

Formalne metode

u softverskom inženjerstvu

09 RG i CSG - programski jezici

ETFBL 24-25

Dunja Vrbaški

$$\begin{aligned} S &\rightarrow aS \\ S &\rightarrow bS \\ S &\rightarrow \epsilon \end{aligned}$$

$$\begin{aligned} S &\rightarrow aSa, \\ S &\rightarrow bSb, \\ S &\rightarrow \epsilon, \end{aligned}$$

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow ab \end{aligned}$$

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow Sb \\ S &\rightarrow \epsilon \end{aligned}$$

$$\begin{aligned} A &\rightarrow aA \\ A &\rightarrow B \\ B &\rightarrow bB \\ B &\rightarrow \epsilon \end{aligned}$$

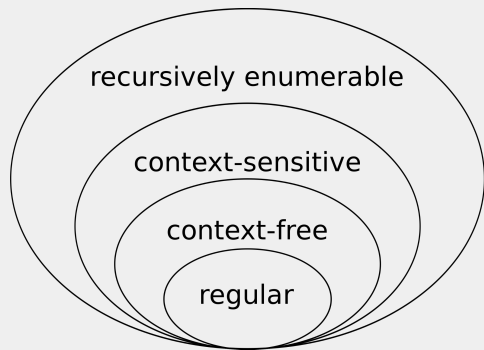
$$\begin{aligned} S &\rightarrow AX \mid XB \\ X &\rightarrow aXb \mid \epsilon \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \end{aligned}$$

$$\begin{aligned} S &\rightarrow SS, \\ S &\rightarrow (S), \\ S &\rightarrow () \end{aligned}$$

Notacija: oznaka za alternativu, umesto navođenja dva pravila ("dve strelice")

$$\begin{aligned} S &\rightarrow aBC \\ S &\rightarrow aSBC \\ CB &\rightarrow CZ \\ CZ &\rightarrow WZ \\ WZ &\rightarrow WC \\ WC &\rightarrow BC \\ aB &\rightarrow ab \\ bB &\rightarrow bb \\ bC &\rightarrow bc \\ cC &\rightarrow cc \end{aligned}$$

Gramatika	Jezika	Mašina	Pravila
T0	rekurzivno nabrojiv	tjuringova mašina	$\gamma \rightarrow \alpha$
T1: CSG	kontekstno osetljiv	linearno ograničeni automati	$\alpha A \beta \rightarrow \alpha \gamma \beta$
T2: CFG	kontekstno slobodan	potisni automati	$A \rightarrow \alpha$
T3: Reg	regularan	konačni automati	$A \rightarrow a$ ili $A \rightarrow aB$

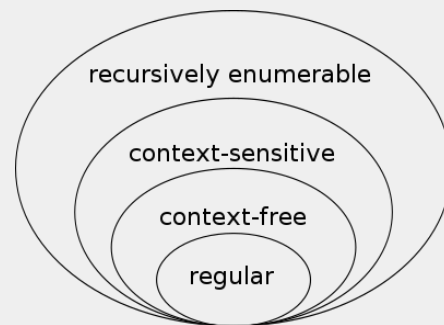


Kojoj klasi jezika pripadaju sledeći jezici?

$$L = \{a^n b^m \mid n, m \geq 1\}$$

$$L = \{a^n b^n \mid n \geq 1\}, \quad L = \{a^n b^m c^{n+m} \mid n, m \geq 1\}$$

$$L = \{a^n b^n c^n \mid n \geq 1\}, \quad L = \{a^n b^m c^{nm} \mid n, m \geq 1\}$$



Šta su leksičke, sintaksne i semantičke greške?

