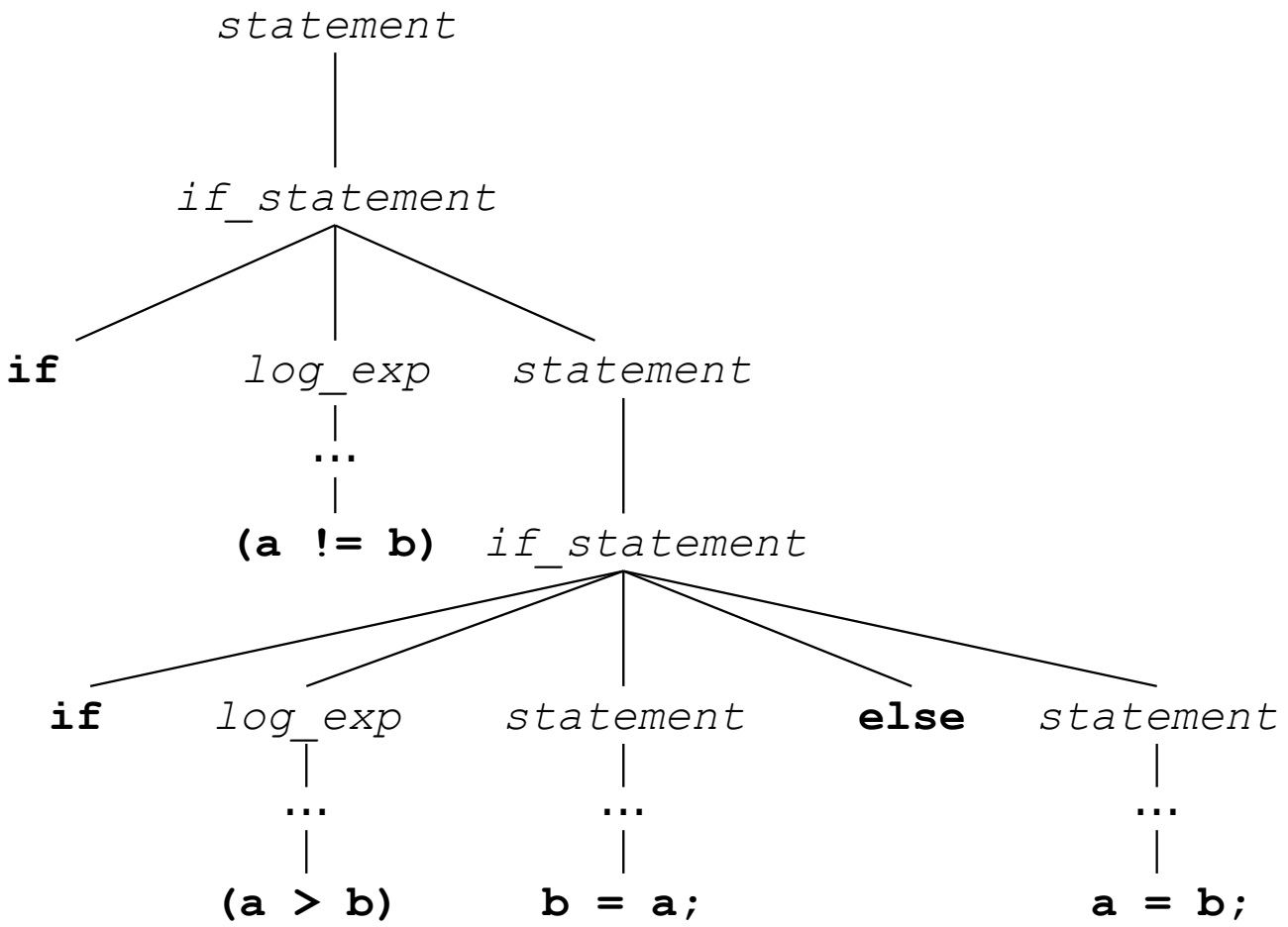
if (a != b) if (*log_exp***)** *statement* **else** *statement* ⇒*

if (a != b) if (a > b) *statement* **else** *statement* ⇒*

if (a != b) if (a > b) b = a; else a = b;

Parser - dvosmislene gramatike

- drvo parsiranja za tačno izvođenje s leva



Parser - dvosmislene gramatike

- Problem dvosmislenosti gramatika je što one u pojedinim koracima izvođenja nude dve ili više ravnopravnih mogućnosti i tako dovode parser u situaciju da ne može automatski da se jednoznačno opredeli za jednu mogućnost – posledica je da razni parseri mogu različito interpretirati isti program

Parser - Yacc

- Generatoru LR parsera moraju biti saopštene nedvosmislene (*unambiguous*) gramatike da bi on u svaki element tabele akcija i prelaza mogao da smesti oznaku samo jedne akcije. U suprotnom se može javiti više akcija kao kandidata za isti element tabele. Ovakve situacije se nazivaju **konflikti**. Za dvosmislene gramatike se ovakvi konflikti razrešavaju davanjem prednosti uvek jednoj akciji. Na primer, kod *shift-reduce* konflikta prednost se uvek može davati *shift* akciji (to rešava problem jednoznačne interpretacije **`if else`** iskaza), a kod *reduce-reduce* konflikta prednost se uvek može davati *reduce* akciji čije pravilo je prvo navedeno. Ako prethodni podrazumevajući način za razrešavanje konflikta, koji primenjuje generator, nije prihvatljiv za korisnika, potrebno je da on generatoru može saopštiti na koji način konflikt treba razrešiti. To se, na primer, može ostvariti navođenjem prioriteta (*precedence*) operatora i redosleda njihove primene (*associativity*)

Parser - Yacc

- Prioritet i redosled primene operatora utiču na način na koji Yacc razrešava *shift-reduce* konflikt. Kada odlučuje da li će da izvrši redukciju po nekom pravilu ili će da pomeri simbol (nekog operatora) sa početka ulaznog niza simbola, Yacc daje prednost redukciji ako je prioritet simbola niži od prioriteta pravila (prioritet pravila je jednak prioritetu krajnje desnog operatora iz desne strane pravila). Redukcija se primenjuje i ako su prioriteti simbola i pravila isti, ali se krajnje desni operator iz desne strane pravila primenjuje s leva u desno. Prioritet pravila se podiže na nivo nekog operatora pomoću oznake **`%prec`** ako se iza pravila navedu ta oznaka i dotični operator.

Parser - Yacc

- *reduce-reduce* konflikt se javlja ukoliko postoji 2 ili više pravila koja se mogu primeniti na isti niz ulaznih simbola.Ovakva situacija označava ozbiljnu grešku u gramatici.

```
sequence  : /* empty */
              { printf ("empty sequence\n"); }
          | maybeword
          | sequence word
              { printf ("added word %s\n", $2); }
          ;
maybeword : /* empty */
              { printf ("empty maybeword\n"); }
          | word
              { printf ("single word %s\n", $1); }
          ;
```

Parser - Yacc

- Greška je u dvosmislenosti: postoji više od jednog načina da parser od pojma ‘word’ dođe do pojma ‘sequence’. Prvi način je redukcija pojma ‘word’ u pojam ‘maybeword’, a zatim redukcija u pojam ‘sequence’ preko drugog pravila. Drugi način je da se ε redukuje u ‘sequence’ preko prvog pravila, a zatim da se ‘sequence’ kombinuje sa ‘word’ pomoću trećeg pravila.
- Takođe, postoji više od jednog načina da se ε redukuje u ‘sequence’. Redukcija je moguća direktno preko prvog pravila, ili indirektno preko pojma ‘maybeword’ i drugog pravila.
- Ove razlike utiču na to koje akcije će biti izvršene. Jedan redosled parsiranja će izvršiti akciju uz drugo pravilo, a drugi akciju uz prvo pravilo i zatim akciju uz treće pravilo. U ovom primeru, menja se izlaz programa.

Parser - Yacc

- Yacc razrešava *reduce-reduce* konflikt tako što odabira pravilo koje je ranije navedeno u gramatici, ali vrlo je rizično osloniti se na ovo. Svaki *reduce-reduce* konflikt se mora dobro proučiti i eliminisati.
- Ispravan način da se definiše pojam 'sequence' je:

```
sequence  : /* empty */
              { printf ("empty sequence\n"); }
          | sequence word
              { printf ("added word %s\n", $2); }
```

Parser

- Prepoznavanje ispravnog niza simbola izvornog programa odgovara konstrukciji drveta parsiranja. To je dovoljan preduslov za sintezu ciljnog programa, jer nakon prepoznavanja iskaza izvornog programskog jezika postaje moguća njihova zamena iskazima ciljnog programskog jezika.

- Yacc specifikacija parsera za μC gramatiku koji konstruiše drvo parsiranja

  (`examples/microC/syntax/parser.y`)
- Lex specifikacija

  (`examples/microC/syntax/scanner.l`)

Parser - primer

- Yacc specifikacija parsera za μC gramatiku koji ispisuje drvo parsiranja

  (**examples/microC/tree/parser.y**)
- Lex specifikacija

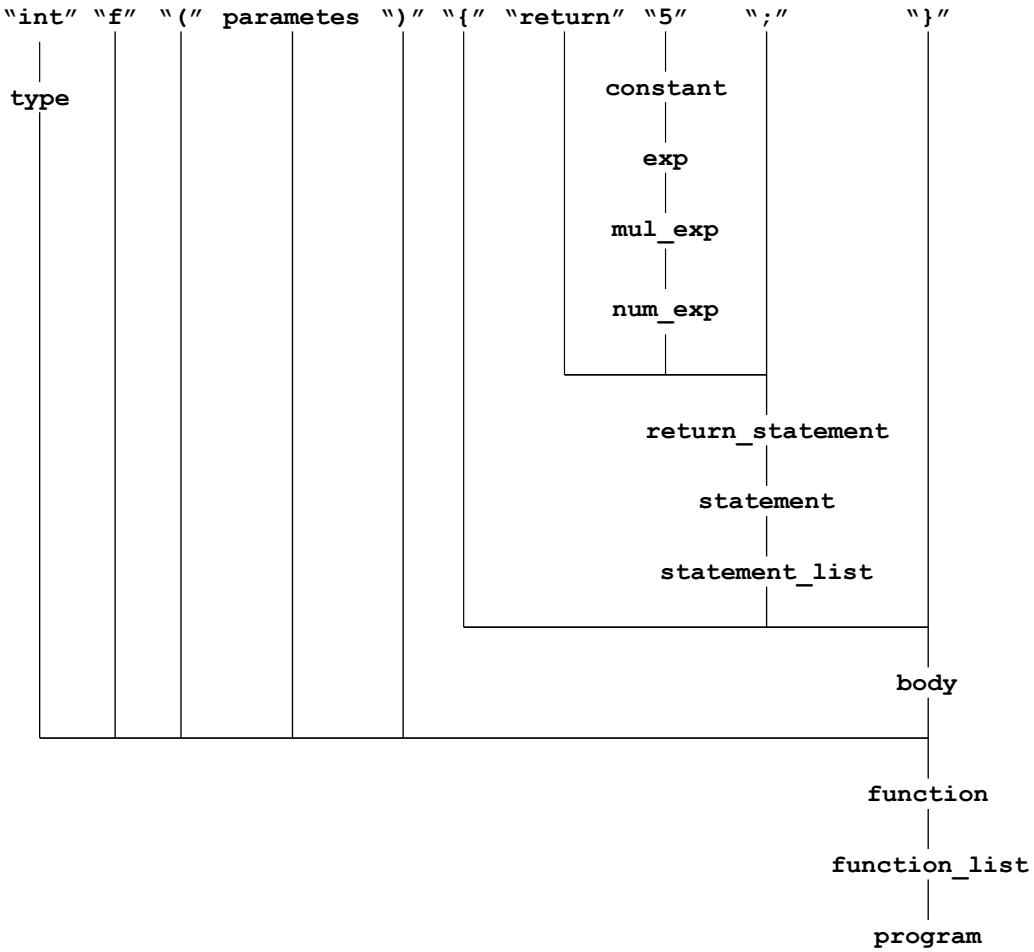
  (**examples/microC/tree/scanner.l**)

- Ako parser može da konstruiše drvo parsiranja za dati program, tada on može da prepozna dotični program, pa je taj program sintaksno ispravan
- Za prikaz konstruisanog drveta parsiranja potrebno je za svaki prepoznati simbol i redukovani pojam prikazati odgovarajući ispis
- U nastavku se razmatra primer LR parsera za μC gramatiku koji pokazuje redosled redukcija, odnosno redosled prepoznavanja pravila
- Podrazumeva se da prethodni primer LR parsera za μC gramatiku koristi odgovarajući Lex skener za istu gramatiku

Primer izlaza LR parsera za μC gramatiku

```
int f() {
      return 5;
}
```

```
START
    "int"

    type
        "f"
            "("

                empty parameters
                    ")"
                        "{"

                            empty statement_list
                                "return"
                                    "5"

                                    constant

                                    exp

                                    mul_exp

                                    num_exp
                                        ";"

                                        return_statement

                                        statement

                                        statement_list
                                            "}"

                                            body

                                            function

                                            function_list

                                            program
STOP
```



Parser - postupak sa greškama

- Nakon otkrivanja greške u programskom tekstu potrebno je navesti
  - opis greške i
  - ukazati na mesto njene pojave
- Na mesto pojave greške ukazuju broj linije programskog teksta sa greškom i eventualno pogrešni deo teksta linije (radi lakšeg snalaženja u linijama programskog teksta, uputno je kao izlaz sintaksne analize predvideti i programski tekst sa rednim brojem svake linije na njenom početku)

μC Parser

- Yacc specifikacija parsera za μC gramatiku koji rukuje sintaksnim greškama

  (**`examples/microC/syntax_errors/parser.y`**)
- Lex specifikacija

  (**`examples/microC/syntax_errors/scanner.l`**)

Semantika

- **Značenje** (semantika) programskog jezika se opisuje na neformalan način

Semantička pravila za μC

- Standardni identifikatori su:
  - rezervisane reči: **`int`**, **`unsigned`**, **`if`**, **`else`**, **`while`**, **`return`**
  - identifikator **`main`** je ime funkcije, za koju se podrazumeva da je definisana u izvršivom μC programu. Izvršavanje μC programa započne izvršavanjem **`main`** funkcije (nakon njenog poziva iz okruženja μC programa). Definicija **`main`** funkcije izgleda:

```
int main ()
{ ... }
```

    Telo **`main`** funkcije definiše korisnik. Ako telo **`main`** (i svake druge) funkcije ne sadrži **`return`** iskaz, podrazumeva se da je povratna vrednost funkcije nedefinisana i da do povratka iz funkcije dolazi po izvršavanju poslednjeg iskaza iz njenog tela.

Semantička pravila za μC

- Područje važenja (doseg, domašaj, vidljivost) identifikatora (*lexical-scope*, *static-scope*)
  - identifikatori se razvrstavaju u globalne i lokalne
  - globalni identifikatori su imena globalnih promenljivih i imena funkcija. Oni su definisani na nivou programa (van funkcija)
  - lokalni identifikatori su imena lokalnih promenljivih. Oni su definisani u okviru funkcija (tu spadaju i parametri funkcija)
  - područje važenja globalnih identifikatora je od mesta njihove definicije do kraja programskog teksta
  - područje važenja lokalnih identifikatora je od mesta njihove definicije do kraja tela funkcije u kojoj su definisani (znači svaka funkcija poseduje svoje lokalne promenljive)
  - identifikatori mogu biti korišćeni samo iza njihove definicije (to proizlazi iz područja njihovog važenja)

Semantička pravila za μC

- Jednoznačnost identifikatora
  - svi globalni identifikatori moraju biti međusobno različiti
  - svi lokalni identifikatori iste funkcije moraju biti međusobno različiti
  - ako postoje identični globalni identifikatori i lokalni identifikatori neke funkcije, tada van te funkcije važe globalni, a unutar nje lokalni identifikatori
  - lokalni identifikatori raznih funkcija mogu biti identični
  - rezervisane reči smeju da se koriste samo u skladu sa svojom ulogom i na globalnom i na lokalnom nivou
  - standardni identifikator **`main`** je rezervisan samo na globalnom nivou

Semantička pravila za μC

- Ispravnost korišćenja identifikatora
  - identifikatori smeju biti korišćeni samo u skladu sa njihovom definicijom:
    - imenu funkcije ne sme biti pridružena vrednost
    - argumenti poziva funkcije moraju da se slažu po broju sa parametrima funkcije
  - na upotrebu identifikatora utiče i njihov tip
  - na primer, tip identifikatora s leve strane iskaza pridruživanja određuje tip izraza sa desne strane ovog iskaza, tako da se u izrazu sa desne strane iskaza priduživanja mogu da pojave samo identifikatori čiji tip je identičan tipu identifikatora sa leve strane ovog iskaza. Isto važi i za konstante

Semantička pravila za μC

- tip izraza iz **`return`** iskaza neke funkcije i tip ove funkcije moraju biti identični
- tipovi korespondentnih parametara funkcije i argumenata iz njenog poziva moraju biti identični
- u istom relacionom izrazu smeju biti samo identifikatori istog tipa. Isto važi i za konstante
- Podrazumeva se da je tip konstante **`int`** ako nije eksplicitno naznačeno da je tip konstante **`unsigned`**

Semantička pravila za μC

- Provera logičkih izraza
  - provera logičkih izraza se završava čim se nedvosmisleno odredi njihova vrednost

