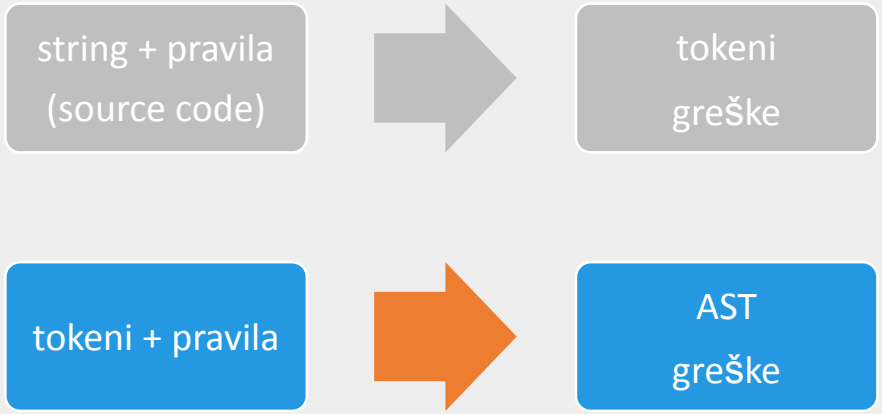
Programski prevodioci

02 Sintaksna analiza

Fakultet tehničkih nauka, Novi Sad
22-23/Z
Dunja Vrbaški



Sintaksna analiza – provera da li je ulazni niz tokena formiran na osnovu pravila po kom se grade konstrukcije u PL

Šta bi bila greška u sintaksnoj analizi?

```
if x == 5
{
   y 3 //komentar
}
```

Zavisi od toga kako definišemo!

Možda uslov mora da ima zagrade.

Možda nijedna struktura/konstrukcija u našem jeziku nema oblik: <identifikator> <broj>.

Možda svaki iskaz mora imati tačku zarez na kraju.
...

Razmatranje tipa promenljive x (da li je x uopšte broj) i da li sme da se vrši ovakvo upoređivanje pripada sledećoj fazi (type checking; semantička analiza)

Nekoliko odluka

- koje tokene ćemo prepoznavati
- koji tokeni imaju vrednosti

- kako skener i parser komuniciraju, kako se preuzimaju tokeni

- kako definisati sintaksu
- kako implementirati parser (sintaksni analizator)
- koje strukture ćemo koristiti

Kako implementirati parser?

- ručno, proizvoljno
- ručno + gramatika
- alat + gramatika

Bez obzira na pristup, pojavljuju se određeni problemi i razmatranja: dvosmislenosti, konflikti, prioriteti, obrada grešaka,...

Postavlja se i pitanje šta je ulaz za sledeću fazu (semantička analiza)?

- ništa – radi se paralelno, odmah utvrđujemo da li je i sintaksno i semantički sve u redu
- neka interna struktura na osnovu koje možemo lako utvrditi da li važe semantička pravila

Postoje bar dva zadatka:

- osnovni: prepoznati da li je izvorni tekst u skladu sa pravilima sintaksne PL
  npr koristeći gramatike (rezultat: ok ili greške)

- uglavnom postoji: kreiranje apstraktnog sintaksnog stabla (AST-Abstract Syntax Tree)
  - + kreiranje tabele simbola
  - ...

MI: osnovno + tabelu simbola razmatramo u sledećoj fazi

Formalni jezici i parsiranje

Mogu se koristiti formalni jezici za opis sintakse programskog jezika.
(formalizacija)

Zašto bismo to radili?