Source code	SC		Programski jezici (3-4)
Leksika	SC + Gramatika L -> Tokeni	Tabela simbola, AST kreiranje	Formalni jezici
Sintaksa	Gramatika S -> AST		Konačni automati
Semantika	AST -> AST'	Tabela simbola, AST korišćenje	Heš tabele, stabla, grafovi – SPA
Generisanje IR	AST' -> IR		Optimizacije (koda/međukoda, lokalne/globalne)
Optimizacije	IR -> IR'		Razvojni alati
Generisanje koda	IR' -> TC		Pomoćni alati
Optimizacije	TC -> TC'		Dizajn softvera i razvojni procesi
Target code	TC		Teorija – Oblast: PL

Leksička analiza - **Skener**



```
if (x == 5)
{
    y = 3; //komentar
}
```

if		(x		==		5)		{		y		=		3	;				}	
----	--	---	---	--	----	--	---	---	--	---	--	---	--	---	--	---	---	--	--	--	---	--

lekseme

```
<KEYW, IF, 1> <LPAREN,, 1> <IDENT, x, 1>  
<ROP, EQ, 1> <NUM, 5, 1> <RPAREN,, 1>  
<LBRAC,, 2> <IDENT, y, 3> <ASSIGN,, 3>  
<SC,, 3> <RBRAC,, 4>
```

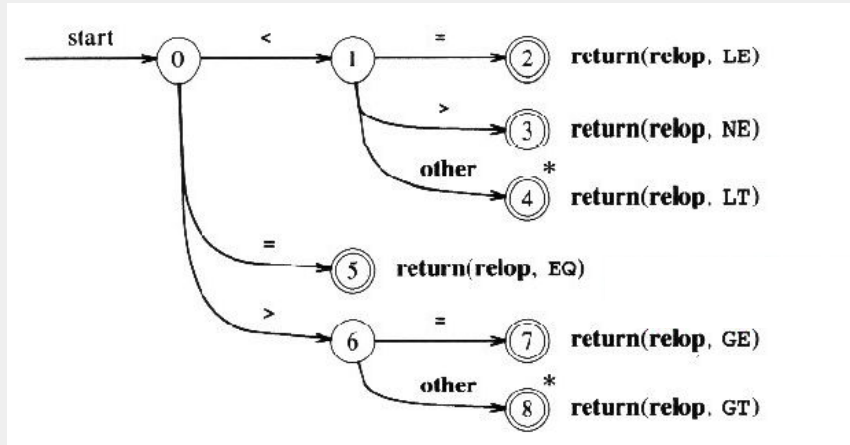
tokeni

Kako implementirati skener?

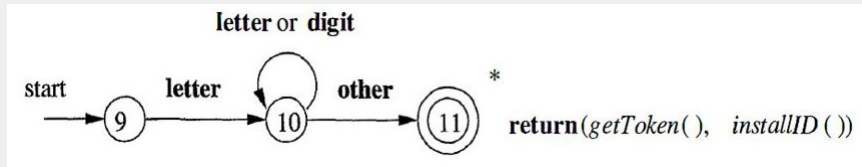
- čitamo string sa ulaza, karakter po karakter
 - prepoznavamo lekseme
 - (kreiramo tokene)
 - informišemo o rezultatu (ok // greška)
-
- **ručno** (while, if/switch, lookahead) → proizvoljno
 - **alat** (flex, lex, antlr...) → treba nam neka formalizacija jezika

Na šta nas ovo asocira?

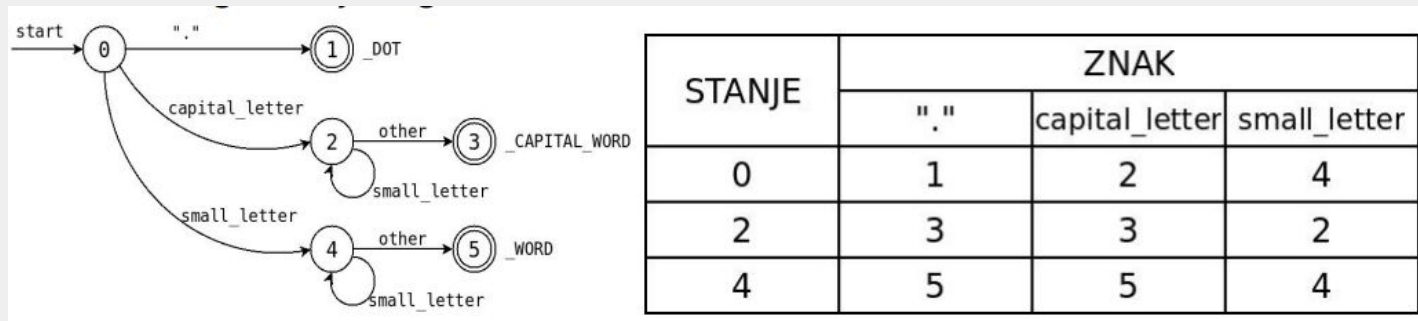
- "prepoznavamo"
- "string"
- "ok vs error"