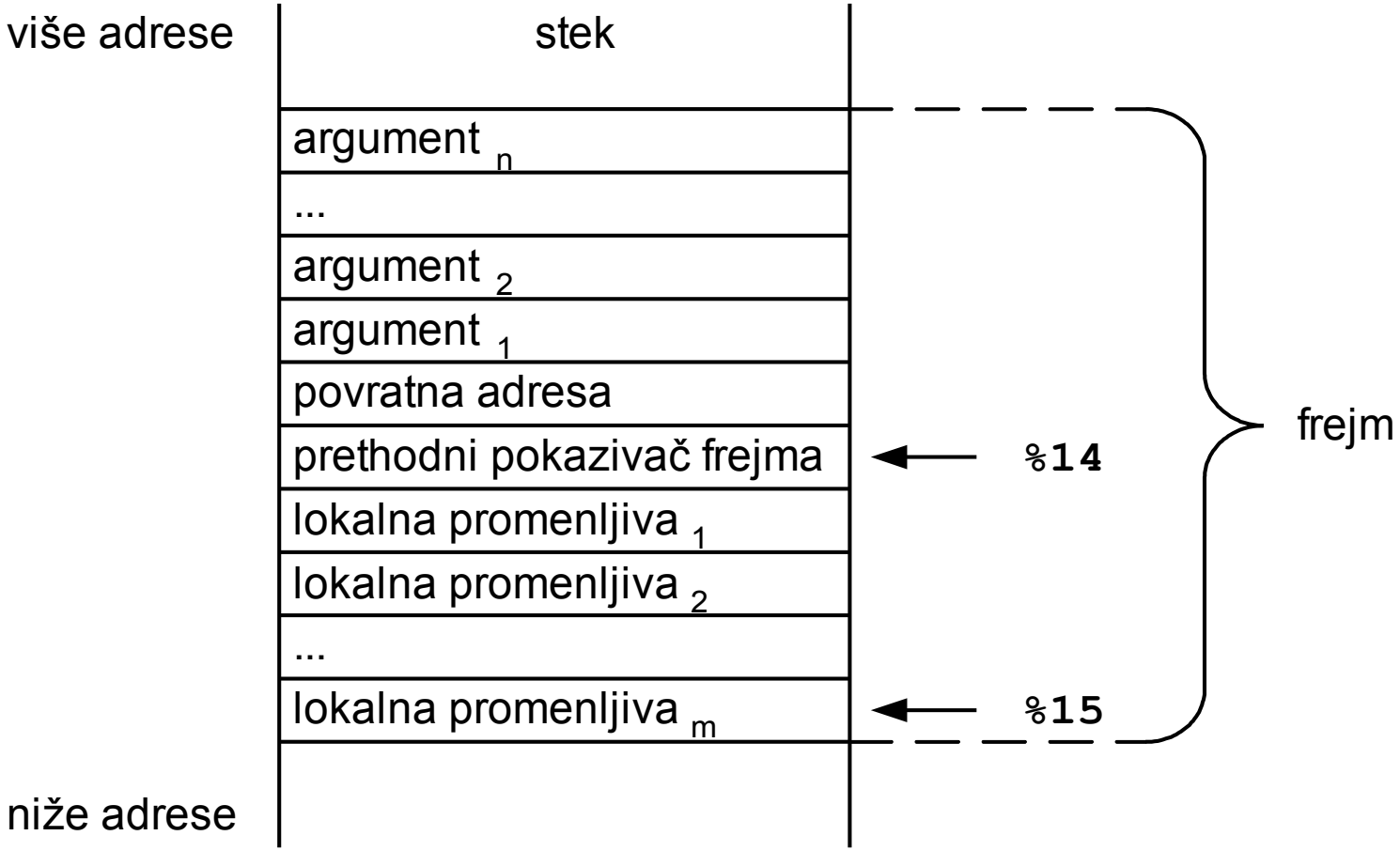
```
(i != 0) && (a/i > 0)
```

Organizacija memorije za μC

- Globalni identifikatori su statični – postoje za sve vreme izvršavanja programa, pa se za njih u vreme kompilacije mogu rezervisati memorijske lokacije
- Lokalni identifikatori su dinamični – postoje samo za vreme izvršavanja funkcija. Za njih se mogu zauzeti memorijske lokacije na početku izvršavanja funkcija. Ove lokacije se moraju osloboditi na kraju izvšavanja funkcija. Zato se lokalnim identifikatorima dodeljuju memorijske lokacije sa steka.
- Deo steka koji se zauzima za izvršavanje neke funkcije se zove (stek) **frejm**.

Organizacija memorije za μC

- Izgled tipičnog frejma:



Organizacija memorije za μC

- Argumenti su vrednosti parametara koji se adresiraju u odnosu na **`%14`**. Tako

  parametru$_1$ odgovara operand **`8(%14)`**

  parametru$_2$ odgovara operand **`12(%14)`**

  ...

  parametru$_n$ odgovara operand **`4+n*4(%14)`**

- I vrednosti lokalnih promenljivih se adresiraju u odnosu na **`%14`**. Tako

  lokalnoj promenljivoj$_1$ odgovara operand **`-4(%14)`**

  lokalnoj promenljivoj$_2$ odgovara operand **`-8(%14)`**

  ...

  lokalnoj promenljivoj$_m$ odgovara operand **`-m*4(%14)`**

- Prethodno je važno jer se u vreme kompilacije ne znaju apsolutne adrese parametara i lokalnih promenljivih

Tabela simbola

- Pronađeni globalni i lokalni identifikatori (simboli) se čuvaju u tabeli simbola

Tabela simbola

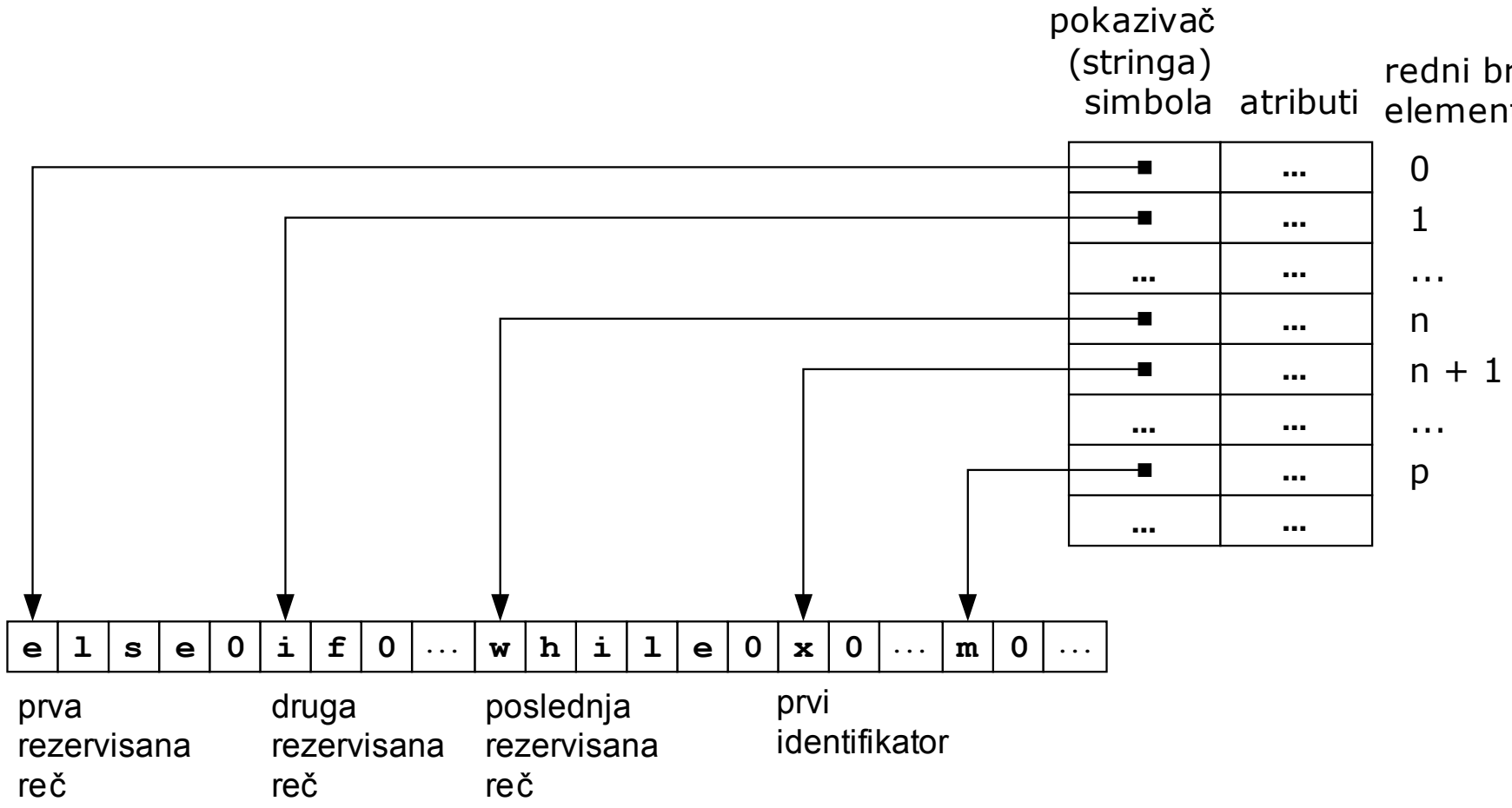


Tabela simbola

- Operacije za rukovanje tabelom simbola
  - **`int insert (char *symbol, unsigned token);`**

    zauzima sledeći prazan element, smešta u njega simbol i token i vraća indeks pomenutog elementa
  - **`int lookup (char *symbol);`**

    vraća indeks elementa koji sadrži simbol ili vraća -1
  - **`void print(void);`**

    ispisuje sadržaj tabele simbola
  - **`void clear_symbols(unsigned begin_index, unsigned end_index);`**

    briše (poništava) elemente tabele simbola sa indeksima od **`begin_index`** do **`end_index`**
- Pomoću operacije **`lookup`** se otkriva da li tabela simbola sadrži prepoznati simbol. Pomoću operacije **`insert`** se simbol i odgovarajući token ubacuju u tabelu simbola.

Tabela simbola

- Skraćenje vremena pretraživanja *(lookup)* tabele simbola
  - za veliki broj simbola u tabeli simbola srednje vreme pretraživanja postaje predugačko
  - srednje vreme pretraživanja tabele simbola se može smanjiti pomoću heš *(hash)* tabele:

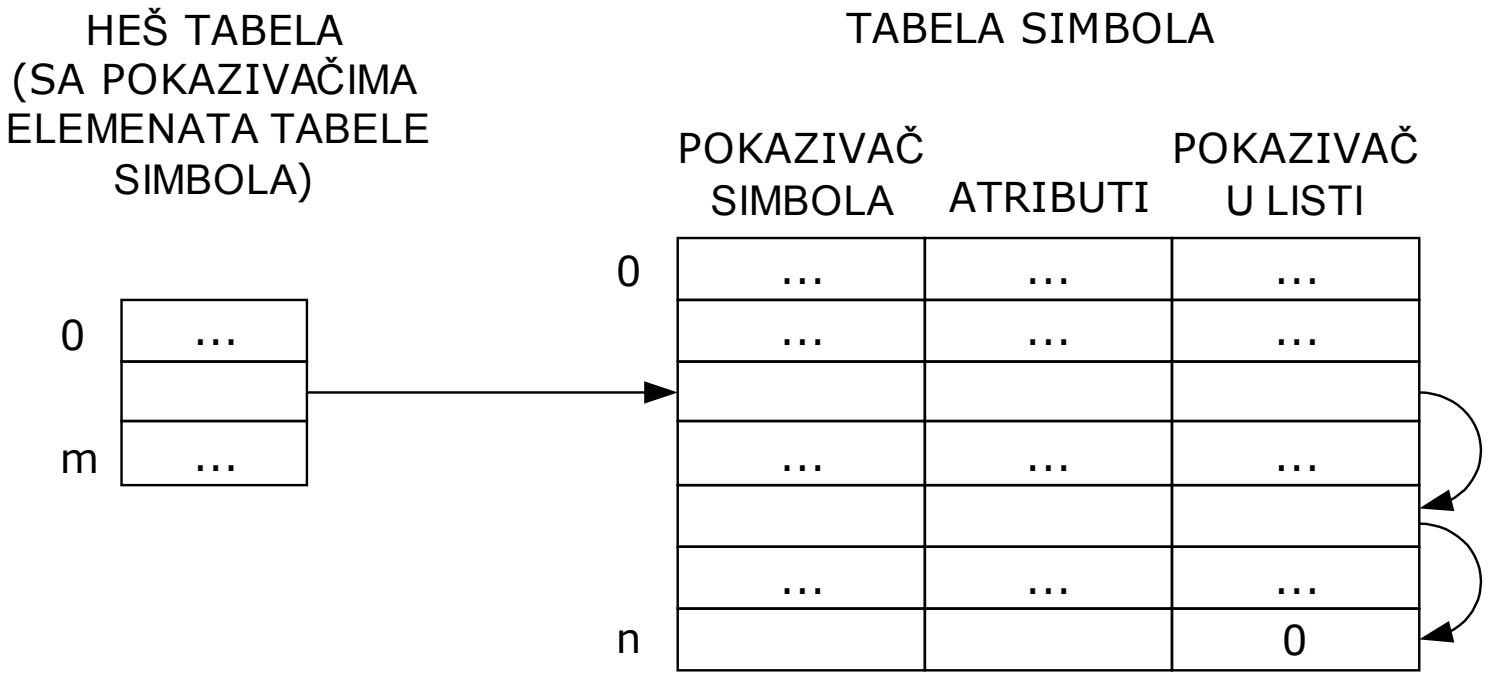


Tabela simbola

- indeks heš tabele daje heš funkcija čiji argument je simbol

```
int key(char *str);
```

- heš funkcija može da daje iste indekse za razne simbole, pa se takvi simboli uvezuju u listu (to omogućuje pokazivač u listi)
- u pretraživanju ovakve tabele simbola sekvencijalno se pretražuje samo lista simbola sa istim heš indeksom, što je znatno kraće od sekvencijalnog pretraživanja cele tabele simbola

```
int find_string(char *str);
```

- funkciju **`key`** koriste funkcije za ubacivanje novog simbola i uklanjanje postojećeg simbola iz heš tabele:

```
int insert_string(char *str);
int extract_string(char *str);
```

Tabela simbola

- Primer heš funkcije (*P.J. Weinberger*)
  - heš broj (pomoću koga se računa heš indeks) se inicijalizuje na 0
  - za svaki znak simbola
    1. biti heš broja se pomeraju u levo za 4 mesta
    2. tako izmenjenom heš broju se dodaje kod znaka
    3. ako među 4 najznačajnija bita heš broja postoje jedinice, tada se modifikuju najmanje značajna 4 bita heš broja pomoću operacije isključivo ili čiji operandi su 4 najznačajnija i 4 najmanje značajna bita heš broja. Nakon toga 4 najznačajnija bita heš broja se anuliraju.
  - ovako dobiveni heš broj se deli brojem elemenata heš tabele (podrazumeva se da je ovaj broj prost broj) i ostatak deljenja predstavlja heš indeks

Tabela simbola

- Primer računanja heš broja za simbol “abc” (**`0x61 0x62 0x63`**)
- Binarne vrednosti heš broja (nakon koraka i):

```
0000 0000 0000 0000  (početna vrednost)
0000 0000 0000 0000  (nakon koraka 1)
0000 0000 0110 0001  (nakon koraka 2)
0000 0000 0110 0001  (nakon koraka 3)
0000 0110 0001 0000  (nakon koraka 1)
0000 0110 0111 0010  (nakon koraka 2)
0000 0110 0111 0010  (nakon koraka 3)
0110 0111 0010 0000  (nakon koraka 1)
0110 0111 1000 0011  (nakon koraka 2)
0000 0111 1000 0101  (nakon koraka 3)
```

Tabela simbola

- I globalni i lokalni identifikatori mogu biti smešteni u istu tabelu simbola (globalni identifikatori su prisutni u tabeli simbola sve vreme kompilacije, a lokalni identifikatori su prisutni u tabeli simbola samo za vreme kompilacije njihove funkcije)
- Pošto u tabeli simbola istovremeno mogu postojati identični globalni i lokalni identifikatori, radi njihovog razlikovanja u tabeli simbola mora biti naznačena vrsta identifikatora (simbola)
- Za svaki identifikator mora postojati i oznaka njegovog tipa
- Za parametre i lokalne promenljive mora postojati njihova relativna pozicija na steku u odnosu na `%14`

Tabela simbola

- Za svaku funkciju mora postojati broj njenih parametara kao i tipovi pojedinih parametara (radi provere ispravnosti njenih poziva). Pošto postoje samo dva tipa, oni se mogu kodirati binarnim ciframa u okviru binarnog broja, tako da najmanje značajna cifra određuje tip prvog parametra, naredna cifra tip drugog parametra i tako dalje
- Radi uniformnosti, u tabeli simbola mogu biti čuvane i konstante. One su lokalne za funkciju i moraju biti jedinstvene
- Takođe, radi uniformnosti, u tabeli simbola mogu biti smeštene i oznake radnih registara. One se smeštaju u tabelu simbola u vreme njene inicijalizacije (`%0` u element sa indeksom `0`, `%1` u element sa indeksom `1`, ..., a `%12` u element sa indeksom `12`) i, poput globalnih identifikatora, prisutne su u tabeli simbola sve vreme kompilacije

Tabela simbola

- Opis mogućih značenja elemenata tabele simbola (za svaki atribut je dovoljan 1 bajt u ovom primeru)

POKAZIVAČ STRINGA SIMBOLA	VRSTA SIMBOLA	TIP SIMBOLA	DODATNI ATRIBUT 1	DODATNI ATRIBUT 2
	RADNI REGISTAR	DA	NE	NE
	GLOBALNA PROMENLJIVA	DA	NE	NE
	FUNKCIJA	DA	BROJ PARAMETARA	TIPOVI PARAMETARA … + $T_2*8^1 + T_1*8^0$
	PARAMETAR	DA	RELATIVNA POZICIJA NA STEKU	NE
	LOKALNA PROMENLJIVA	DA	RELATIVNA POZICIJA NA STEKU	NE
	KONSTANTA	DA	NE	NE

Tabela simbola

- Broj elemenata u tabeli simbola ograničava najveći mogući broj identifikatora u programu i može biti uzrok neuspešne kompilacije
- Broj bita za tipove parametara ograničava najveći mogući broj parametara u funkciji i može biti uzrok neuspešne kompilacije
- Broj radnih registara takođe uvodi ograničenje u radu kompajlera i može biti uzrok neuspešne kompilacije

Semantička analiza

- U okviru semantičke analize proverava se poštovanje semantičkih pravila
- Radi provere poštovanja semantičkih pravila u tabelu simbola se smeštaju odgovarajući atributi identifikatora, čim sintaksna analiza nedvosmisleno ukaže na vrednost atributa. Provera poštovanja semantičkih pravila se zasniva na korišćenju ovih atributa.
- Pojedini koraci provere poštovanja semantičkih pravila se obavljaju nakon prepoznavanja pojedinih sintaksnih pravila u toku sintaksne analize. Ovi koraci se nazivaju **semantičke akcije** i vezuju se za pomenuta sintaksna pravila
- Uporedni prikaz sintaksnih pravila i za njih vezanih semantičkih akcija obrazuje **sintaksno zasnovanu definiciju** (*syntax-directed definition*)