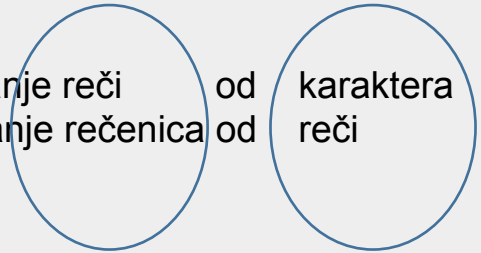
- Formalna i jednostavna reprezentacija sintakse jezika

- Bison - na osnovu definisane gramatike automatski kreira parser
- Ručno - možemo iskoristiti za bolju implementaciju

Pokušaćemo formalno da opišemo pravila strukture koda napisanog u PL

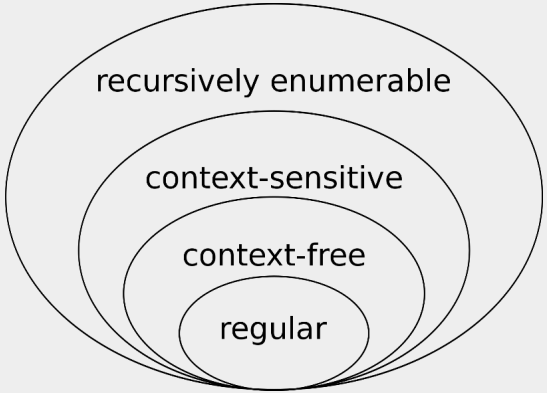
```
if_naredba = if (uslov) naredbe else naredbe
naredbe = if_naredba ili for_naredba ili naredba_dodele
...
```

Skeniranje – konstruisanje reči od karaktera
Parsiranje – konstruisanje rečenica od reči



stringovi *azbuka*

Uglavnom mislimo na CFG gramatike (Context Free Grammar, kontekstno slobodna gramatika)



Definicija: Gramatika je uređena četvorka $G = (\Sigma, N, P, S)$ gde je:

- Σ skup simbola (terminala)
- N skup pojmova (neterminala)
- P skup pravila izvođenja
- S početni pojam

- Σ, N, P *su neprazni, konačni, skupovi*
- $\Sigma \cap N = \varnothing$
- P *je skup pravila u obliku* $(\Sigma \cup N)^* \mathbf{N} (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$

Definicija: Gramatika je uređena četvorka $G = (\Sigma, N, P, S)$ gde je:

- Σ skup simbola (terminala)
- N skup pojmova (neterminala)
- P skup pravila izvođenja
- S početni pojam

- Σ, N, P *su neprazni, konačni, skupovi*
- $\Sigma \cap N = \varnothing$
- *P je skup pravila u obliku* $(\Sigma \cup N)^* \mathbf{N} (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$

Kod CFG gramatike pravila su oblika: $\mathbf{N} \rightarrow (\Sigma \cup N)^*$

Kod CFG gramatike pravila su oblika:

$$Pojam \rightarrow X_1 X_2 \ldots X_n$$

$$X_i \; je \; pojam \; ili \; simbol \; ili \; \varepsilon$$

pojam = neterminal
često koristimo izraze “leva strana” i “desna strana pravila”

$$S \to aSa$$
$$S \to bSb$$
$$S \to a$$
$$S \to b$$
$$S \to \varepsilon$$

Pravila, pre svega, ukazuju na konkatenaciju (redosled).

Često su rekurzivna čime se, praktično, omogućava ponavljanje.

Pravila bez rekurzije, praktično, omogućavaju zaustavljanje rekurzije.

Više pravila za isti pojam, praktično, omogućava uniju (alternative).

Prazna reč, praktično i između ostalog, omogućava zaustavljanje rekurzije/ponavljanja.

Izvođenje

CFG:

- Konačan skup simbola (terminala) Σ
- Konačan skup pojmova (neterminala) N
- Konačan skup pravila P u obliku: $Pojam \rightarrow X_1 X_2 \dots X_n$
- Početni pojam S

Gramatika definiše pravila po kojima se grade reči (stringovi, rečenice?) u jeziku.
Svaki string je morao nastati primenom pravila u određenom redosledu.

Izvođenje – koraci koji su doveli do izgradnje stringa. Niz primenjenih pravila.