* pročitani simbol se uzima u obzir za sledeći token



ako pp da je KW specijalan slučaj identifikatora



regularni izrazi → tabela prelaza

generisani skener - simulator konačnog automata koji koristi tabelu prelaza

Primer

```
[ \t\n]+      { /* skip */ }

"if"         { return _IF; }
"else"       { return _ELSE; }
"return"     { return _RETURN; }

"int"        { return _TYPE; }
"unsigned"   { return _TYPE; }

"("          { return _LPAREN; }
")"          { return _RPAREN; }
"{"          { return _LBRACKET; }
"}"          { return _RBRACKET; }
";"          { return _SEMICOLON; }
"="          { return _ASSIGN; }

"+"         { return _AROP; }
"-"         { return _AROP; }

"<"         { return _RELOP; }
"=="        { return _RELOP; }

[a-zA-Z][a-zA-Z0-9]* { return _ID; }
[+-]?[0-9]{1,10}     { return _INT_NUMBER; }
[0-9]{1,10}[uU]      { return _UINT_NUMBER; }

\\/\\..*      { /* skip */ }
.             { printf("LEXICAL ERROR"); }
```

↑
regularni izrazi

↑
tokeni

Zadatak

Kako izgledaju formalne specifikacije leksike nekih popularnih jezika?
Kako su implementirani skeneri?

U C-u, šta je rezultat za ---a u okviru različitih izraza? Greška ili ne?
Šta neko može očekivati da ovde bude rezultat?
Šta (neki) kompajler kaže? Da li je vaša pretpostavka bila tačna?

Šta je programski jezik?

Pogledati: <https://esolangs.org>
Implementirati neki?

Sintaksna analiza - Parser

string
leksička pravila



tokeni
greške

```
if (8k == 5)
{
    y = 3 * 2;
}
```

tokeni
sintaksna pravila



AST
greške

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

$S \rightarrow e \mid SS \mid (S)$

Primer konstrukta koji nam je potreban za PL: Balansiranost zagrada

- Prvo pravilo - zaustavi rekurziju
- Drugo pravilo - kad se spoje dva stringa koja su balansirana opet se dobija balansirani string.
- Treće pravilo - balansirani string u zagradama je opet balansiran

→ rekurzivno razmišljanje - vrlo često kod formiranja pravila gramatike PL

ostale zagrade: { }, < >, []

Zašto ovaj jezik nije regularan? Dokaz?

Regularni izrazi (jezici) nisu više dovoljni da se opišu pravila sintakse mnogih programskih jezika.

Nisu dovoljno ekspresivni.