Semantička analiza

- U nastavku su navedeni pojmovi i opis semantičkih akcija vezanih za prepoznavanje ovih pojmova:
  - *program*
    - proveriti da li u tabeli simbola postoji **main** funkcija
  - *variable*
    - proveriti da li je promenljiva već definisana – ako nije upisati atribute (vrsta simbola i tip)
  - *parameter*
    - kao i za *variable* uz dopisivanje relativne pozicije na steku
    - brojati parametre i registrovati njihove tipove
  - *function*
    - kao i za *variable*

Semantička analiza

- *assignment*
  - proveriti da li su tipovi leve i desne strane isti
- *argument*
  - brojati argumente i registrovati njihove tipove
- *function_call*
  - uporediti broj parametara i argumenata i njihove tipove
- *identifier*
  - proveriti da li je simbol definisan i da li vrsta simbola odgovara mestu njegovog korišćenja
- *mul_exp*, *num_exp*, *rel_exp*
  - proveriti da li su tipovi levog i desnog operanda isti
- *return*
  - proveriti da li je tip *num_exp* isti kao i tip funkcije

Semantička analiza - primer

- primer semantičke analize
  (**examples/microC/semantic/scanner.l**,
  examples/microC/semantic/parser.y)

Primer generisanja koda – globalne promenljive

- Globalne promenljive
  - za svaku promenljivu generisati asemblersku direktivu sa odgovarajućom labelom
  - primer globalne promenljive

```
int a;
```

  - generisani kod

```
a:
        WORD    1
```

Primer generisanja koda – iskaz pridruživanja

- Iskaz pridruživanja
  - primer iskaza pridruživanja

    ```
    a = (-a + b) * (c + d) - e
    ```

    (podrazumeva se da su **a**, **b**, **c**, **d** i **e** celobrojne globalne promenljive)
  - za smeštanje međurezultata izraza koriste se radni registri

Primer generisanja koda – iskaz pridruživanja

- generisani kod za iskaz: **`a = (-a + b) * (c + d) - e`**

```
SUBS   $0,a,%0          linija 1
ADDS   %0,b,%0          linija 2
ADDS   c,d,%1           linija 3
MULS   %0,%1,%0         linija 4
SUBS   %0,e,%0          linija 5
MOV    %0,a             linija 6
```

- podrazumeva se da su svi radni registri slobodni. U *liniji 1* se zauzima radni registar `%0`. On se oslobađa i ponovo zauzima u *liniji 2*. U *liniji 3* se zauzima radni registar `%1`. U *liniji 4* se oslobađaju radni registri `%0` i `%1`, a ponovo se zauzima radni registar `%0`. U *liniji 5* se oslobađa i ponovo zauzima radni registar `%0`, a u *liniji 6* se oslobađa radni registar `%0`.

Primer generisanja koda – iskaz pridruživanja

- radni registar se zauzima za smeštanje rezultata svakog aritmetičkog izraza sa:
  - dva operanda i jednim operatorom

    (*num_exp*, *mul_exp*)
  - unarnim operatorom i jednim operandom

    (*exp*)
- radni registar se oslobađa čim se preuzme njegova vrednost

Primer generisanja koda – iskaz pridruživanja

- pošto se međurezultati izraza koriste u suprotnom redosledu od onog u kome su izračunati, radni registri, koji se koriste za smeštanje međurezultata, se zauzimaju i oslobađaju po principu steka. Kao "pokazivač steka" koristi se promenljiva **`free_reg_num`**, koja sadrži broj prvog slobodnog radnog registra. Zauzimanje radnog registra se sastoji od preuzimanja vrednosti promenljive **`free_reg_num`** i njenog inkrementiranja. Oslobađanje radnog registra se sastoji od dekrementiranja promenljive **`free_reg_num`**.
- treba zapaziti da je broj registra istovremeno i indeks elementa tabele simbola

Primer generisanja koda – iskaz pridruživanja

- broj zauzetog radnog registra služi kao vrednost sintaksnog pojma koji odgovara izrazu čiji rezultat radni registar sadrži
- prekoračenje broja radnih registara (**`free_reg_num > 12`**) predstavlja fatalnu grešku u radu kompajlera

Primer generisanja koda – funkcija

- Funkcija
  - primer funkcije

```
int f(int x, int y)
{  int z;
   return x + y; }
```

  - za smeštanje povratne vrednosti funkcije koristi se radni registar **%13**

Primer generisanja koda – funkcija

- generisani kod

```
f:                                          linija 1
        PUSH   %14                          linija 2
        MOV    %15,%14                      linija 3
        SUB    %15,$4,%15                   linija 4
f_body:                                     linija 5
        ADDS   8(%14),12(%14),%0            linija 6
        MOV    %0,%13                       linija 7
        JMP    f_exit                       linija 8
f_exit:                                     linija 9
        MOV    %14,%15                      linija 10
        POP    %14                          linija 11
        RET                                 linija 12
```

Primer generisanja koda – funkcija

- u *linijama 2 i 3* se postavlja pokazivač frejma
- u *liniji 4* se zauzima prostor na steku za lokalnu promenljivu **`z`**. Da bi se odredila veličina prostora za lokalne promenljive, potreban je brojač lokalnih promenljivih **`var_num`**. Veličina prostora za lokalne promenljive je **`var_num * 4`**. Ovaj prostor se zauzima samo ako je **`var_num > 0`**.
- u *linijama 6 i 7* se računa povratna vrednost funkcije i smešta u registar **`%13`**. Ako funkcija ne sadrži **`return`** iskaz, kao povratna vrednost funkcije služi zatečeni sadržaj registra **`%13`**, koji je nepoznat u vreme definisanja funkcije.
- u *liniji 10* se oslobađa prostor za lokalne promenljive, a u *liniji 11* se vraća prethodna vrednost u pokazivač frejma.

Primer generisanja koda – poziv funkcije

- Poziv funkcije
  - primer poziva funkcije

    `a = f(a + b, c - d);`

    (podrazumeva se da su **`a`**, **`b`**, **`c`** i **`d`** celobrojne globalne promenljive)
  - generisani kod

```
ADDS   a,b,%0          linija 1
SUBS   c,d,%1          linija 2
PUSH   %1              linija 3
PUSH   %0              linija 4
CALL   f               linija 5
ADDU   %15,$8,%15      linija 6
MOV    %13, a          linija 7
```

Primer generisanja koda – poziv funkcije

- u *linijama 1 i 2* se računaju argumenti i čuvaju u radnim registrima (pretpostavka je da su svi radni registri slobodni)
- u *linijama 3 i 4* argumenti se smeštaju na stek
- u *liniji 6* oslobađa se prostor koji su zauzimali argumenti. Veličina oslobađanog prostora je **`arg_num * 4`**. Ovaj prostor se oslobađa samo ako je **`arg_num > 0`**.
- u *liniji 7* se isporučuje povratna vrednost funkcije

Primer generisanja koda – poziv funkcije

- ako su argumenti konstante ili promenljive, kao u slučaju poziva

```
a = f(1,b);
```

(podrazumeva se da su **a** i **b** celobrojne globalne promenljive) onda nisu potrebni radni registri, jer se konstante i promenljive direktno mogu smeštati na stek:

```
PUSH  b
PUSH  $1
CALL  f
ADDU  %15,$8,%15
MOV   %13,a
```

Primer generisanja koda – poziv funkcije

- pošto argumentima odgovaraju ili radni registri ili konstante ili promenljive, svakom argumentu je pridružen indeks elementa tabele simbola koji je rezervisan ili za pomenuti radni registar, ili za pomenutu konstantu ili za pomenutu promenljivu
- pošto se argumenti na stek smeštaju od poslednjeg ka prvom, zgodno je na posebnom **steku argumenata** kompajlera čuvati njihove indekse. Ovi indeksi se smeštaju na stek argumenata od prvog ka poslednjem argumentu, tako da su pripremljeni za generisanje naredbi koje smeštaju argumente na stek. Za čuvanje broja argumenata iz poziva funkcije zgodno je uvesti poseban **stek broja argumenata** kompajlera (ovaj stek omogućuje da se pamte brojevi argumenata kada se u pozivu neke funkcije kao argument javi opet poziv funkcije)

Primer generisanja koda – poziv funkcije

- ako su argumenti pozivi novih funkcija

```
a = f(1,f2(3));
```

(podrazumeva se da je **a** celobrojna globalna promenljiva, a da funkcija **f2** ima jedan parametar tipa **int**)

- generisani kod

```
PUSH    $3                  linija 1
CALL    f2                  linija 2
ADDU    %15,$4,%15          linija 3
MOV     %13,%0              linija 4
PUSH    %0                  linija 5
PUSH    $1                  linija 6
CALL    f                   linija 7
ADDU    %15,$8,%15          linija 8
MOV     %13,a               linija 9
```

Primer generisanja koda – poziv funkcije

- kada se u pozivu neke funkcije kao argument javi opet poziv funkcije, prvo se izvršava poziv te funkcije da bi se izračunala vrednost odgovarajućeg argumenta
- u *linijama 1, 2 i 3* se izvršava poziv funkcije **`f2`** da bi se izračunala vrednost drugog argumenta poziva funkcije **`f`**

- U slučaju rekurzivne pojave poziva funkcije na mestu argumenta poziva neke druge funkcije zgodno je uvesti poseban **stek poziva funkcija** kompajlera. Ovaj stek omogućuje da se pamte indeksi (imena) funkcija u tabeli simbola čiji pozivi su se javili kao argument poziva neke funkcije.

Primer generisanja koda – iskaz poziva funkcije

- Iskaz poziva funkcije
  - primer iskaza poziva funkcije

```
f(1,2);
```

  - generisani kod

```
PUSH   $2
PUSH   $1
CALL   f
ADDU   %15,$8,%15
```

Primer generisanja koda – `if` iskaz

- `if` iskaz sa `else` delom
  - primer `if` iskaza

```
if(a > b)
  a = 1;
else
  a = 2;
```

(podrazumeva se da su `a` i `b` celobrojne označene globalne promenljive)

  - generisani kod

Primer generisanja koda – `if` iskaz

```
if0:                                    linija 1
        CMPS    a,b                     linija 2
        JLES    false0                  linija 3
true0:                                  linija 4
        MOV     $1,a                    linija 5
        JMP     exit0                   linija 6
false0:                                 linija 7
        MOV     $2,a                    linija 8
exit0:                                  linija 9
```

Primer generisanja koda – `if` iskaz

- Labele u generisanom kodu moraju biti jedinstvene. To se postiže generisanjem broja koji se dodaje na kraj labele.
- Zato se uvodi promenljiva **`lab_no`** koja sadrži aktuelni broj (labele).
- Generisanje jednoznačnih brojeva se postiže inkrementiranjem promenljive **`lab_no`**.
- Broj uz labelu **`if`** se čuva jer se koristi i uz labele **`true`** i **`exit`**. Za to služi promenljiva **`self_lab_no`**.

Primer generisanja koda – `if` iskaz

- `if` iskaz sa `else` delom
  - primer `if` iskaza

```
if(a > b && c > d || a > c && b > d || a > e)
  a = 1;
else
  a = 2;
```

(podrazumeva se da su `a`, `b`, `c`, `d` i `e` celobrojne označene globalne promenljive)

  - generisani kod (podrazumeva se da se generisanje koda nastavlja na prethodni primer)

Primer generisanja koda – `if` iskaz

```
if1:                                    linija 1
        CMPS    a,b                     linija 2
        JLES    false1                  linija 3
        CMPS    c,d                     linija 4
        JGTS    true1                   linija 5
false1:                                 linija 6
        CMPS    a,c                     linija 7
        JLES    false2                  linija 8
        CMPS    b,d                     linija 9
        JGTS    true1                   linija 10
false2:                                 linija 11
        CMPS    a,e                     linija 12
        JLES    false3                  linija 13
true1:                                  linija 14
        MOV     $1,a                    linija 15
        JMP     exit1                   linija 16
false3:                                 linija 17
        MOV     $2,a                    linija 18
exit1:                                  linija 19
```

Primer generisanja koda – `if` iskaz

- Pošto se kao **`then`** iskaz **`if`** iskaza može da pojavi novi **`if`** iskaz i tako dalje, neophodno je sačuvati vrednost promenljive **`self_lab_no`**, jer će ona biti izmenjena u toku generisanja koda za novi **`if`** iskaz.
- Iz istih razloga se mora sačuvati i vrednost promenljive **`lab_no`** pošto se ona koristi uz poslednju labelu **`false`** u delu koda koji još nije izgenerisan za spoljašnji **`if`** iskaz.
- Za čuvanje vrednosti ovih promenljivih se koristi poseban **stek labela** kompajlera. Na njega se ove vrednosti smeštaju pre tretiranja **`then`** iskaza. Nakon tretiranja **`then`** iskaza ove vrednosti se preuzimaju sa steka labela. Isto važi i za vrednost promenljive **`self_lab_no`** pre i nakon tretiranja **`else`** iskaza.

Primer generisanja koda – `if` iskaz

- Od *linije 2* do *linije 13* je opisano određivanje vrednosti logičkog izraza. Generisanje naredbi poređenja je vezano za pojam *`rel_exp`*. Kao vrednost ovog pojma služi vrednost relacionog operatora, da bi se na osnovu nje mogla izgenerisati ispravna naredba uslovnog skoka. Njeno generisanje je vezano za `&&` operator pojma *`and_exp`*, `||` operator pojma *`log_exp`* ili za pojam *`log_exp`*.

Primer generisanja koda – `if` iskaz

- `if` iskaz bez `else` dela
  - primer `if` iskaza

```
if(a > b)
  a = 1;
```

(podrazumeva se da su `a` i `b` celobrojne označene globalne promenljive)

  - generisani kod (podrazumeva se da se generisanje koda nastavlja na prethodni primer)

Primer generisanja koda – `if` iskaz

```
if4:
        CMPS   a,b
        JLES   false4
true4:
        MOV    $1,a
        JMP    exit4
false4:
exit4:
```

Primer generisanja koda – `if` iskaz

- `if` iskaz bez `else` dela
  - primer `if` iskaza

```
if(a > b && c > d || a > c && b > d || a > e)
  a = 1;
```

(podrazumeva se da su `a`, `b`, `c`, `d` i `e` celobrojne označene globalne promenljive)

  - generisani kod (podrazumeva se da se generisanje koda nastavlja na prethodni primer)

Primer generisanja koda – `if` iskaz

```
if5:                                    linija 1
        CMPS    a,b                     linija 2
        JLES    false5                  linija 3
        CMPS    c,d                     linija 4
        JGTS    true5                   linija 5
false5:                                 linija 6
        CMPS    a,c                     linija 7
        JLES    false6                  linija 8
        CMPS    b,d                     linija 9
        JGTS    true5                   linija 10
false6:                                 linija 11
        CMPS    a,e                     linija 12
        JLES    false7                  linija 13
true5:                                  linija 14
        MOV     $1,a                    linija 15
false7:                                 linija 16
exit5:                                  linija 17
```

- Od *linije 2* do *linije 13* je opisano određivanje vrednosti logičkog izraza

Primer generisanja koda – **`while`** iskaz

- **`while`** iskaz
  - primer **`while`** iskaza

```
while(a > b)
    a = e;
```

(podrazumeva se da su **`a`**, **`b`** i **`e`** celobrojne označene globalne promenljive)

  - za generisanje koda primenjuje se pristup objašnjen u **`if`** iskazu
  - generisani kod (podrazumeva se da se generisanje koda nastavlja na prethodni primer)

Primer generisanja koda – **`while`** iskaz

```
while8:
        CMPS   a,b
        JLES   false8
true8:
        MOV    e,a
        JMP    while8
false8:
```

Primer generisanja koda – **`while`** iskaz

- **`while`** iskaz
  - primer **`while`** iskaza

```
while(a > b && c > d || a > c && b > d || a > e)
    a = e;
```

(podrazumeva se da su **`a`**, **`b`**, **`c`**, **`d`** i **`e`** celobrojne označene globalne promenljive)

  - za generisanje koda primenjuje se pristup objašnjen u **`if`** iskazu
  - generisani kod (podrazumeva se da se generisanje koda nastavlja na prethodni primer)

Primer generisanja koda – **`while`** iskaz

```
while9:                                  linija 1
        CMPS    a,b                      linija 2
        JLES    false9                   linija 3
        CMPS    c,d                      linija 4
        JGTS    true9                    linija 5
false9:                                  linija 6
        CMPS    a,c                      linija 7
        JLES    false10                  linija 8
        CMPS    b,d                      linija 9
        JGTS    true9                    linija 10
false10:                                 linija 11
        CMPS    a,e                      linija 12
        JLES    false11                  linija 13
true9:                                   linija 14
        MOV     e,a                      linija 15
        JMP     while9                   linija 16
false11:                                 linija 17
```

- Od *linije 2* do *linije 13* je opisano određivanje vrednosti logičkog izraza

Primer generisanja koda – **`break`** iskaz

- **`break`** iskaz
  - primer **`break`** iskaza

```
while(a < 5) {
   if(a == 3)
      break;
   a = a + 1;
}
```

(podrazumeva se da je **`a`** celobrojna označena globalna promenljiva) i da se **`break`** iskaz sme naći samo unutar **`while`** iskaza

  - generisani kod (podrazumeva se da se generisanje koda nastavlja na prethodni primer)

Primer generisanja koda – **`break`** iskaz

```
while12:
        CMPS    a,$5
        JGES    false12
true12:
if13:
        CMPS    a,$3
        JNE     false13
true13:
        JMP     false12         //break
        JMP     exit13
false13:
exit13:
        ADDS    a,$1,%0
        MOV     %0,a
        JMP     while12
false12:
```

Primer generisanja koda – **`continue`** iskaz

- **`continue`** iskaz
  - primer **`continue`** iskaza

```
while(a < 5) {
    if(a == 3)
        continue;
    a = a + 1;
}
```

(podrazumeva se da je **`a`** celobrojna označena globalna promenljiva) i da se **`continue`** iskaz sme naći samo unutar **`while`** iskaza

  - generisani kod (podrazumeva se da se generisanje koda nastavlja na prethodni primer)