CFG:

- Konačan skup simbola (terminala) Σ
- Konačan skup pojmova (neterminala) N
- Konačan skup pravila P u obliku: $Pojam \rightarrow X_1 X_2 \ldots X_n$
- Početni pojam S

Jezik: $L = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

$S \rightarrow aSa$
$S \rightarrow bSb$
$S \rightarrow a$
$S \rightarrow b$
$S \rightarrow \varepsilon$

Jezik – svi stringovi koji se mogu dobiti nekim izvođenjem

$\Rightarrow$ primena pravila (jedan korak izvođenja)
$\Rightarrow^*$ izvođenje, primena niza pravila (više koraka izvođenja)

Jezik $L = \{0^n 1^n \mid n \geq 1\}$

simboli: $\{0, 1\}$
pojmovi: $\{S\}$
početni pojam: S
pravila:
$S \to 0S1$
$S \to 01$

Primeri izvođenja:

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow \ 00001111$
$S \Rightarrow^* 00001111$
$00001111 \in L$

$S \Rightarrow 01$
$S \Rightarrow^* 01$
$01 \in L$

$S \Rightarrow 0S1 \Rightarrow 0011$
$S \Rightarrow^* 0011$
$0011 \in L$

Jezik $L = \{0^n 1^n \mid n \geq 1\}$

simboli: $\{0, 1\}$
pojmovi: $\{S\}$
početni pojam: S
pravila:
$S \to 0S1$
$S \to 01$

$\Rightarrow^n$ nekad se navodi i koje pravilo je bilo primenjeno

$$S \Rightarrow^1 0S1 \Rightarrow^1 00S11 \Rightarrow^1 000S111 \Rightarrow^2 00001111$$

Pregledniji zapis:

$$\begin{aligned} & S \\ \Rightarrow\ & 0S1 \\ \Rightarrow\ & 00S11 \\ \Rightarrow\ & 000S111 \\ \Rightarrow\ & 00001111 \end{aligned}$$

BNF (Backus-Naur) forma

pravila sintakse PL se često navode u BNF formi

- pojmovi (neterminali) se navode koristeći “<” i “>”, ostalo su simboli (terminali)
- ::= se koristi umesto →

<assignment statement> ::= <variable> = <expression>

Zapisi budu i drugačiji:

- izostavlja se <>, ali se nekako pravi razlika između pojmova i simbola
- koristi se →
- koristi se :=
- | se koristi za alternative (umesto navođenja posebnih pravila)

assignment statement → variable = expression

```
time   → hour ":" minute
       | hour ":" minute ":" second

hour   → 2DIGIT

minute → 2DIGIT

second → 2DIGIT


2DIGIT → DIGIT DIGIT

DIGIT  → “0”|”1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9“
```

Pojmovi (NONTERMINALs): time, hour, minute, second, 2DIGIT, DIGIT
Simboli (TERMINALs): “0”, “1”, “2”…, “9”, “:”

Terminalno, završno, nema dalje produkcija
Terminali se ne mogu naći sa leve strane

```
time   → hour ":" minute
       | hour ":" minute ":" second

hour   → 2DIGIT

minute → 2DIGIT

second → 2DIGIT


2DIGIT → DIGIT DIGIT

DIGIT  → “0”|”1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9“
```

2DIGIT, DIGIT - mogli bi biti tokeni, prepoznati u leksičkoj analizi, na osnovu druge gramatike (gramatika leksike, regularni izrazi). Tada bi oni bili terminali za tu gramatiku.

Skener – azbuka karakteri, RG
Parser – azbuka tokeni, CFG

```
text
   → sentence
   → text sentence

sentence
   → words dot

words
   → capital_word
   → words word
   → words capital_word

word
   → small_letter
   → word small_letter

capital_word
   → capital_letter
   → capital_word small_letter

dot
   → "."
```

Primer jezika i odgovarajuća gramatika za opis teksta.

Danas pada kiša.	OK
pada.	ERR
kiša pada	ERR

```
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"
```

Šta bi još mogli biti tokeni?

Odluka zavisi od potreba!