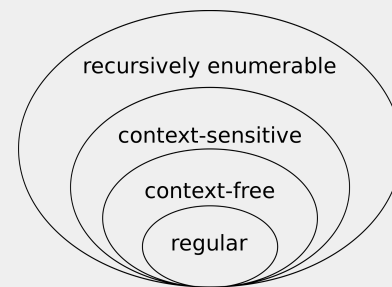
Veći deo sintakse poznatih programskih jezika je kontekstno-slobodan.

```
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*



`assignment_stmt` → ID ASSIGN `num_exp` SC

`num_exp` → exp

`num_exp` → `num_exp` PLUS exp

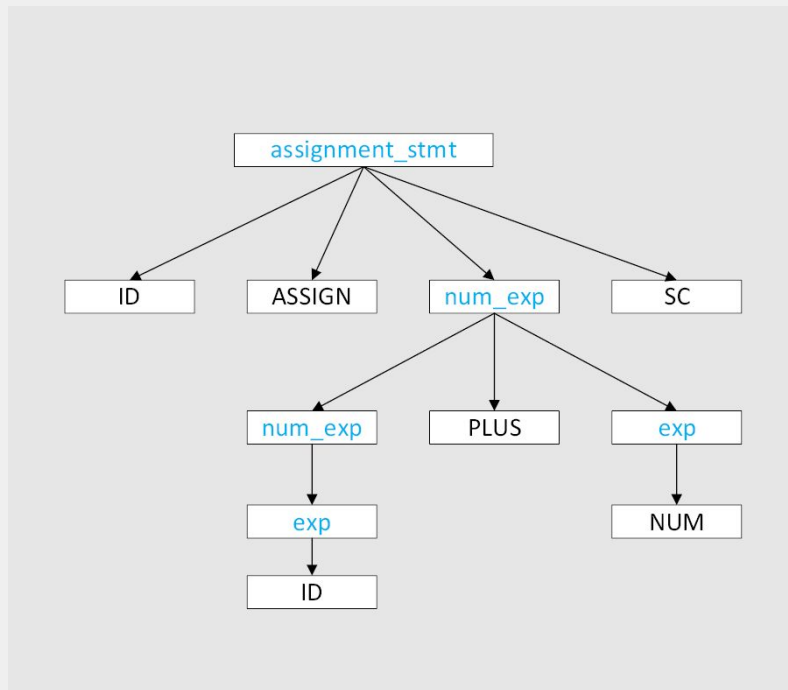
`num_exp` → `num_exp` MINUS exp

exp → NUM

exp → ID

a = b + 3;

w = ID ASSIGN ID PLUS NUM SC



Čitajući listove s leva na desno dobijamo polazni string

Kako bismo zapisali izvođenje?

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, \dots, X_n predstavlja pravilo izvodjenja $Y \rightarrow X_1 X_2 \dots X_n$

Struktura podataka: stablo
Zašto je značajno?

Programski jezici - prevođenje

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.

CFG gramatika → stablo parsiranja

Source code → struktura podataka

Obilasci:

- semantička pravila; sistem tipova
- analiza koda; optimizacije
- generisanje koda

Postoje problemi.