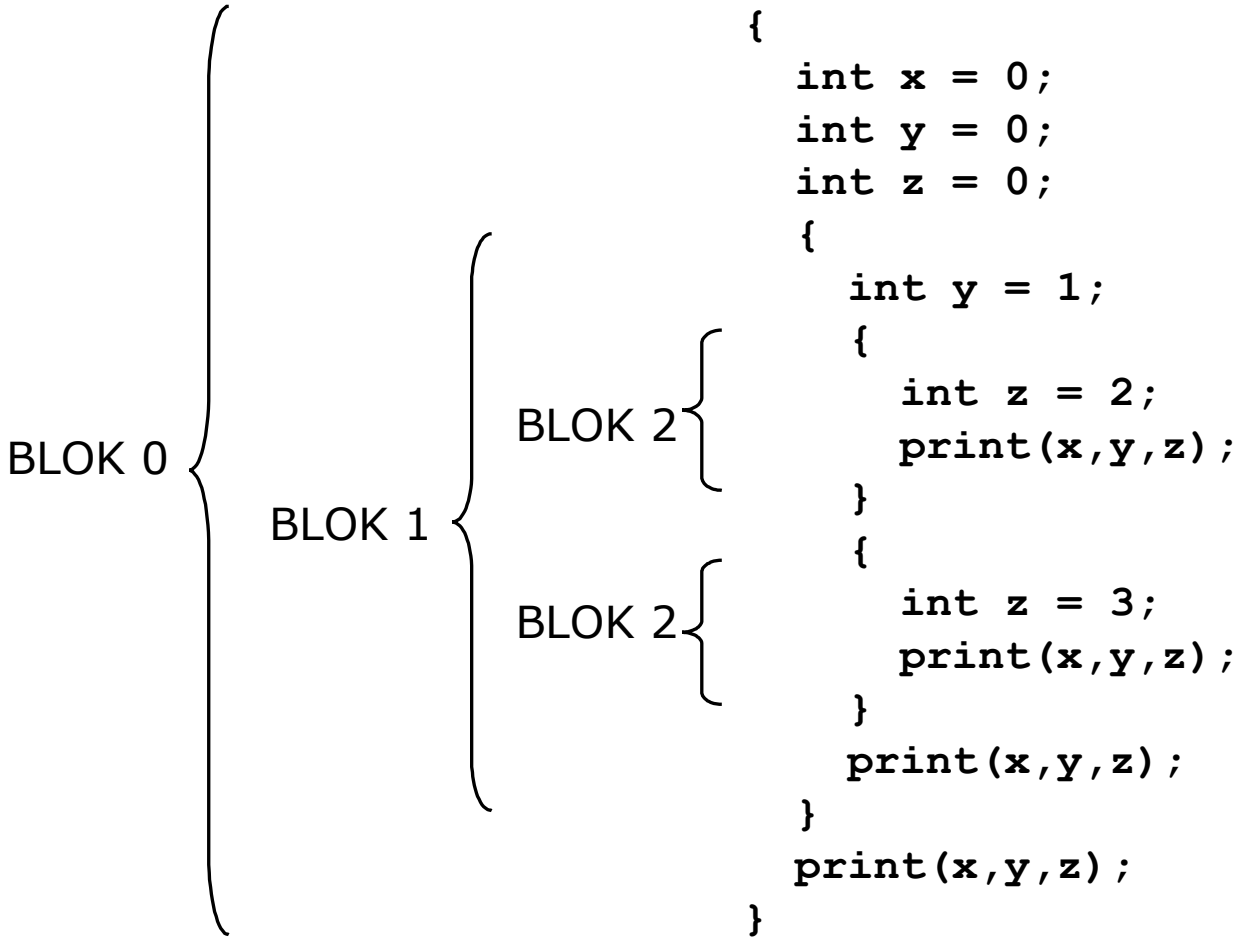
Primer generisanja koda – **`continue`** iskaz

```
while14:
        CMPS    a,$5
        JGES    false14
true14:
if15:
        CMPS    a,$3
        JNE     false15
true15:
        JMP     while14         //continue
        JMP     exit15
false15:
exit15:
        ADDS    a,$1,%0
        MOV     %0,a
        JMP     while14
false14:
```

Generisanje koda – *C* blokovi

- *C* blokovi
  - blokovi se međusobno razlikuju po rednom broju koji im dodeljuje kompajler
  - u tabeli simbola za svaku lokalnu promenljivu bloka mora biti vezan redni broj bloka
  - kompajler koristi redne brojeve blokova kod određivanja područja važenja identifikatora (kada u bloku n naiđe na neku promenljivu, kompajler traži tu promenljivu za blok n, a ako je traženje neuspešno, ono se ponavlja za prethodni blok):

Generisanje koda – *C* blokovi




```
main()
{
  int x = 0;
  int y = 0;
  int z = 0;
  {
    int y = 1;
    {
      int z = 2;
      print(x,y,z);
    }
    {
      int z = 3;
      print(x,y,z);
    }
    print(x,y,z);
  }
  print(x,y,z);
}
```

Generisanje koda – *C* blokovi

- rezultat štampanja

```
0    1    2
0    1    3
0    1    0
0    0    0
```

- Lokalne promenljive blokova se čuvaju u frejmu bloka na steku (za razliku od frejma poziva funkcije, frejm bloka ne sadrži argumente kao ni povratnu vrednost)
- Frejm bloka se stvara na ulazu u blok, a uništava na izlasku iz bloka

Generisanje koda – *Pascal* potprogrami

- Definicija potprograma u potprogramu *(Pascal)*
  - za svaki potprogram je potrebna posebna tabela simbola, koja sadrži njegove lokalne identifikatore
  - tako potencijalno nastaje hijerarhijska organizacija tabela simbola i hijerarhijska organizacija frejmova ovakvih potprograma

Generisanje koda - slogovi

- Slogovi *(C struct, Pascal record)*
  - za svaki slog je potrebna posebna tabela simbola, koja sadrži njegova polja sa relativnom pozicijom u slogu kao dodatnim atributom
  - za slog

```
struct { int x;
         int y; } z
```

je potrebno zauzeti **2** lokacije. Iskazu

```
z.y = z.x;
```

odgovara kod

```
MOVA   z,%0
MOV    0(%0),4(%0)
```

(podrazumeva se da je radni registar `%0` slobodan i da je u njega smeštena početna adresa sloga **z**)

Generisanje koda - nizovi

- Nizovi
  - za svaki niz u tabeli simbola treba registrovati i broj njegovih elemenata
  - za niz

    ```
    int n[10];
    ```

    je potrebno zauzeti `10` lokacija. Iskazu

    ```
    n[0] = n[1];
    ```

    odgovara kod

    ```
    MOVA  n,%0
    MOV   4(%0),0(%0)
    ```

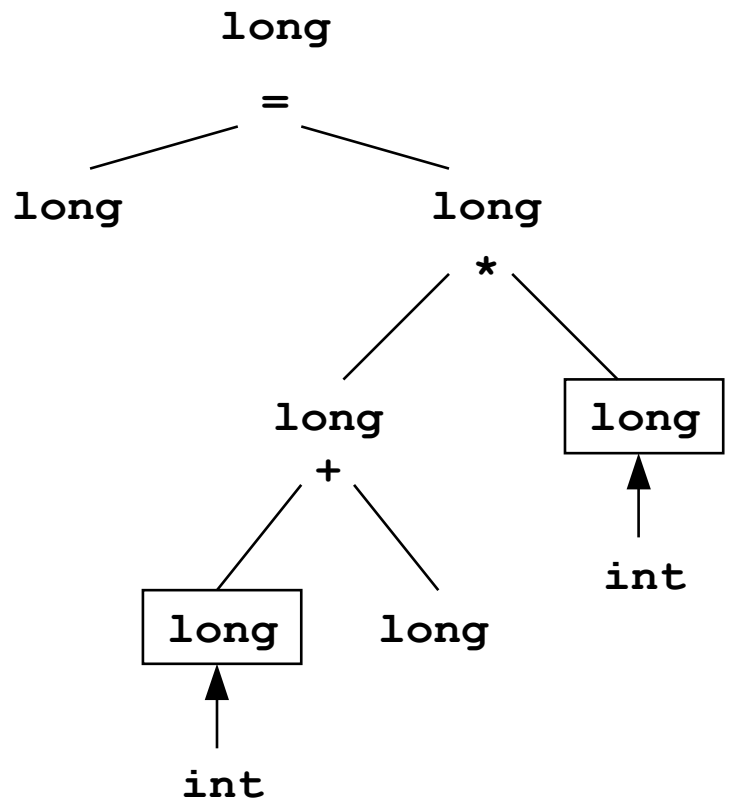
    (podrazumeva se da su radni registar `%0` slobodan i da je u njega smeštena početna adresa niza `n`)

Generisanje koda – konverzije tipova

- Konverzije tipova u izrazima
  - u izrazima sa promenljivim i konstantama raznih ali kompatibilnih tipova (**`int`** i **`long`** ili **`int`** i **`float`**) potrebno je obaviti podrazumevajuće konverzije tipova (**`int`** u **`long`** ili **`int`** u **`float`**) pre korišćenja vrednosti promenljivih ili konstanti
  - konverzije tipova se zasnivaju na određivanju tipova izraza i iskaza. Za određivanje tipova izraza (iskaza) potrebno je izraze (iskaze) predstaviti u obliku grafa čiji čvorovi sadrže tipove podizraza (podiskaza).
  - Konverzija se obavlja (ako je moguća) kada se prepozna binarni izraz sa operandima raznih ali kompatibilnih tipova, ili kada se prepozna dodela vrednosti jednog tipa promenljivoj različitog kompatibilnog tipa
  - Radi konverzije kompajler generiše odgovarajuću naredbu za konverziju

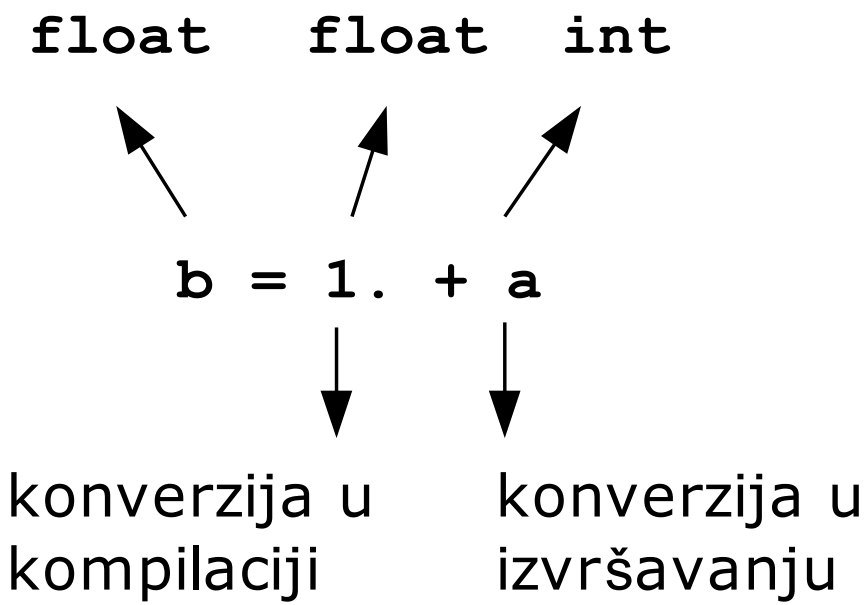
Generisanje koda – konverzije tipova

```
long = (int + long) * int
```



strelica označava konverziju

Generisanje koda – konverzije tipova



Generisanje koda – vrednost logičkog izraza

- Pridruživanje vrednosti logičkih izraza
  - iskazu

```
bool = a > b && c > d || a > c && b > d || a > e;
```

(podrazumeva se da su sve promenljive globalne, celobrojne i označene) odgovara kod

Generisanje koda – vrednost logičkog izraza

```
                CMPS    a,b
                JLES    false0
                CMPS    c,d
                JGTS    true0
false0:
                CMPS    a,c
                JLES    false1
                CMPS    b,d
                JGTS    true0
false1:
                CMPS    a,e
                JLES    false2
true0:
                MOV     $1,bool
                JMP     next0
false2:
                MOV     $0,bool
next0:
```

(podrazumeva se da su brojevi uz labele jedinstveni)

Generisanje koda – **`switch`** iskaz

- **`switch`** iskaz
  - iskazu

```
switch (state)
{
  case 10 : state = 1;
            break;
  case 20 : state = 2;
            break;
  default : state = 0;
}
```

(podrazumeva se da je **`state`** globalna celobrojna označena promenljiva) odgovara kod

Generisanje koda – **`switch`** iskaz

```
                CMPS    state,10
                JEQ     case0_0
                CMPS    state,20
                JEQ     case0_1
                JMP     default0
case0_0;
                MOV     $1,state
                JMP     next0
case0_1:
                MOV     $2,state
                JMP     next0
default0:
                MOV     $0,state
next0:
```

(podrazumeva se da je prvi broj `0` uz labele jedinstven)

- da su izostavljeni **`break`** iskazi, bile bi izostavljene i prve dve naredbe **`JMP next0`**

Generisanje koda – `for` iskaz

- `for` iskaz
  - iskazima

```
suma = 0;
for (i = 0; i <= 5; i++)
  suma = suma + i;
```

(podrazumeva se da su `suma` i `i` globalne, celobrojne i označene promenljive) odgovara kod

Generisanje koda – `for` iskaz

```
                MOV   $0,suma
                MOV   $0,i
for0:
                CMPS  i,$5
                JGTS  next0
                ADDS  suma,i,%0
                MOV   %0,suma
                ADDS  i,$1,%0
                MOV   %0,i
                JMP   for0
next0:
```

(podrazumeva se da su brojevi uz labele jedinstveni)

MICKO (MIkro C KOmpajler)

- putanja: **`examples/microC/micko/`**

- definicije konstanti (**`defs.h`**)
- Lex specifikacija (**`scanner.l`**)
- Yacc specifikacija (**`parser.y`**)
- definicija i **implementacija** tabele simbola (**`symtab.h`**, **`symtab.c`**)
- definicija i **implementacija** steka (**`stack.h`**, **`stack.c`**)
- funkcije za semantičku analizu (**`semantic.h`**, **`semantic.c`**)
- funkcije za generisanje koda (**`codegen.h`**, **`codegen.c`**)

Osvrt na gramatike

- Gramatika definiše jezik kao skup svih nizova simbola koji mogu biti generisani njenom primenom
- Podela gramatika se zasniva na osobinama pravila gramatika koja imaju oblik:

  `leva_sekvenca -> desna_sekvenca`

  (i **`leva_sekvenca`** i **`desna_sekvenca`** u principu sadrže i pojmove i simbole)

Osvrt na gramatike

- tip0 - **opšte gramatike**
  - **`desna_sekvenca`** pravila može biti prazna

Osvrt na gramatike

- tip1 - **kontekstno zavisne gramatike** (*non-context-free grammars*)
  - poštuju ograničenje da je dužina (ili broj pojmova i simbola iz) **`leva_sekvenca`** pravila manja od dužine ili jednaka dužini **`desna_sekvenca`** pravila
  - kod ovih gramatika na izvođenja iz nekog pojma može uticati kontekst koji formira **`leva_sekvenca`** pravila – znači isti pojam može imati razna izvođenja za razne kontekste

Osvrt na gramatike

- tip2 - **kontekstno nezavisne gramatike** (*context-free grammars*)
  - poštuju ograničenje da **`leva_sekvenca`** pravila sadrži samo jedan pojam, pa tako nema konteksta koji može uticati na izvođenja iz datog pojma
  - ovakve gramatike ne mogu da izraze na primer zahtev da identifikatori moraju biti deklarisani pre korišćenja, jer to podrazumeva da tretiranje identifikatora zavisi od konteksta

Osvrt na gramatike

- tip3 - **regularne gramatike** (*regular grammars*)
  - poštuju ograničenje da **`leva_sekvenca`** pravila sadrži samo jedan pojam, a da **`desna_sekvenca`** pravila sadrži najviše ili jedan simbol ili jedan pojam i jedan simbol (s tim da pojam uvek ili prethodi simbolu ili sledi iza njega)
  - regularne gramatike ne mogu opisati balansirane ili umetnute nizove poput nizova parova levih i desnih zagrada
  - za svaki regularni izraz postoji ekvivalentna regularna gramatika koja definiše isti jezik kao i pomenuti regularni izraz

Osvrt na gramatike

- Regularni izrazi su uvedeni radi leksičke analize
  - jer se pomoću njih lako i koncizno izražavaju leksička pravila i
  - jer se iz njih automatski generišu skeneri efikasnije nego iz proizvoljne gramatike

Osvrt na gramatike

- Automatsko generisanje skenera se zasniva na konvertovanju regularnih izraza u konačne automate.
- Pored determinističkih postoje i **nedeterministički konačni automati**. Kod njih iz nekog čvora može izlaziti više spojnica labeliranih istim znakovima. Zahvaljujući tome, nedeterministički automati imaju manje čvorova od determinističkih, ali po cenu da ne mogu među spojnicama koje su labelirane istim znakovima da unapred odaberu spojnicu koja vodi ka prepoznavanju datog simbola. U ovim okolnostima neuspeh u prepoznavanju simbola ne znači obavezno grešku, nego podrazumeva **vraćanje skeniranja u natrag** (*backtracking*) radi pokušaja sa nekom od preostalih alternativnih spojnica. Zbog toga su nedeterministički automati u proseku sporiji od determinističkih.

Osvrt na gramatike

- Kontekstno nezavisne gramatike su uvedene radi sintaksne analize
- Iz skupa kontekstno nezavisnih gramatika izdvajaju se dva podskupa:
  - LL(1) gramatike koje su prilagođene silaznom parsiranju i
  - LR(1) gramatike koje su prilagođene uzlaznom parsiranju

Problemi silaznog parsiranja

- Problemi primene silaznog parsiranja na μC gramatiku

LL(1) gramatike

- Silazno parsiranje proverava da li je ulazni niz simbola u skladu sa datom gramatikom tako što proverava da li gramatika dozvoljava
  - da prvi simbol iz ulaznog niza bude na prvom mestu, a
  - da drugi simbol iz ulaznog niza sledi iza prvog simbola itd.
- Postupak provere se pojednostavljuje ako se za svaki pojam gramatike uvede odgovarajući **sintaksni potprogram** koji proverava da li je **posmatrani simbol** (iz ulaznog niza) prihvatljiv sa stanovišta dotičnog pojma
- Podrazumeva se da sintaksni potprogram jednog pojma poziva sintaksne potprograme drugih pojmova da bi zajedno proverili ispravnost ulaznog niza simbola
- Zahvaljujući saradnji sintaksnih potprograma, silazno parsiranje započinje pozivom sintaksog potprograma polaznog pojma. Iz njega se pozivaju sintaksni potprogrami pojmova koji se pominju sa desne strane pravila polaznog pojma itd.