Proširene BNF forme

Dodaju se oznake

- […] – ponavljanje sadržaja 0 ili 1 put
- { … } – ponavljanje sadržaja 0 ili više puta
- (...) – grupisanje

- nekad: “,” konkatenacija, “;” kraj

- nekad: i *, +, ?

```
text
   → sentence
   → text sentence

sentence
   → words dot

words
   → capital_word
   → words word
   → words capital_word

word
   → small_letter
   → word small_letter

capital_word
   → capital_letter
   → capital_word small_letter

dot
   → "."
```

```
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"
```

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


word
   → small_letter { small_letter }
capital_word
   → capital_letter { small_letter }
dot
   → "."
```

```
text
   → sentence
   → text sentence

sentence
   → words dot

words
   → capital_word
   → words word
   → words capital_word
```

pp: capital_word, word, dot su terminali

Kako funkcionišu ova pravila izvođenja?

Ovo je tekst.

`w = capital_word word word dot`

text
⇒ *sentence*
⇒ *words* dot
⇒ *words* word dot
⇒ words word word dot
⇒ capital_word word word dot

text $\Rightarrow^*$ w

```
text
   → sentence
   → text sentence

sentence
   → words dot

words
   → capital_word
   → words word
   → words capital_word
```

pp: capital_word, word, dot su terminali

Kako funkcionišu ova pravila izvođenja?

Ovo je tekst.

`w = capital_word word word dot`

text
⇒ *sentence*
⇒ *words* dot
⇒ *words* word dot
⇒ words word word dot
⇒ capital_word word word dot

$\text{text} \Rightarrow^* w$

Koji je bio misaoni postupak kojim smo došli do ovog niza pravila izvođenja?

Koji bi bio misaoni postupak za utvrđivanje greške kod neispravnih stringova?
“Pada kiša”, “pada kiša.” i “pada”

Na osnovu pravila izvođenja se može implementirati parser koji odgovara na pitanje:
Da li niz tokena može biti kreiran na osnovu gramatike / Da li postoji izvođenje?

Usput, čitavo prevođenje može biti zasnovano na gramatici i parsiranju.

2 osnovna tipa parsera:

Silazno parsiranje (top-down)

Pokušavamo da od početnog pojma izvedemo ulazni niz simbola prateći pravila produkcije
Akcije: Leva strana pravila se zamenjuje desnom

Uzlazno parsiranje (bottom-up)

Pokušavamo da od ulaznog niza simbola redukcijom dobijemo početni pojam
Akcije: Desna strana pravila se zamenjuje levom

TD – jednostavnije za direktnu implementaciju
BU – širu klasu gramatika podržavaju

Silazno parsiranje (top-down)

Pokušavamo da od početnog pojma izvedemo ulazni niz simbola prateći pravila produkcije
Akcije: Leva strana pravila se zamenjuje desnom

text
→ *sentence*
→ *text sentence*

Recursive descent parser
Rekurzivni spust

- Postoji funkcija za svaki pojam odgovorna za obradu tog pojma
- Početna funkcija odgovara početnom pojmu i ona poziva odgovarajuće funkcije (za pojmove sa desne strane pravila)
- Rekurzivne funkcije zbog rekurzivne prirode gramatike PL
- Algoritam je intuitivan i relativno jednostavan za direktnu implementaciju (ručno)

Problemi

text
→ *sentence*
→ *text sentence*

Šta raditi kod alternativa – koju pozvati? Odrediti heuristiku.

Šta je greška? – kraj izvršavanja pre kraja ulaznog teksta, pogrešan izbor alternative?

Šta raditi?
- Backtracking ili
- Nešto složenije, ali pametnije (predictive parsing, First and Follow skupovi, LL(k),...)

- Leva rekurzija može biti problem - postoje heurisitike za uklanjanje rekurzije.

Uzlazno parsiranje (bottom-up)

Pokušavamo da od ulaznog niza simbola redukcijom dobijemo početni pojam
Akcije: Desna strana pravila se zamenjuje levom

shift-reduce algoritam

- Preuzima se jedan simbol (token) iz ulaznog niza *(shift)*
- Dodaje se na kraj prethodno formirane sekvence pojmova/simbola
- Proverava se da li novoformirana sekvenca odgovara desnoj strani nekog pravila
    - ako odgovara – zamenjuje se čitava sekvenca pojmom sa leve strane *(reduce)*
    - ako ne odgovara – nastavlja se preuzimanje

→ Pravila izvođenje se otkrivaju u obrnutom redosledu od top-down.

U nastavku – prikaz izvršavanja algoritma na primeru.
Pre toga, praktičan uvod u bison.

bison

Bison je alat za generisanje parsera.

Na osnovu gramatike sintakse nekog jezika generiše funkciju koja predstavlja parser za taj jezik. Jednostavno se integriše sa flexom koji se koristi za leksičku analizu.

- bottom-up
- shift-reduce
- moguće gramatike su podskup CFG gramatika

Teorija:
Regularnim jezicima (gramatikama) odgovaraju konačni automati (finite automata)
CF jezicima (gramatikama) odgovaraju potisni automati (pushdown automata)

Flex – koristi za automatsko generisanje skenera
Bison - koristi za automatsko generisanje parsera

Simuliraju/implementiraju automate koji prepoznaju stringove jezika (leksika, sintaksa) generisanog odgovarajućom gramatikom.

IDEJA

MI: Sintaksa se predstavi gramatikom, formalno
TEORIJA: Za CFG postoji odgovarajući potisni automat (konačni automat + stek/memorija)
ALAT: Formira odgovarajući automat koji će biti zadužen za prepoznavanje

→ imamo mašinu/alat/implementaciju za prepoznavanje programskih konstrukcija i prihvatanje ulaznog programskog koda

POSTUPAK

1. Definisati tokene
2. Definisati gramatiku sintakse
3. Povezati flex i bison
4. Omogućiti da flex vraća odgovarajuće tokene
5. Kompajlirati (u određenom redosledu) obe stvari da se dobiju skener i parser
6. Pokretanje

POSTUPAK

1. Definisati tokene (`jezik.y`)
2. Definisati gramatiku sintakse (`jezik.y`)
3. Povezati flex i bison (include + deklaracije)
4. Omogućiti da flex vraća odgovarajuće tokene (`jezik.l` → `return IF`)
5. Kompajlirati (u određenom redosledu) obe stvari da se dobiju skener, parser i main (`make`)
   `lex.yy.c` sa `yylex()`
   `jezik.tab.c` sa `yyparse()`
6. Pokretanje

jezik.l