U prethodnom primeru prilično jasan odabir pravila.

Nekad, prilikom izvođenja, možemo krenuti različitim “putevima”.

Posmatrajmo sledeću gramatiku:

$E \rightarrow \text{num}$

$E \rightarrow E + E$

$5 + 3 + 1$

$w = \text{num plus num plus num}$

w - polazna reč

Traži se odgovor na pitanje: Da li pripada jeziku koji generiše gramatika?

→ Sintaksno ispravan ili neispravan program.

Reč pripada jeziku ako postoji izvođenje od početnog pojma.

$E \rightarrow \text{num}$

$E \rightarrow E + E$

$5 + 3 + 1$

w = num plus num plus num

Leftmost derivation (najlevlje izvođenje, izvođenje s leva)

Izvođenje u kom se uvek primenjuje pravilo na prvi pojam sa leve strane

Rightmost derivation (najdešnje izvođenje, izvođenje s desna)

Izvođenje u kom se uvek primenjuje pravilo na prvi pojam sa desne strane

Postoje izvođenja koja ne prate ni jedno od ove dve heuristike.

Leftmost

$E \rightarrow \text{num}$

$E \rightarrow E + E$

5 + 3 + 1

w = num + num + num

E

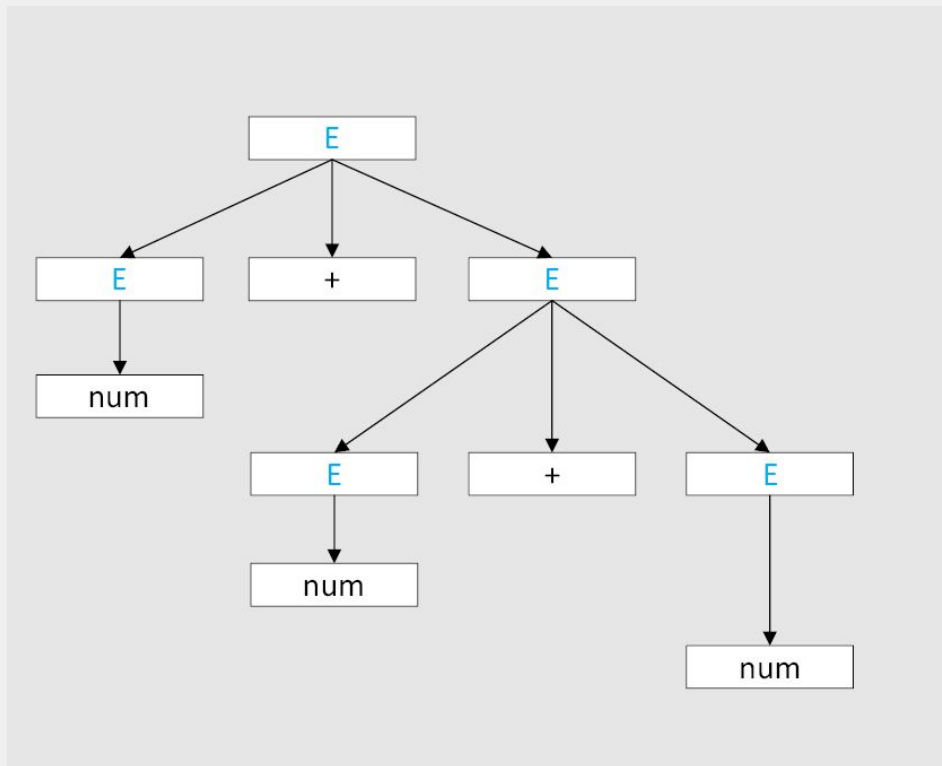
$\Rightarrow E + E$

$\Rightarrow \text{num} + E$

$\Rightarrow \text{num} + E + E$

$\Rightarrow \text{num} + \text{num} + E$

$\Rightarrow \text{num} + \text{num} + \text{num}$



Rightmost

$E \rightarrow \text{num}$

$E \rightarrow E + E$

$5 + 3 + 1$

$w = \text{num} + \text{num} + \text{num}$

E

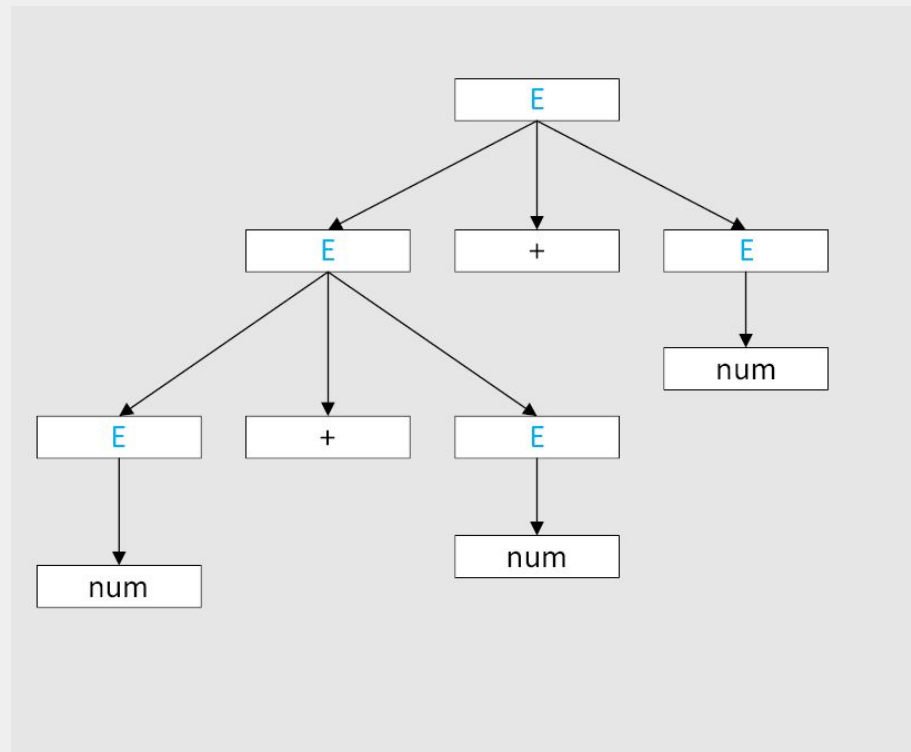
$\Rightarrow E + E$

$\Rightarrow E + \text{num}$

$\Rightarrow E + E + \text{num}$

$\Rightarrow E + \text{num} + \text{num}$

$\Rightarrow \text{num} + \text{num} + \text{num}$



Ni L ni R

$E \rightarrow \text{num}$

$E \rightarrow E + E$

5 + 3 + 1

w = num + num + num

E

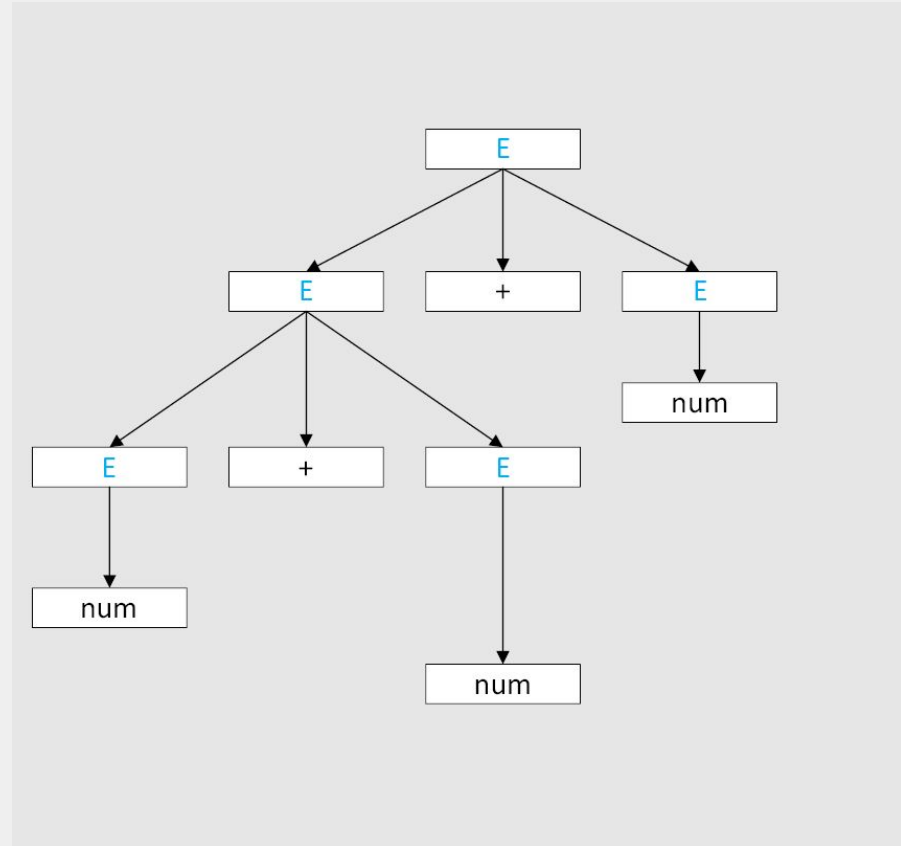
$\Rightarrow E + E$

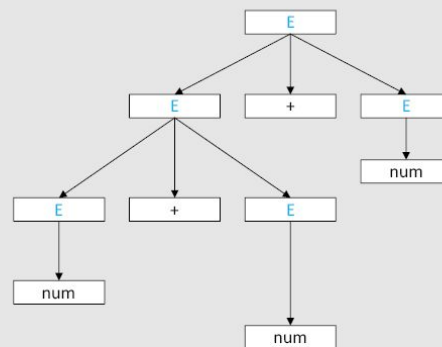
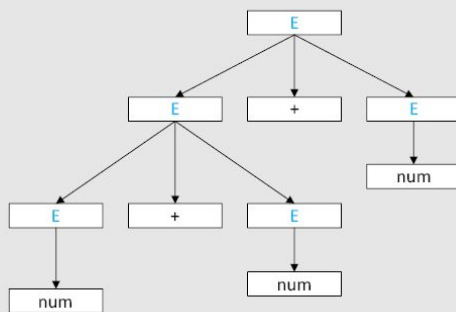
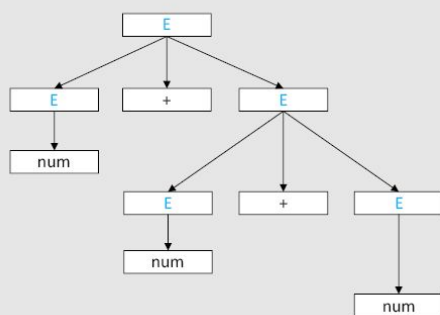
$\Rightarrow E + \text{num}$

$\Rightarrow E + E + \text{num}$