LL(1) gramatike

- Ovakvo parsiranje se naziva **rekurzivno spuštanje** (*recursive-descent parsing*), jer pozivi sintaksnih potprograma dovode do spuštanja niz drvo parsiranja. Pri tome su neizbežni rekurzivni pozivi sintaksnih potprograma zbog rekurzivne prirode konekstno nezavisnih gramatika. Na primer, sintaksni potprogram pojma `statement` poziva sintaksni potprogram pojma `compound_statement`, a on, posredstvom sintaksnog potprograma pojma `statement_list` poziva sintaksni potprogram pojma `statement`.
- Za rekurzivno spuštanje je zgodno da se na osnovu posmatranog simbola uvek jednoznačno može odrediti sintaksni potprogram koji se sledeći poziva. Ovakvo rekurzivno spuštanje se naziva **predvidivim** (*predictive recursive-descent parsing*).

LL(1) gramatike

- Kod predvidivog rekurzivnog spuštanja ne dolazi do vraćanja parsiranja unatrag (*backtracking*), jer je u svakom sintaksnom potprogramu (na svakom koraku parsiranja) unapred određeno koji simboli su prihvatljivi, a za svaki od njih je obezbeđeno da jednoznačno usmerava nastavak parsiranja
- Predvidivo rekurzivno spuštanje zahteva posebne gramatike
- Jedno svojstvo ovakvih gramatika je da se međusobno razlikuju svi početni simboli, kojima započinju razni nizovi simbola, izvedeni primenom raznih (alternativnih) pravila nekog pojma. Ovakvi početni simboli dotičnog pojma obrazuju njegov **početni skup simbola** (*FIRST*).
- Na primer, početni skup simbola pojma *type*

  type → "int" | "unsigned"

  sačinjavaju simboli **int** i **unsigned**.

LL(1) gramatike

- Jedinstvenost simbola iz početnog skupa simbola nekog pojma obezbeđuje jednoznačno usmeravanje predvidivog rekurzivnog spuštanja
- Kod određivanja početnog skupa simbola nije dovoljno konsultovati samo desne strane pravila koja opisuju izvođenja iz datog pojma. Ako desne strane ovih pravila započinju novim pojmovima, tada se moraju konsultovati i desne strane pravila koja opisuju izvođenja iz novih pojmova i tako redom.
- Na primer, početni skup simbola pojma *variable*:

$$\textit{variable} \rightarrow \textit{type}\ \texttt{identifier}$$

sačinjavaju simboli **int** i **unsigned**.

LL(1) gramatike

- Iz ograničenja da simboli iz početnog skupa simbola nekog pojma moraju biti jedinstveni sledi da leva rekurzija u pravilima i identični počeci desnih strana alternativnih pravila istog pojma nisu prihvatljivi, jer narušavaju jedinstvenost simbola iz početnog skupa simbola nekog pojma.
- Levu rekurziju sadrži pravilo pojma *variable_list*:

variable_list → *variable* “;”

| *variable_list variable* “;”

Ako je “**int**” posmatrani simbol, tada on u sintaksnom potprogramu prethodnog pojma ne upućuje jednoznačno na izbor jedne od dve alternative (jer su početni simboli ovih alternativa identični). Međutim, takva dilema nestaje ako se pravila pojma *variable_list* preformulišu (*elimination of left recursion*):

LL(1) gramatike

variable_list → *variable* “;” *variable_list1*

variable_list1 → *variable* “;” *variable_list1*

| ε

jer je, nakon uvođenja novog pojma *variable_list1*, preostala samo jedna alternativa u pravilu pojma *variable_list*

- Identične početke desnih strana sadrže pravila pojma *if*:

if → “**if**” “**(**” *log_exp* “**)**” *statement*

if → “**if**” “**(**” *log_exp* “**)**” *statement* “**else**” *statement*

LL(1) gramatike

- Ako je "**if**" posmatrani simbol u sintaksnom pravilu prethodnog pojma, tada on ne upućuje jednoznačno na izbor jedne od dve alternative (jer su početni simboli ovih alternativa identični). Međutim, takva dilema nestaje, ako se pravila pojma *if* preformulišu (*left factoring*):

if → "**if**" "**(**" *log_exp* "**)**" *statement else*

else → "**else**" *statement*

| ε

jer je, nakon uvođenja novog pojma *else*, preostala samo jedna alternativa u pravilu pojma *if*.

LL(1) gramatike

- Pored početnog skupa simbola, za svaki pojam je važan i **sledbenički skup simbola** (*FOLLOW*). Njega obrazuju simbol kraja datoteke i simboli koji mogu da slede iza niza simbola izvedenog iz dotičnog pojma
- Sledbenički skup olakšava oporavak od grešaka u toku kompilacije, jer omogućuje da se, nakon otkrivanja pogrešnog simbola u okviru sintaksnog potprograma nekog pojma, ignorišu simboli koji se u ulaznom nizu nalaze između pogrešnog simbola i simbola koji pripada sledbeničkom skupu ovog pojma *(panic mode error-recovery)*. Kada se koristi na prethodni način, sledbenički skup se naziva i **sinhronizacioni skup**, a njegovi elementi **sinhronizacioni simboli**. Oni omogućuju nastavak kompilacije nakon otkrivanja greške s ciljem da se u jednoj kompilaciji otkrije što više grešaka.

LL(1) gramatike

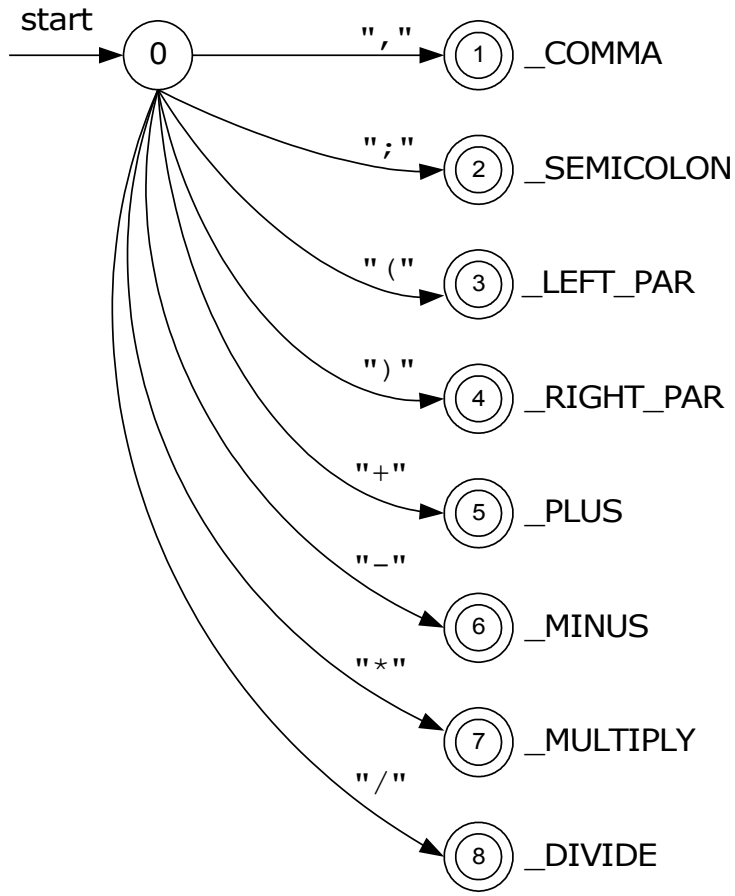
- LL(1) gramatike (***L****eft –to-right scanning of the input, producing a* ***L****eftmost derivation, and using* ***1*** *input symbol of lookahead at each step to make parsing action decisions*) poštuju ograničenje da početni skupovi simbola njenih raznih pojmova ne sadrže iste simbole.
- Zato LL(1) gramatike:
  - nemaju levih rekurzija i
  - nisu dvosmislene.
- LL(1) gramatike su podesne za ručno pravljenje kompajlera, primenom predvidivog rekurzivnog spuštanja.

Primer LL(1) gramatike – *µPascal* gramatika

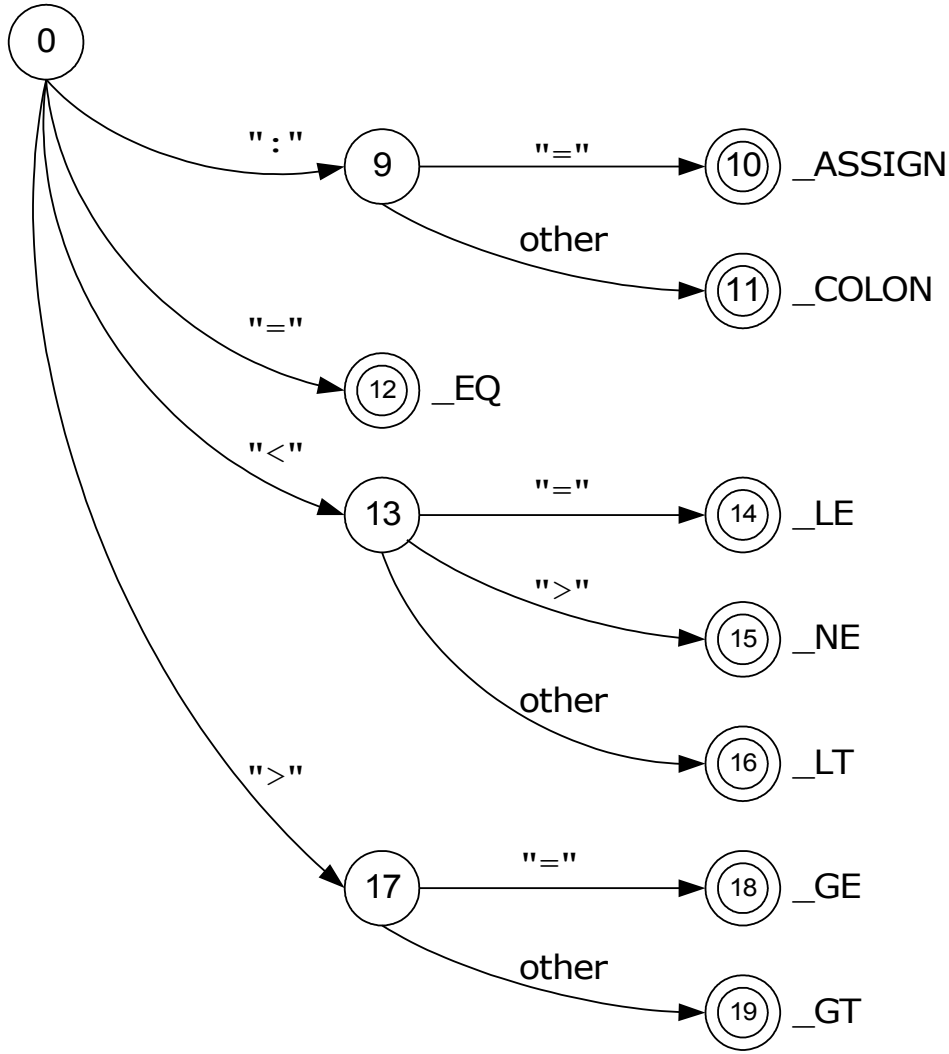
- *µPascal gramatika*
  (**grammars/MicroPascalGrammar.pdf**)

Primer LL(1) gramatike – *µPascal* gramatika

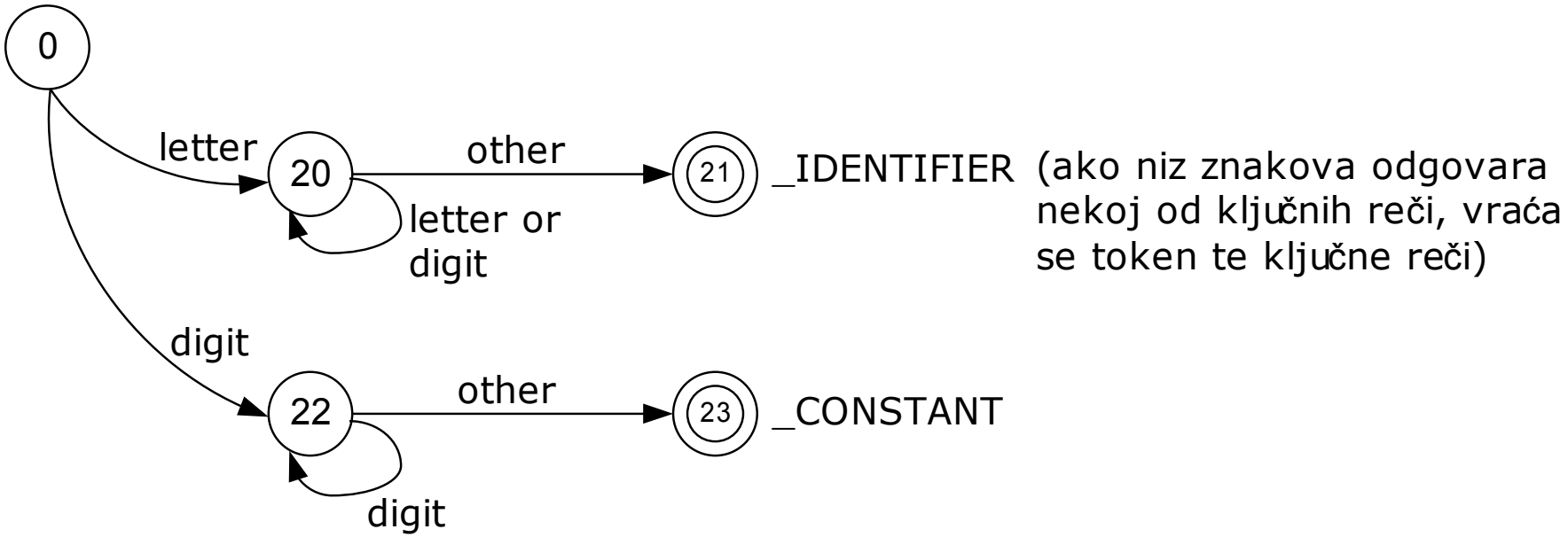
- Dijagrami prelaza za gramatiku simbola, izvedenu iz *µPascal* gramatike:



Primer LL(1) gramatike – *µPascal* gramatika



Primer LL(1) gramatike – *µPascal* gramatika



LL(1) gramatike

- Dijagram prelaza se može izraziti programskim jezikom *C* pomoću funkcije koja sadrži **`switch`** iskaz i čiji slučajevi (**`case`**) odgovaraju pojedinim stanjima skenera. U okviru pojedinih od ovih slučajeva posebni **`if`** iskazi opisuju prelaske u nova stanja. Pri tome njihove alternative (grane) opisuju akcije skenera na pojavu znakova iz skupova znakova koji labeliraju usmerene spojnice. Ovakva *C* funkcija predstavlja skener za konkretnu gramatiku simbola.

Primer – skener *μPascal* gramatike

- Funkcija koja opisuje skener:

```
unsigned long scanner(void);
```

  - vodi računa o broju linije (promenljiva **`line`**)
  - u slučaju greške štampa "*LEXICAL ERROR on character*" i pogrešan znak, vraća se u stanje **`0`** i pokušava nastavak skeniranja
  - u slučaju kraja datoteke vraća token **`_EOF`**

Primer – skener *μPascal* gramatike

- definicija konstanti
  (**`examples/microP/defs.h`**)
- *μPascal skener*
  (**`examples/microP/scanner.c`**)

Primer – parser *μPascal* gramatike

- Ručno napisani parser koji realizuje predvidivo rekurzivno spuštanje za *μPascal* gramatiku

1. Primer rada parsera *μPascal* gramatike

```
program absolute

proc abs(x : int)
    begin
        if x < 0
        then abs := -x
        else abs := x
    end

begin
    abs(-8)
end
```

1. Primer rada parsera *μPascal* gramatike

```
START

PROGRAM in line 1
IDENTIFIER in line 1
PROC in line 3
IDENTIFIER in line 3
LEFT_PAR in line 3
IDENTIFIER in line 3
COLON in line 3
INT in line 3
VARIABLE
VARIABLE_LIST
RIGHT_PAR in line 3
BEGIN in line 4
IF in line 5
IDENTIFIER in line 5
OPERAND
EXPRESSION
LT in line 5
CONSTANT in line 5
OPERAND
EXPRESSION
CONDITION
THEN in line 6
IDENTIFIER in line 7
ASSIGN in line 7
MINUS in line 7
IDENTIFIER in line 7
OPERAND
EXPRESSION
ASSIGNMENT_STATEMENT
STATEMENT
ELSE in line 8
IDENTIFIER in line 9
ASSIGN in line 9
IDENTIFIER in line 9
OPERAND
EXPRESSION
ASSIGNMENT_STATEMENT
STATEMENT
IF_STATEMENT
STATEMENT
STATEMENT_LIST
END in line 10
PROCEDURE
PROCEDURE_LIST
BEGIN in line 12
IDENTIFIER in line 13
LEFT_PAR in line 13
MINUS in line 13
CONSTANT in line 13
OPERAND
EXPRESSION
RIGHT_PAR in line 13
PROCEDURE_CALL_STATEMENT
STATEMENT
STATEMENT_LIST
END in line 14
PROGRAM

STOP
```

2. Primer rada parsera *μPascal* gramatike

```
program absolute
var
proc abs(x := int)
    begin
        if x < 0
        then abs := -x
        else abs = x
    end

begi
    abs(-8)
end
```

2. Primer rada parsera *μPascal* gramatike

```
START

PROGRAM in line 1
IDENTIFIER in line 1
VAR in line 2
PARSING ERROR - EXPECTED
     IDENTIFIER in line
     3
VARIABLE
VARIABLE_LIST
PROC in line 3
IDENTIFIER in line 3
LEFT_PAR in line 3
IDENTIFIER in line 3
PARSING ERROR - EXPECTED
     COLON in line 3
SKIPED ASSIGN in line 3
INT in line 3
VARIABLE
VARIABLE_LIST
RIGHT_PAR in line 3
BEGIN in line 4
IF in line 5
IDENTIFIER in line 5
OPERAND
EXPRESSION
LT in line 5
CONSTANT in line 5
OPERAND
EXPRESSION
CONDITION
THEN in line 6
IDENTIFIER in line 7
ASSIGN in line 7
MINUS in line 7
IDENTIFIER in line 7
OPERAND
EXPRESSION
ASSIGNMENT_STATEMENT
STATEMENT
ELSE in line 8
IDENTIFIER in line 9
PARSING ERROR - EXPECTED
          := or ( in line 9
SKIPED EQ in line 9
SKIPED IDENTIFIER in line 9
STATEMENT
IF_STATEMENT
STATEMENT
STATEMENT_LIST
END in line 10
PROCEDURE
PROCEDURE_LIST
PARSING ERROR - EXPECTED
BEGIN in line 12
SKIPED IDENTIFIER in line 12
SKIPED IDENTIFIER in line 13
SKIPED LEFT_PAR in line 13
SKIPED MINUS in line 13
SKIPED CONSTANT in line 13
SKIPED RIGHT_PAR in line 13
SKIPED END in line 14
PARSING ERROR - EXPECTED
STATEMENT
STATEMENT
STATEMENT_LIST
PROGRAM

STOP
```

Primer – parser *µPascal* gramatike

- *µPascal* parser
  (**`examples/microP/parser.c`**)

LR(1) gramatike

- LR(1): "***L****eft-to right scanning of the input, for constructing a* ***R****ightmost derivation in reverse, and using* ***1*** *input symbol of lookahead for making parsing decision*"
- LR parseri
  - mogu prepoznati praktično sve jezike opisane pomoću kontekstno nezavisnih gramatika
    - pokrivaju veći broj gramatika od LL parsera (generalniji su od LL parsera)
  - imaju efikasnu implementaciju
    - koriste najgeneralniji *"non backtracking shift-reduce parsing"* metod
    - otkrivaju sintaksne greške u najmanjem mogućem broju koraka
  - nisu podesni za ručno pravljenje, ali mogu biti automatski generisani pomoću generatora LR parsera

LR(1) gramatike

- Generatori LR parsera mogu da ukažu na dvosmislenosti gramatike i njena problematična pravila sa stanovišta LR parsiranja
- Glavni problem za generatore LR parsera je pravljenje tabele akcija i prelaza
- Za pravljenje tabele akcija i prelaza potrebno je (1) uvesti stanja koja odgovaraju mogućim situacijama na steku LR parsera i (2) za svako od njih odrediti akciju na pojavu nekog od simbola iz ulaznog niza. Akcija je ili izbacivanje ovog simbola iz ulaznog niza simbola (*shift*) uz smeštanje zadanog stanja na stek ili redukovanje (*reduce*) stanja na steku primenom zadanog pravila. Nakon redukcije potrebno je na osnovu sadržaja steka odabrati novo stanje.