```
#include "syntax.tab.h"




%%
... regularni izrazi ...

"if" { return _IF; }

%%

int main() {
   yylex();
}
```

jezik.y

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

%token _IF
%token _ID
...

%%
... gramatika ...

if_statement...

%%

int main() {
   return yyparse();
}
```

```
…

%%
… gramatika …

if_statement…    { printf(“…”); }

%%

int main() {
   return yyparse();
}
```

Možemo dodati korisničke akcije

```
text
   → sentence
   → text sentence

sentence
   → words dot

words
   → capital_word
   → words word
   → words capital_word

word
   → small_letter
   → word small_letter

capital_word
   → capital_letter
   → capital_word small_letter

dot
   → "."
```

Primer jezika i odgovarajuća gramatika za opis teksta.

Danas pada kiša.	OK
pada.	ERR
kiša pada	ERR

```
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"
```

```
text
   → sentence
   → text sentence

sentence
   → words dot

words
   → capital_word
   → words word
   → words capital_word

word
   → small_letter
   → word small_letter

capital_word
   → capital_letter
   → capital_word small_letter

dot
   → "."
```

```
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"
```

```
"."          { return DOT; }

[A-Z][a-z]*  { return CAPITAL_WORD; }

[a-z]+       { return WORD; }
```

jezik.l

```
text
  : sentence
  | text sentence
  ;

sentence
  : words DOT
  ;

words
  : CAPITAL_WORD
  | words CAPITAL_WORD
  | words WORD
  ;
```

jezik.y

Možemo zadati i sledeći zadatak:
Prebrojati koliko ima reči i rečenica u ulaznom tekstu.

Kako?

Možemo zadati i sledeći zadatak:
Prebrojati koliko ima reči i rečenica u ulaznom tekstu.

```
text
  : sentence
  | text sentence
  ;

sentence
  : words DOT
      { sentence_counter++; }
  ;
```

```
words
  : CAPITAL_WORD
      { word_counter++; }
  | words CAPITAL_WORD
      { word_counter++; }
  | words WORD
      { word_counter++; }
  ;
```

Dodajemo korisničke akcije. Gde ispisujemo ove vrednosti?