LR(1) gramatike

- Za pravila

  (1) $E \rightarrow E$ "+" **id**

  (2) $E \rightarrow$ **id**

  (**id** je simbol identifikatora, kojeg prepoznaje skener)

  mogući izgled tabele akcija i prelaza je:

STANJA	AKCIJE I PRELAZI ZA ULAZNE SIMBOLE			PRIMER SADRŽAJA STEKA (ZA ULAZ **id + id**)
	id	**+**	**eof**	
0	*shift* i novo stanje 1	greška	greška	0 (stek prazan)
1	greška	*reduce* po pravilu 2 i novo stanje 2	*reduce* po pravilu 2 i novo stanje 2	0 1 (**id**)
2	greška	*shift* i novo stanje 3	ulazni niz simbola je prihvaćen	0 2 (*E*)
3	*shift* i novo stanje 4	greška	greška	0 2 3 (*E* +)
4	greška	*reduce* po pravilu 1 i novo stanje 2	*reduce* po pravilu 1 i novo stanje 2	0 2 3 4 (*E* + **id**)

LR(1) gramatike

- Za netrivijalne gramatike ručno pravljenje tabele akcija i prelaza je komplikovano, pa se obavlja automatski po nekom od sledećih algoritama:
  - *Simple LR* (SLR) – lak za implementaciju, ali primenljiv za najmanji broj LR gramatika
  - *Canonical LR* – najteži za implementaciju, ali primenljiv za najveći broj LR gramatika
  - *Look-ahead LR* (LALR) – srednji po težini implementacije i mogućnostima primene – koristi se u Yacc-u

Osvrt na kompajlere

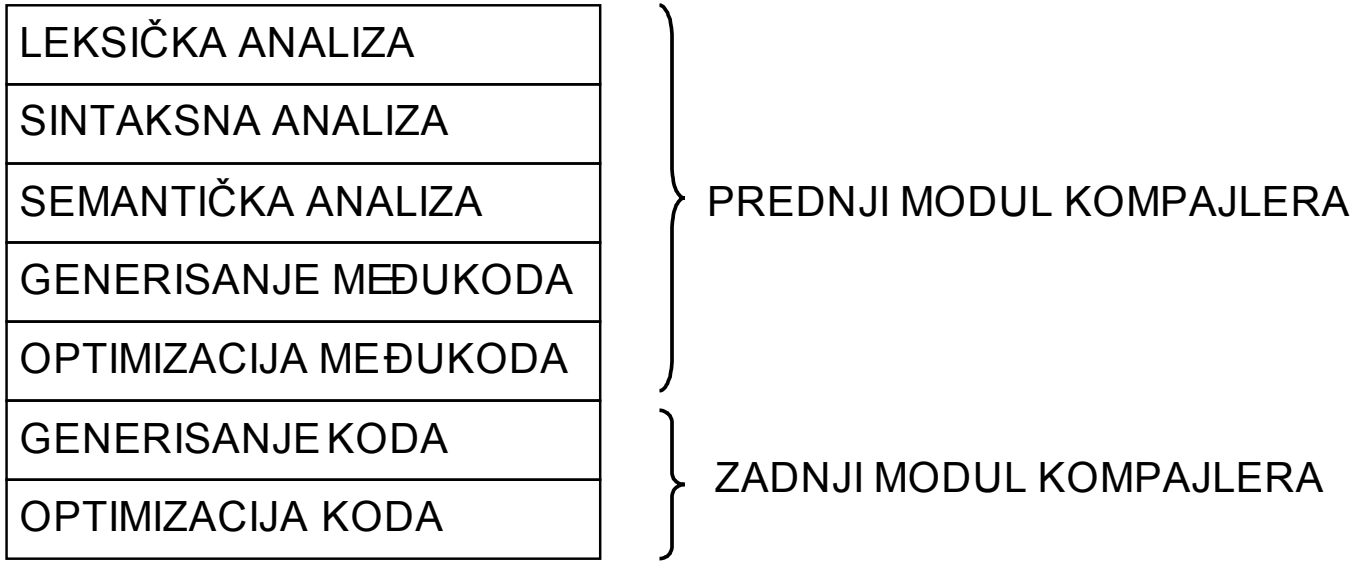
- **Faze u kompilaciji**
  - leksička analiza
  - sintaksna analiza
  - semantička analiza
  - generisanje koda
  - optimizacija
- Kompilacione faze se grupišu na faze zavisne od izvornog jezika i na faze zavisne od ciljnog jezika:
  - faze zavisne od izvornog jezika (analiza) pripadaju **prednjem modulu kompajlera** *(front end)*
  - faze zavisne od ciljnog jezika (sinteza) pripadaju **zadnjem modulu kompajlera** *(back end)*

Osvrt na kompajlere

- Praktična važnost podele kompajlera na prednji i zadnji modul proizlazi iz činjenice da se prevođenje sa jednog izvornog jezika na više ciljnih jezika može ostvariti pravljenjem jednog prednjeg modula i više zadnjih modula kompajlera, a da se prevođenje sa više izvornih jezika na jedan ciljni jezik može ostvariti pravljenjem više prednjih i jednog zadnjeg modula
- Podela kompajlera na prednji i zadnji modul je važna za olakšavanje prilagođavanja kompajlera novim zahtevima i novim okolnostima
- Komunikacija prednjeg i zadnjeg modula kompajlera podrazumeva uvođenje **međukoda** *(intermediate code)*. On predstavlja (1) ciljni jezik za prednji modul kompajlera, a (2) izvorni jezik za zadnji modul kompajlera.
- Međukod je jezik hipotetskog računara (apstraktne mašine)

Osvrt na kompajlere

- Uvođenje međukoda dovodi do pojave faze generisanja međukoda i podele faze optimizacije na dve zasebne faze
- Pregled faza u kompilaciji:



- Za međukod je važno da olakša implementaciju faza generisanja međukoda i koda, kao i faze optimizacije

Osvrt na kompajlere

- Kompajler može sadržati samo prednji modul. Tada je njegov ciljni jezik međukod, koga izvršava (interpretira) poseban program – **interpreter**.

Osvrt na kompajlere: upotreba drveta parsiranja

- Drvo parsiranja sadrži reprezentaciju kompletnog i ispravnog (validnog) programa u izvornom jeziku
- Drvo parsiranja omogućuje:
  - prevođenje
  - interpretiranje
  - primenu procesa suprotnog parsiranju - *unparse*, i prikazivanje korisniku raznih osobina programa (u obliku običnog teksta, u obliku *XML*-a, u nekom grafičkom obliku, ...)
  - proveru da li su sve promenljive inicijalizovane pre upotrebe (samo neki jezici ovo definišu kao deo semantičkih pravila, ali mnogi ne)
  - otkrivanje raznih osobina programa
  - ...

Osvrt na kompajlere

- U kompilaciji mogu da se izdvoje **prolazi** – to su delovi kompilacije koji obuhvataju čitanje ulazne datoteke (na primer sa izvornim programom) i pisanje izlazne datoteke (na primer sa ciljnim programom)

Osvrt na kompajlere

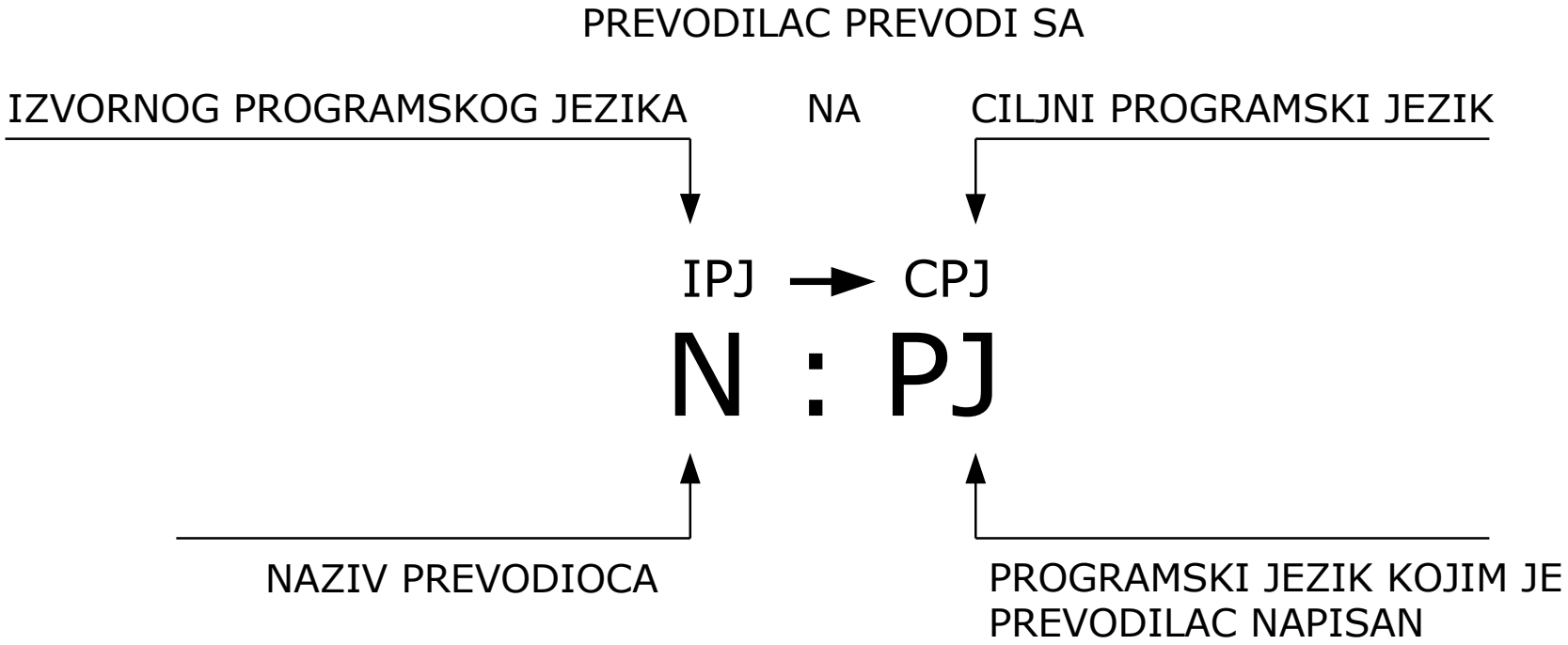
- Broj faza koje obuhvata jedan prolaz se menja od kompajlera do kompajlera
  - jednoprolazni kompajler – sve faze u jednom prolazu (karakterističan za neke *Pascal* kompajlere. Njihov ciljni jezik je međukod, nazvan *P-code* zbog svojstva portabilnosti sa računara na računar. Portabilnost P-koda se zasniva na pravljenju interpretera P-koda za razne računare)
  - dvoprolazni kompajler – prvi prolaz odgovara prednjem modulu kompajlera, a drugi prolaz odgovara zadnjem modulu kompajlera (karakterističan za *C* kompajlere)
- Za jednoprolazne kompajlere, kao i za prednji modul dvoprolaznih kompajlera, je karakteristično da je parser centralni potprogram kompajlera koji obavlja fazu sintaksne analize i poziva ostale potprograme, zadužene za preostale faze kompilacije

Osvrt na kompajlere

- Osobine kompajlera
  - brzina prevođenja (manje prolaza – veća brzina prevođenja)
  - kvalitet generisanog koda (brzina izvršavanja i memorijski zahtevi)
  - postupak sa greškama
  - prenosivost (na novi ciljni jezik – *retargetability,* i na novi računar – *rehostability*)
  - lakoća održavanja

Osvrt na kompajlere

- Kompajler karakterišu ne samo njegov izvorni i ciljni jezik, nego i njegov implementacioni jezik
- Oznaka kompajlera (programskog prevodioca):

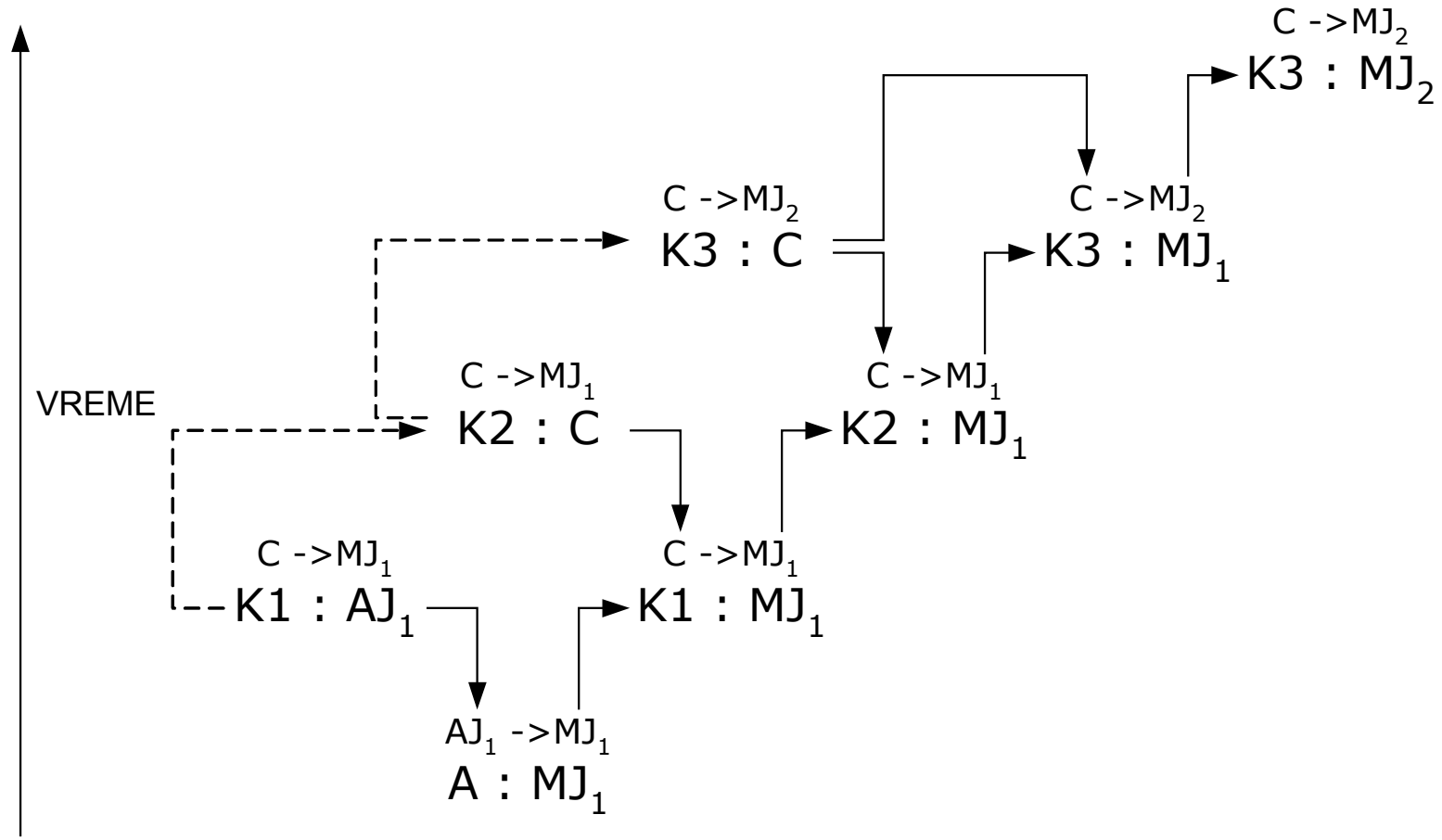


Osvrt na kompajlere

- Kompajleri čiji ciljni jezik ne pripada računaru na kome se oni izvršavaju se nazivaju **kros-kompajleri** *(cross-compiler).*
- Pravljenje kompajlera se značajno olakšava ako se oslanja na postojeće kompajlere i kros-kompajlere

Osvrt na kompajlere

- Primer hronologije razvoja kompajlera (AJ_n – asemblerski jezik "n", MJ_m – mašinski jezik "m"):



Osvrt na kompajlere

- Postupak u kome kompajler kompilira “sam sebe” ili neku svoju verziju ili drugi kompajler *(bootstraping)* je uobičajen u toku pravljenja kompajlera za novi računar ili za usavršavanje kompajlera za isti računar

Interpreter

- Program koji oponaša hipotetski računar kome odgovara međukod
- Realizacija interpretera podrazumeva postojanje struktura podataka koje opisuju *run-time* stanje programa:
  - vrednosti koje se pojavljuju u programu
  - *activation record* (ekvivalent stek frejmu) za svaku pozvanu funkciju
    - veze između lokalnih promenljivih
    - pokazivač na pozivajući *activation record* (*dynamic link*)
    - pokazivač na *lexically-enclosing activation record* (*static link*)

Interpreter

- Hipotetski računar koga oponaša interpreter je obično računar sa stek arhitekturom
- Primer hipotetskog računara: *JVM* (*Java Virtual Machine*)
- Posmatraće se podskup *JVM* označen skraćenicom *IJVM* (*integer JVM*)

IJVM memorijski model

- Memorija *IJVM* se može posmatrati kao niz od 4GB ili kao niz od 1GB reči, gde svaka reč sadrži 4 bajta
- *Java Virtual Machine* ne adresira direktno, već preko nekoliko implicitnih adresa (pokazivača). *IJVM* naredbe mogu pristupiti memoriji samo indeksiranjem preko ovih pokazivača.

IJVM memorijski model

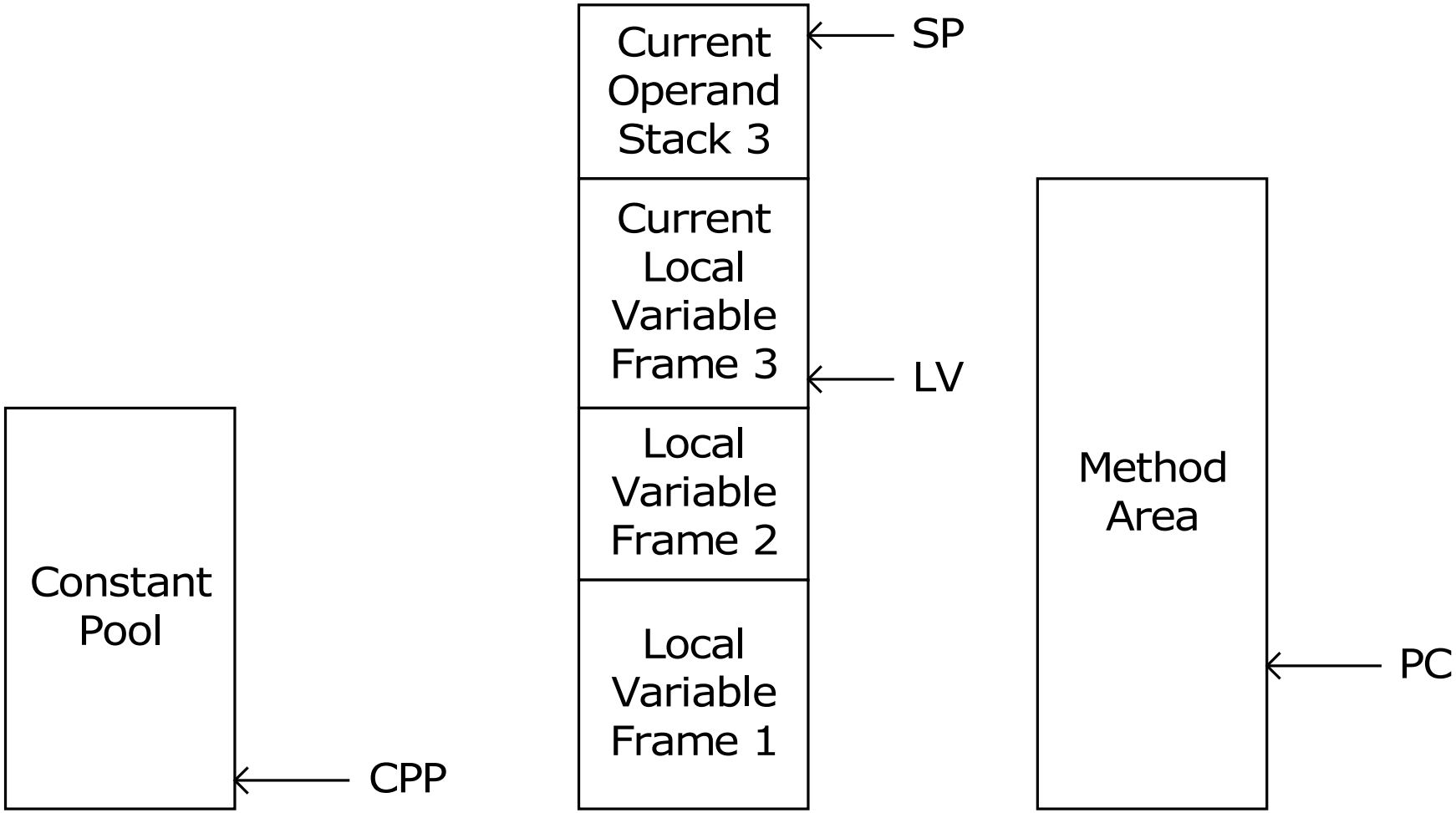
- Memorija *IJVM* je podeljena u 4 dela:
  1. *Constant Pool*. *IJVM* program ne može da ga menja. Ovaj deo memorije sadrži konstante, stringove i pokazivače na druge delove memorije koji mogu biti referencirani. Puni se kada program dospe u memoriju i kasnije se ne može menjati. Postoji implicitni registar *CPP* koji sadrži adresu prve reči iz *constant pool*.
  2. *Local Variable Frame.* Prilikom svakog poziva metode, alocira se prostor za smeštanje promenljivih korišćenih tokom životnog veka poziva. Taj prostor se zove frejm lokalnih promenljivih. Na početku frejma nalaze se parametri (sa argumentima) koji se koriste pri pozivu metode. Frejm lokalnih promenljivih ne uključuje stek operanada. Postoji implicitni registar koji sadrži adresu prve lokacije frejma lokalnih promenljivih. Ovaj registar se zove *LV*.

IJVM memorijski model

3. *Operand Stack.* Java prevodioc garantuje da stek frejm neće preći određenu veličinu, jer veličinu steka izračunava unapred. Prostor za stek operanada se alocira tačno iznad frejma lokalnih promenljivih. Postoji implicitni registar koji sadrži adresu poslednje reči na steku. Za razliku od *CPP* i *LV*, ovaj pokazivač, *SP*, se menja tokom izvršavanja metode jer se operandi smeštaju na stek i skidaju sa njega.
4. *Method Area*. Postoji još i deo memorije koji sadrži program (koji se u *UNIX* procesu zove "*text area*"). Implicitni registar koji sadrži adresu naredbe koja će biti sledeća izvršavana se zove *Program Counter* ili *PC*. Za razliku od ostalih regiona memorije, *Method Area* se tretira kao niz bajtova.

- Treba zapaziti razliku između registara. Registri *CPP*, *LV* i *SP* pokazuju na reči, i predstavljaju ofset po broju reči. Nasuprot tome, *PC* sadrži bajt adresu, pa tako i sva aritmetika nad njim menja adresu za broj bajtova - a ne za broj reči.

IJVM memorijski model



Skup naredbi za *IJVM*

- Skup naredbi za *IJVM* je dat u tabeli
- Svaka naredba se sastoji od koda operacije i ponekad operanda
- Prva kolona prikazuje heksadecimalni kod instrukcije (*byte code*). U drugoj je naveden mnemonik asemblerskog jezika, a treća kolona sadrži kratak opis naredbe.

Skup naredbi za *IJVM*

Hex	Mnemonic	Značenje
0x10	BIPUSH byte	smešta bajt na vrh steka
0x59	DUP	smešta reč sa vrha steka na vrh steka
0xA7	GOTO *offset*	usmerava nastavak izvršavanja na naredbu *offset*
0x60	IADD	skida dve reči sa vrha steka, sabere ih i rezultat smesti na vrh steka
0x7E	IAND	skida dve reči sa vrha steka, napravi njihov Boolean AND i rezultat smesti na vrh steka
0x6C	IDIV	skida dve reči sa vrha steka, podeli ih i rezultat smesti na vrh steka
0x99	IFEQ *offset*	skida reč sa vrha steka i ako je 0 usmerava nastavak izvršavanja na naredbu *offset*
0x9B	IFLT *offset*	skida reč sa vrha steka i ako je manja od 0 usmerava nastavak izvršavanja na naredbu *offset*
0x9F	IF_ICMPEQ *offset*	skida dve reči sa vrha steka i ako su iste usmerava nastavak izvršavanja na naredbu *offset*
0x84	IINC *varnum const*	dodaje konstantu lokalnoj promenljivoj
0x15	ILOAD *varnum*	smešta lokalnu promenljivu na vrh steka

Skup naredbi za *IJVM*

Hex	Mnemonic	Značenje
0x68	IMUL	skida dve reči sa vrha steka, pomnoži ih i rezultat smesti na vrh steka
0xB6	INVOKEVIRTUAL *disp*	poziva metodu
0x80	IOR	skida dve reči sa vrha steka, napravi njihov Boolean OR i rezultat smesti na vrh steka
0xAC	IRETURN	povratak iz metode sa celobrojnom vrednošću
0x36	ISTORE *varnum*	skida reč sa vrha steka i smešta je u lokalnu promenljivu *varnum*
0x64	ISUB	skida dve reči sa vrha steka, oduzme ih i rezultat smesti na vrh steka
0x13	LDC_W *index*	smešta konstantu iz *constant pool* na vrh steka
0x00	NOP	radi ništa
0x57	POP	briše reč sa vrha steka
0x5F	SWAP	zamenjuje dve reči na vrhu steka

Primer

Java program

```
i = j + k;
if (i == 3)
  k = 0;
else
  j = j - 1;
```

odgovarajući Java asemblerski program

```
    ILOAD j          // i = j + k
    ILOAD k
    IADD
    ISTORE i
    ILOAD i          // if (i == 3)
    BIPUSH 3
    IF_ICMPEQ L1
    ILOAD j          // j = j - 1
    BIPUSH 1
    ISUB
    ISTORE j
    GOTO L2
L1: BIPUSH 0         // k = 0
    ISTORE k
L2:
```

Osvrt na kompajlere – vrste međukoda

- Međukod može imati oblik
  - **sintaksnog drveta**
  - **postfiksne** (poljske) **notacije** (primenjena kod *Pascal* kompajlera) ili **prefiksne notacije**
  - **troadresnog koda** koji odgovara hipotetskom asemblerskom jeziku (primenjen kod *C* kompajlera)

Osvrt na kompajlere – vrste međukoda

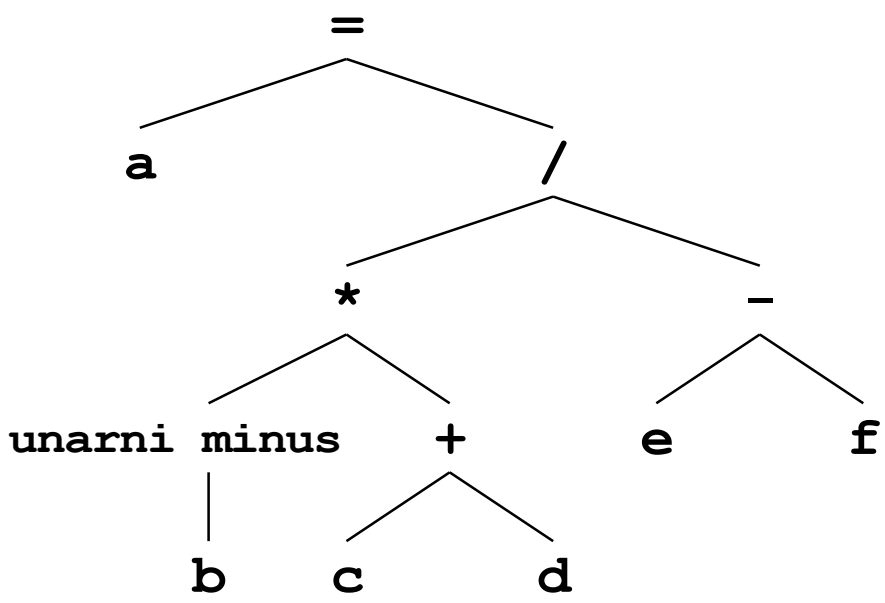
- Sintaksno drvo izražava prirodnu hijerarhijsku strukturu programa pokazujući redosled izvršavanja operacija programa (za razliku od toga drvo parsiranja pokazuje postupak izvođenja programa iz gramatike)

Osvrt na kompajlere – vrste međukoda

- Za iskaz:

```
a = -b * (c + d) / (e - f);
```

Sintaksno drvo izgleda:



Osvrt na kompajlere – vrste međukoda

- Postfiksna notacija je prilagođena apstraktnoj stek mašini, uniformno tretira sve operatore i ne zahteva zagrade
  - za iskaz (koji koristi infiksnu notaciju):

```
a = -b * (c + d) / (e - f);
```

odgovarajuća postfiksna notacija izgleda

```
a b unarni_minus c d + * e f - / =
```

Osvrt na kompajlere – vrste međukoda

- Pomenuti primer postfiksne notacije može biti izražen pomoću prethodno navedenih naredbi:

```
a = -b * (c + d) / (e - f);
```

```
BIPUSH    0
ILOAD     b
ISUB
ILOAD     c
ILOAD     d
IADD
IMUL
ILOAD     e
ILOAD     f
ISUB
IDIV
ISTORE    a
```

Osvrt na kompajlere – vrste međukoda

- Troadresni kod ne mora imati simbolički oblik hipotetskog asemblerskog jezika, nego može imati numerički oblik, gde pojedini brojevi predstavljaju kod naredbe i kodove operanada. Ako kao kodovi operanada služe indeksi elemenata tabele simbola, tada se generatoru koda prepušta da zauzima memoriju za operande. Numerički oblik troadresnog koda je zgodniji za zadnji modul kompajlera.

Generisanje međukoda

- Ranije navedeni primeri generisanja koda su zapravo primeri generisanja međukoda

Optimizacija međukoda

- Pre optimizacije međukoda, koju vrši kompajler, moguća je optimizacija izvornog koda, koja je u nadležnosti programera
  - za ovu optimizaciju neophodno je analizirati izvorni kod
  - u tome pomažu posebni programi **profajleri** *(profiler)* koji prikupljaju podatke o ponašanju analiziranog programa u toku njegovog izvršavanja i ukazuju kako često se izvršavaju njegovi pojedini delovi (na primer ovakvi podaci mogu ukazati na broj izvršavanja pojedinih potprograma analiziranog programa)
  - profajleri mogu da prikupljaju statističke podatke o ponašanju analiziranog programa (*prof* i *gprof* profajleri) ili egzaktne podatke o ponašanju analiziranog programa (*qpt* profajler)
  - u prvom slučaju se podaci prikupljaju u okviru obrada vremenskih prekida, a u drugom slučaju se modifikuje izvršni oblik analiziranog programa da bi se prikupili tačni podaci u toku njegovog izvršavanja

Optimizacija međukoda

- Optimizacija međukoda je mašinski nezavisna optimizacija
- Optimizacija međukoda podrazumeva transformaciju međukoda koja ne menja njegovo značenje ali merljivo ubrzava njegovo izvršavanje ili smanjuje memorijske zahteve
- Kompajleri koji obavljaju ovakve transformacije se nazivaju optimizujući kompajleri *(optimizing compilers)*
- Optimizacija međukoda se zasniva na **analizi upravljačkog toka** *(control-flow analisys)* sa ciljem da se otkriju **bazni blokovi** *(basic blocks)*, odnosno sekvence naredbi koje se uvek izvršavaju od prve do poslednje, bez mogućnosti zaustavljanja ili grananja pre poslednje naredbe u sekvenci
- Ovakvi bazni blokovi predstavljaju čvorove **dijagrama toka** *(flow graph).* Njih povezuju usmerene spojnice koje pokazuju **redosled izvršavanja** *(flow of control).*

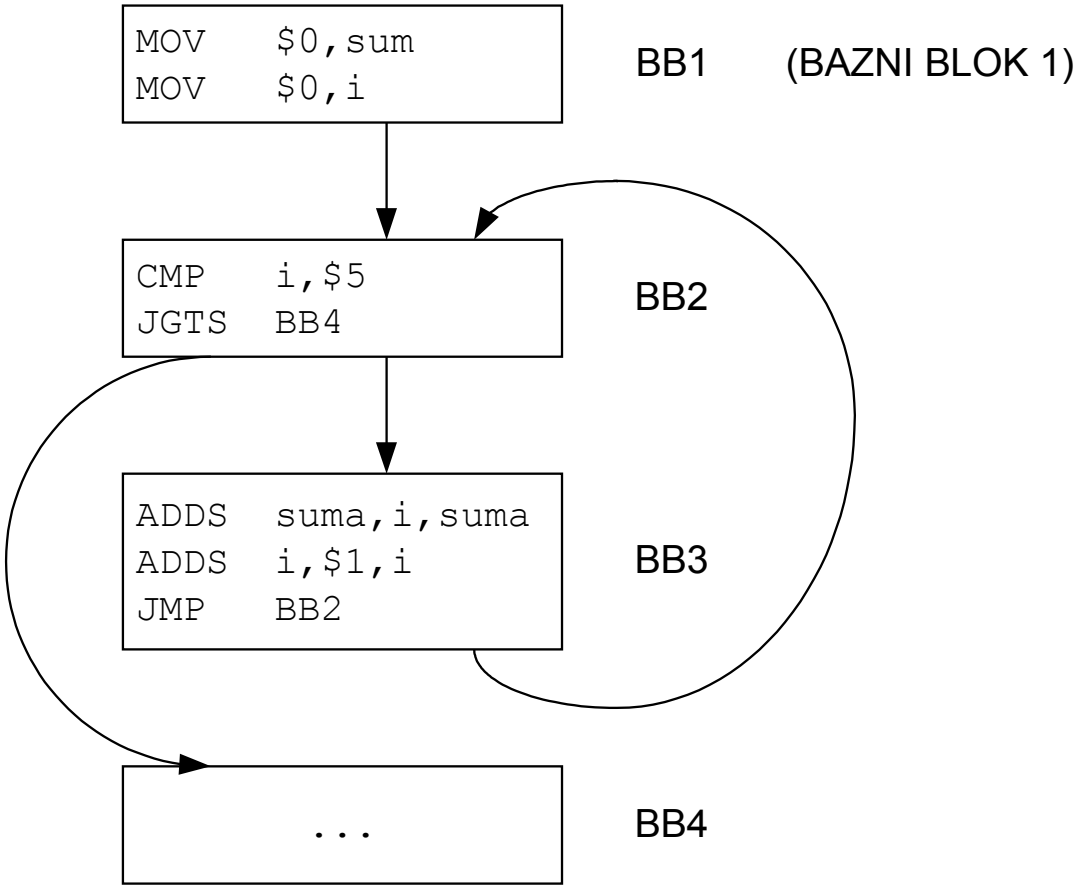
Optimizacija međukoda

- Na primer, međukodu:

```
                MOV   $0,suma
                MOV   $0,i
for0:
                CMP   i,$5
                JGTS  next0
                ADDS  suma,i,suma
                ADDS  i,$1,i
                JMP   for0
next0:
                ...
```

Optimizacija međukoda

odgovara dijagram toka:



Optimizacija međukoda

- Optimizacija je **lokalna** ako posmatra samo jedan bazni blok, inače je **globalna**
- Optimizacija međukoda pokušava da transformiše bazne blokove radi
  - eliminacije zajedničkih podizraza
  - eliminacije suvišnih naredbi *(redundant code)* i nedostupnih naredbi *(dead code)*
  - optimizacije petlji *(loop)*
  - algebarskih transformacija
- Odluka o transformaciji baznih blokova se donosi na osnovu poznavanja načina korišćenja promenljivih u programu. Radi toga se u okviru optimizacije međukoda obavlja **analiza toka podataka** *(data-flow analisys).* Cilj ovakve analize je da se za svaku promenljivu odredi poslednji bazni blok u kome se ona koristi (u kome se preuzima njena vrednost). Rezultati analize toka podataka se čuvaju u tabeli simbola.