Kako funkcioniše?

shift-reduce algoritam

- Preuzima se jedan simbol (token) iz ulaznog niza *(shift)*
- Dodaje se na kraj prethodno formirane sekvence pojmova/simbola
- Proverava se da li novoformirana sekvenca odgovara desnoj strani nekog pravila
    - ako odgovara – zamenjuje se čitava sekvenca pojmom sa leve strane *(reduce)*
    - ako ne odgovara – nastavlja se preuzimanje

Stek

- preuzeti simboli se smeštaju na stek (shift)
- prilikom redukcije prepoznata sekvenca (desna strana pravila) se skida sa steka, a na stek se postavlja pojam na koji se sekvenca redukuje (leva strana pravila)

Tabela prelaza

- upravlja radom parsera
- action i goto tabele
- akcije: **SHIFT, REDUCE, ERROR, ACCEPT (STOP)**
  - shift - koje je novo stanje + stek
  - reduce - po kom pravilu + stek
  - ERROR - prazno, dosta elemenata
  - ACCEPT - uspešno parsiranje

	$token_1$	...	$token_m$
$stanje_0$			
...			
$stanje_n$			

„Ovo je tekst.“

CAPITAL_WORD WORD WORD DOT EOF

text
- → *sentence*
- → *text sentence*

sentence
- → *words* DOT

words
- → CAPITAL_WORD
- → *words* WORD
- → *words* CAPITAL_WORD

„Ovo je tekst.“

WORD WORD DOT EOF

text
- → *sentence*
- → *text sentence*

sentence
- → *words* DOT

words
- → CAPITAL_WORD
- → *words* WORD
- → *words* CAPITAL_WORD

CAPITAL_WORD

„Ovo je tekst.“

WORD WORD DOT EOF

CAPITAL_WORD

text
- → *sentence*
- → *text sentence*

sentence
- → *words* DOT

words
- → CAPITAL_WORD
- → *words* WORD
- → *words* CAPITAL_WORD

„Ovo je tekst.“

WORD WORD DOT EOF

text
- → *sentence*
- → *text sentence*

sentence
- → *words* DOT

words
- → CAPITAL_WORD
- → *words* WORD
- → *words* CAPITAL_WORD

words

„Ovo je tekst.“

WORD DOT EOF

text
- → *sentence*
- → *text sentence*

sentence
- → *words* DOT

words
- → CAPITAL_WORD
- → *words* WORD
- → *words* CAPITAL_WORD

WORD
words

„Ovo je tekst.“

WORD DOT EOF

WORD
words

text
→ *sentence*
→ *text sentence*

sentence
→ *words* DOT

words
→ CAPITAL_WORD
→ *words* WORD
→ *words* CAPITAL_WORD

„Ovo je tekst.“

WORD DOT EOF

text
- → *sentence*
- → *text sentence*

sentence
- → *words* DOT