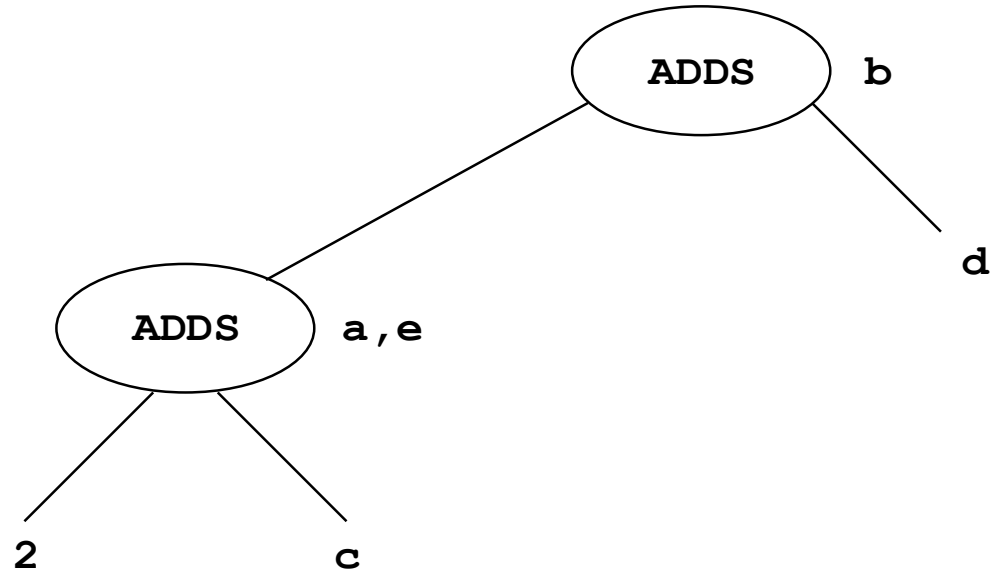
Optimizacija međukoda

- Za otkrivanje zajedničkih podizraza korisno je crtati **usmerene aciklične grafove** *(directed acyclic graph - DAG)* baznih blokova
- za bazni blok

```
ADDS   $2,c,a
ADDS   a,d,b
ADDS   $2,c,e
```

usmereni aciklični graf izgleda:

Optimizacija međukoda



- Iz grafa je očigledno da se promenljivim **a** i **e** pridružuje vrednost zajedničkog podizraza, pa se bazni blok može transformisati u:

Optimizacija međukoda

```
ADDS   $2,c,a
ADDS   a,d,b
MOV    a,e
```

- Primer suvišne naredbe je naredba koja pridružuje vrednost promenljivoj u baznom bloku od koga se ona više ne koristi. Ako se promenljiva **e** ne koristi iza baznog bloka iz prethodnog primera, taj bazni blok se može transformisati u

```
ADDS   $2,c,a
ADDS   a,d,b
```

(suvišne naredbe se obično javljaju nakon pojedinih transformacija baznih blokova u toku optimizacije)

Optimizacija međukoda

- Optimizacija petlji ima za cilj da smanji vreme izvršavanja petlje
  - to se može postići smanjenjem broja naredbi u telu petlje tako što se naredbe koje ne zavise od petlje pomeraju iz njenog tela unapred (ispred tela petlje). Na primer, ako se u petlji pristupa svim elementima niza **`n`** čiji indeksi **`i`** su iz intervala od 5 do 10 (**`var n:array[5..10] of integer`**), tada petlja sadrži računanje adresa pojedinih elemenata:

    `adresa_n + (i - 5) * 4`

    (podrazumeva se da je **`adresa_n`** adresa prvog elementa **`n[5]`** niza **`n`** i da ona nije poznata u kompilaciji, a da svaki element niza zauzima 4 bajta)

    Prethodni izraz se može transformisati u:

    `i * 4 + (adresa_n - 20)`

    a računanje izraza iz zagrade se može pomeriti ispred petlje, tako da se samo njegova vrednost koristi u petlji.

Optimizacija međukoda

- Algebarske transformacije obuhvataju
  - izračunavanje vrednosti podizraza čiji operandi su poznati u vreme kompilacije i zamena tih podizraza izračunatim vrednostima (znači umesto `2 * 3.14` koristi se `6.28`)
  - zamena skupljih operacija jeftinijim operacijama (umesto `m ** 2` koristiti `m * m`, umesto `2 * m` koristiti `m + m`, itd)
  - eliminisanje suvišnih izraza (`m + 0` zameniti sa `m`, a `m * 1` sa `m`)

Optimizacija međukoda

- Posebna vrsta optimizacije je **parcijalna** *(peephole)* **optimizacija** u okviru koje se posmatraju kratke sekvence uzastopnih naredbi i zamenjuju kraćim i bržim sekvencama. Sekvencu naredbi obrazuju naredbe koje se vide kroz zamišljeni prorez *(peephole),* pri čemu se podrazumeva da se pomenuti prorez pomera preko naredbi od početka ka kraju programa.
  - parcijalna optimizacija je najdelotvornija kada se uzastopno višestruko ponavlja
  - u toku parcijalne optimizacije traže se pojave unapred zadanih slučajeva kao što su:
    - suvišne *(redundant)* naredbe u koje spadaju i suvišni skokovi
    - nedostupne naredbe *(unreachable code/dead code)*
- Znači uoče se tipični slučajevi neefikasnog međukoda i za svaki od slučajeva se sprovodi parcijalna optimizacija da bi se pojave dotičnog slučaja pronašle i eliminisale

Optimizacija međukoda

- Primeri suvišnih naredbi:
  - naredba poređenja koja sledi iza identične naredbe, a da između njih nije bilo izmena uslovnih *(condition code)* bita iz status registra

```
CMP     a,b
JEQ     false0
CMP     a,b             ←
JLEU    false1
```

  - naredba punjenja *(load)* registra koja sledi iza identične naredbe, a da između njih nije bilo izmena sadržaja registra i obrnuto *(store)*

```
MOV   $7, a             ←
ADD   %0, %1, %0
MOV   $7, a             ←
```

  - uzastopne *load* i *store* naredbe koje se odnose na isti registar i istu memorijsku lokaciju

```
a = … ;      MOV   %0, a          ←
                   …
b = a … ;    MOV   a, %0          ←
```

  - cilj naredbe bezuslovnog skoka je naredba bezuskovnog skoka

Optimizacija međukoda

- Primeri nedostupnih naredbi (koje ne mogu biti izvršene)
  - naredbe koje slede iza naredbe bezuslovnog skoka, a nisu cilj neke druge naredbe skoka

```
        JEQ     Labela1
        JNE     Labela2
        MOV     $12, %0          ←
Labela1
        …
Labela2
        …
```

Optimizacija međukoda

- Suvišne i nedostupne naredbe se izbacuju
- Operandi naredbi suvišnih skokova se menjaju tako da se izbegavaju suvišni skokovi

Generisanje koda

- Generator koda:
  - preuzima ispravan međukod i tabelu simbola, a
  - proizvodi (1) izvršni (apsolutni) mašinski kod ili
    (2) objektni (relokatibilni) mašinski kod ili
    (3) asemblerski kod
- Podrazumeva se da je proizvedeni kod ispravan i da efikasno koristi resurse računara
- Generisanje koda se zasniva na određivanju načina implementacije svake naredbe međukoda pomoću naredbi koda

Generisanje koda

- Na primer, naredba hipotetskog asemblerskog jezika:

```
ADDS   x,y,z
```

može biti implementirana pomoću sledećih naredbi asemblerskog jezika za procesor *Intel 8086*:

```
mov    x,%ax
add    y,%ax
int0
mov    %ax,z
```

- Generisanje koda može da proizvede suvišne naredbe. Na primer za međukod:

```
ADDS   x,y,z
ADDS   z,p,q
```

generisani kod je:

Generisanje koda

```
mov    x,%ax        naredba 1
add    y,%ax        naredba 2
into                naredba 3
mov    %ax,z        naredba 4
mov    z,%ax        naredba 5
add    p,%ax        naredba 6
into                naredba 7
mov    %ax,q        naredba 8
```

a u njemu je peta naredba suvišna, a potencijalno i četvrta, ako se promenljiva **z** ne koristi iza dotičnog baznog bloka

- Zato se u optimizaciju koda obično uključuje parcijalna optimizacija, koja traži pojave slučajeva kao što su prethodno pomenuti

Generisanje koda

- Prilikom generisanja koda treba voditi računa o specifičnostima mašine, pa naredbu međukoda

```
ADDS   x,$1,x
```

  treba implementirati naredbom

```
inc    x
```

  a ne pomoću prethodno korišćene četiri naredbe.
- Za generisanje koda je važno korišćenje registara (procesora), jer je registarsko adresiranje najbrži i najkraći način adresiranja
- Zato je potrebno najčešće korišćene vrednosti čuvati u registrima
- To se ostvaruje određivanjem promenljivih kojima treba dodeliti registar i zauzimanjem registara za pojedine od ovih promenljivih (*C* kompajler dozvoljava programeru da označi promenljive kojima treba dodeliti registre)

Generisanje koda

- Obično se unapred napravi raspodela registara, tako da se grupa registara (koje ne koristi ni procesor ni operativni sistem) proglasi **radnim registrima** i stavi na raspolaganje kompajleru
- Rukovanje radnim registrima postaje problematično kada ponestane radnih registara. Tada treba osloboditi neki od radnih registara. To podrazumeva spašavanje sadržaja registra u memorijsku lokaciju promenljive za koju je dotični registar bio zauzet, ako se dotična promenljiva i dalje koristi. Izbor registra koji se oslobađa treba napraviti tako da se minimiziraju spašavanja sadržaja registara u memorijske lokacije. Za izbor registra koji se oslobađa koristi se **algoritam bojenja grafa** *(graph coloring).* Čvorove grafa predstavljaju promenljive koje su dodeljene registrima. Međusobno su povezani čvorovi promenljivih koje se istovremeno koriste. Za bojenje se koristi onoliko boja koliko ima radnih registara. Čvorovi grafa se boje tako da međusobno povezani čvorovi budu obojeni različitim bojama. Na ovaj način se sprečava da za dve promenljive, koje se istovremeno koriste, a dodeljeni su im registri, bude zauzet isti radni registar.

Generisanje koda

- Za uspešno generisanje koda potrebno je
  - pamtiti kako se koristi svaki radni registar (za šta se uvodi posebna struktura podataka)
  - pamtiti za svaku vrednost (konstante, promenljive, funkcije, ...) gde se nalazi (zato se koristi tabela simbola)
  - pripremiti generisanje naredbe zauzimanjem radnog registra i pronalaženjem gde se nalaze potrebne vrednosti
  - generisati naredbu (vodeći računa da se odabere najbolja alternativa, ako se mogu birati razne naredbe i načini adresiranja)
  - ažurirati podatke o korišćenju radnih registara i o mestima smeštanja vrednosti

Generisanje koda

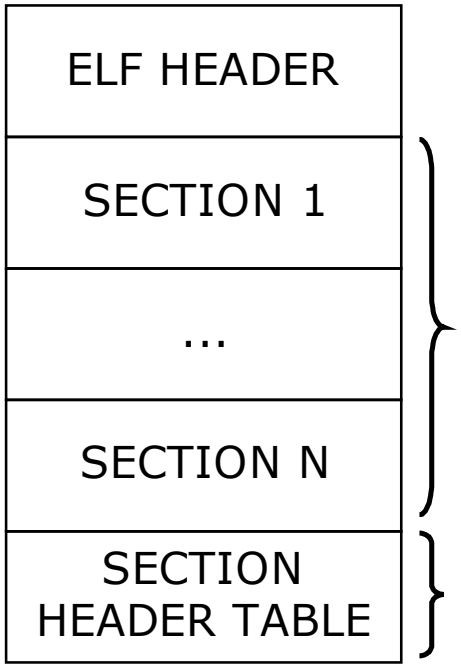
- Rukovanje radnim registrima može uticati na izbor redosleda (nezavisnih) naredbi (čiji međusobni položaj je proizvoljan)
- Postoje **generatori generatora koda** *(code-generator generators)* koji automatizuju pravljenje generatora koda *(burg, iburg)*

Optimizacija koda

- Zavisi od optimizacije međukoda
- Obuhvata parcijalnu optimizaciju
- Ima za cilj da na najbolji način iskoristi specifičnosti računara

Format objektne datoteke

- ELF *(Executable and Linkable Format)* format objektne datoteke (*UNIX/LINUX*):



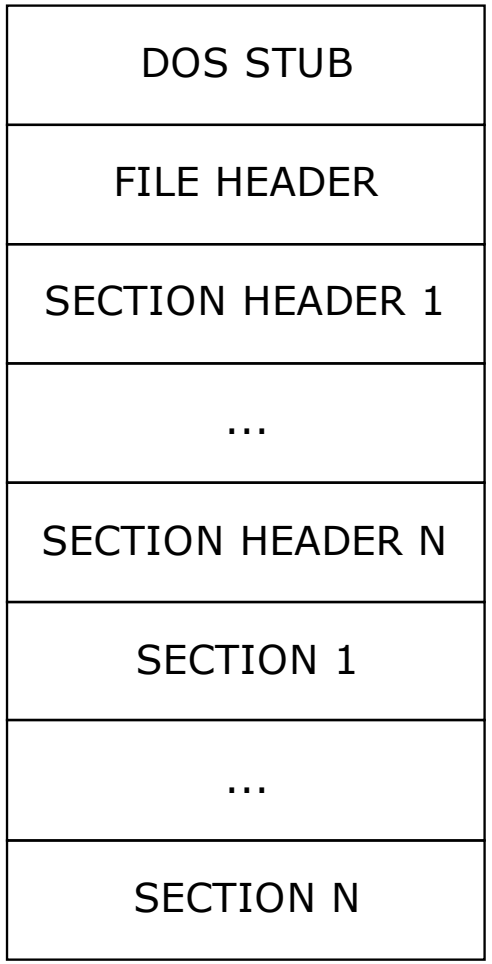
ELF zaglavlje se uvek nalazi na početku datoteke i opisuje organizaciju datoteke

Sekcija sadrži naredbe, podatke, tabelu simbola, informacije o relokaciji, itd.
Neke od specijalnih sekcija koje se mogu naći u ELF objektnoj datoteci su: .text (naredbe), .data (inicijalizovane promenljive), .bss (neinicijalizovane promenljive), .debug (debug informacije), .symtab (tabela simbola), itd.

Sve sekcije u objektnoj datoteci se mogu naći pomoću ove tabele. Svako polje tabele odgovara jednoj sekciji u datoteci, i sadrži ime, tip, početnu adresu, veličinu, poravnanje itd.

Format objektne datoteke

- PE *(Portable Executable)* format objektne datoteke (*Windows*):



Svaka PE datoteka je izvršiva pod MS-DOS-om.

Zaglavlje datoteke govori za koju mašinu je binarni kod predviđen, koliko sekcija sadrži, vreme linkovanja, da li sadrži izvršni kod ili DLL, itd.

Zaglavlje svake sekcije opisuje da li sadržaj sekcije predstavlja izrvšiv kod, inicijalizovane promenljive, neinicijalizovane promenljive, da li se promenljive samo mogu čitati ili se u njih može i pisati, itd.

Svaka sekcija sadrži podatke o poravnanju, podatke o vrsti sadržaja, da li može biti "shared" itd., i sam sadržaj - ono što je neophodno za izvršavanje programa.
Postoji nekoliko predefinisanih sekcija: .text (sadrži kod), .bss, .rdata, .data (sadrže neinicijalizovane, read-only i ostale promenljive), .rsrc (sadrži informacije o resursima), .debug (sadrži debug informacije).

Literatura

- Aho A.V., Sethi R., Ullman J.D., “Compilers – principles, techniques, and tools”, Addison-Wesley, Reading, Massachusetts, 1986
- Kulenović A., “Osnove projektovanja kompajlera”, Svetlost, Sarajevo, Jugoslavija, 1991
- Muchnick S.S., “Advanced compiler design & implementation”, Morgan Kaufmann, San Francisco, 1997
- M. E. Lesk and E. Schmidt, “Lex - A Lexical Analyzer Generator”
- Stephen C. Johnson, “Yacc: Yet Another Compiler-Compiler”, AT&T Bell Laboratories, New Jersey
- Michael L. Scott, “Programming Language Pragmatics”, Morgan Kaufmann, San Francisco, 2000