words
- → CAPITAL_WORD
- → *words* WORD
- → *words* CAPITAL_WORD

words

„Ovo je tekst.“

DOT EOF

text
- → *sentence*
- → *text sentence*

sentence
- → *words* DOT

words
- → CAPITAL_WORD
- → *words* WORD
- → *words* CAPITAL_WORD

WORD
words

„Ovo je tekst.“

DOT EOF

text
→ *sentence*
→ *text sentence*

sentence
→ *words* DOT

words
→ CAPITAL_WORD
→ *words* WORD
→ *words* CAPITAL_WORD

WORD
words

„Ovo je tekst.“

DOT EOF

text
→ *sentence*
→ *text sentence*

sentence
→ *words* DOT

words
→ CAPITAL_WORD
→ *words* WORD
→ *words* CAPITAL_WORD

words

„Ovo je tekst.“

EOF

text
- → *sentence*
- → *text sentence*

sentence
- → *words* DOT

words
- → CAPITAL_WORD
- → *words* WORD
- → *words* CAPITAL_WORD

DOT
words

„Ovo je tekst.“

EOF

text
- → *sentence*
- → *text sentence*

sentence
- → *words* DOT

words
- → CAPITAL_WORD
- → *words* WORD
- → *words* CAPITAL_WORD

DOT
words

„Ovo je tekst.“

EOF

text
→ *sentence*
→ *text sentence*

sentence
→ *words* DOT

words
→ CAPITAL_WORD
→ *words* WORD
→ *words* CAPITAL_WORD

sentence

„Ovo je tekst.“

EOF

text
→ *sentence*
→ *text sentence*

sentence
→ *words* DOT

words
→ CAPITAL_WORD
→ *words* WORD
→ *words* CAPITAL_WORD

sentence

„Ovo je tekst.“

EOF

text
→ *sentence*
→ *text sentence*

sentence
→ *words* DOT

words
→ CAPITAL_WORD
→ *words* WORD
→ *words* CAPITAL_WORD

„Ovo je tekst.“

text
→ *sentence*
→ *text sentence*

sentence
→ *words* DOT

words
→ CAPITAL_WORD
→ *words* WORD
→ *words* CAPITAL_WORD

EOF
text

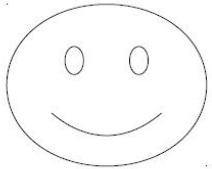
„Ovo je tekst.“

text
→ *sentence*
→ *text sentence*

sentence
→ *words* DOT

words
→ CAPITAL_WORD
→ *words* WORD
→ *words* CAPITAL_WORD

EOF
text



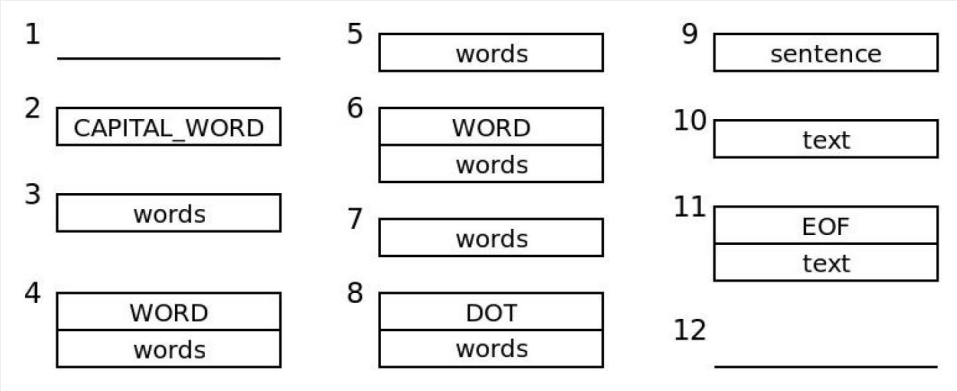
„Ovo je tekst.“

text
→ *sentence*
→ *text sentence*

sentence
→ *words* DOT

words
→ CAPITAL_WORD
→ *words* WORD
→ *words* CAPITAL_WORD

```
„Ovo je tekst.“
CAPITAL_WORD WORD WORD DOT EOF
```



Stablo parsiranja

Struktura podataka koja odgovara gramatičkoj strukturi ulaznog stringa.

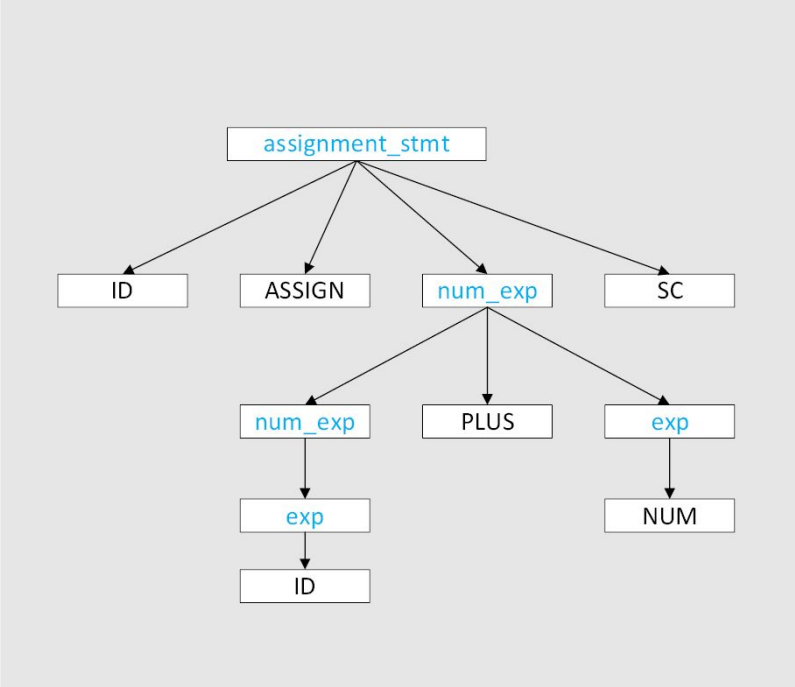
Stablo parsiranja:

- koren je početni pojam
- listovi predstavljaju simbole (terminale) ili prazan simbol
- unutrašnji čvorovi predstavljaju pojmove (neterminale)
- unutrašnji čvor koji predstavlja pojam Y i koji ima naslednike X_1,X_2,...X_n predstavlja pravilo izvodjenja $Y \rightarrow X_1 X_2 \dots X_n$

assignment_stmt → ID ASSIGN num_exp SC
num_exp → exp
num_exp → num_exp PLUS exp
num_exp → num_exp MINUS exp
exp → NUM
exp → ID

a = b + 3;
w = ID ASSIGN ID PLUS NUM SC

terminali (tokeni) – označeni velikim slovom
gramatika slična kao miniC

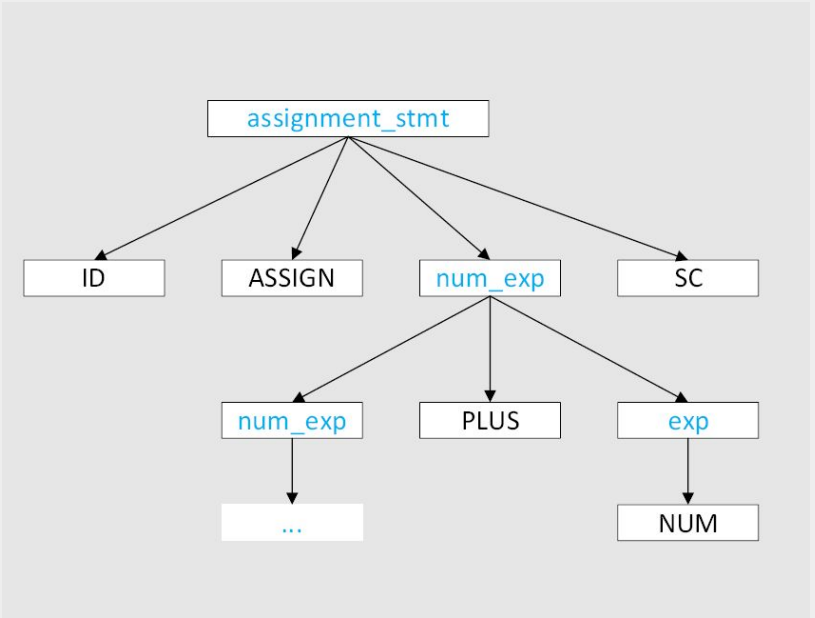


Kako bismo zapisali izvođenje?

Čitajući listove s leva na desno dobijamo polazni string

assignment_stmt → ID ASSIGN num_exp SC
num_exp → exp
num_exp → num_exp PLUS exp
num_exp → num_exp MINUS exp
exp → NUM
exp → ID

~~a = b + 3;~~
a = b + c + 3;
w = ID ASSIGN ID PLUS ID PLUS NUM SC



Parsiranje praktično formira stablo parsiranja
↔
Parsiranje je proces pronalaženja stabla parsiranja

top-down od korena ka listovima
bottom-up od listova ka korenu

U prevodiocu može postojati - eksplicitno ili implicitno

stablo parsiranja, stablo izvođenja, konkretno sintaksno stablo != apstraktno sintaksno stablo