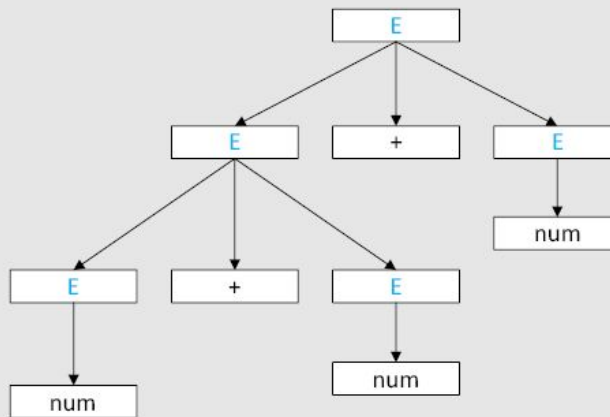
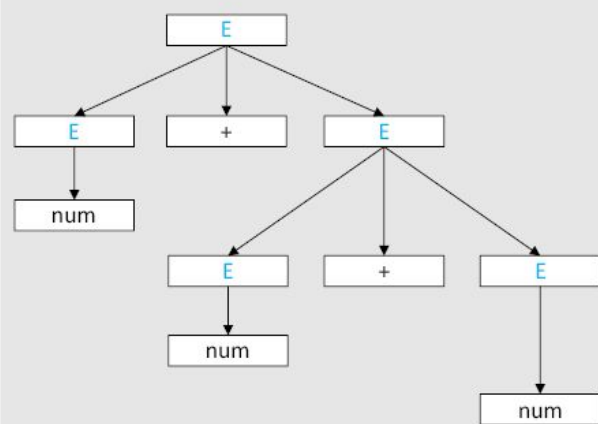
$\Rightarrow \text{num} + E + \text{num}$

$\Rightarrow \text{num} + \text{num} + \text{num}$





Da mi implementiramo, koje stablo bi nam odgovaralo?



Može se primetiti da stablo modelira **asocijativnost** operacija

$num + (num + num)$

$(num + num) + num$

$$5 + 3 + 1$$

Ako je operator + levo asocijativan operand 3 će biti preuzet od strane levog + operatora
 $(5 + 3) + 1$

Ako je operator + desno asocijativan operand 3 će bit preuzet od strane desnog + operatora
 $5 + (3 + 1)$

Može biti značajno kod evaluacije.

Šta da nije bio operator sabiranja?

Posmatrajmo sledeću gramatiku

(na trenutak zaboravimo na to što imamo dvostruku rekurziju)

$E \rightarrow \text{num}$

$E \rightarrow E + E$

$E \rightarrow E - E$

$5 - 3 + 1$

$w = \text{num} - \text{num} + \text{num}$

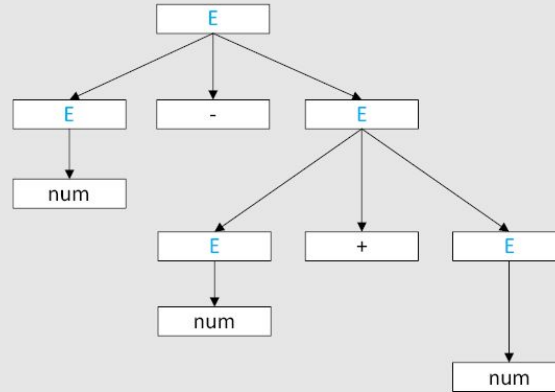
$E \rightarrow \text{num}$

$E \rightarrow E + E$

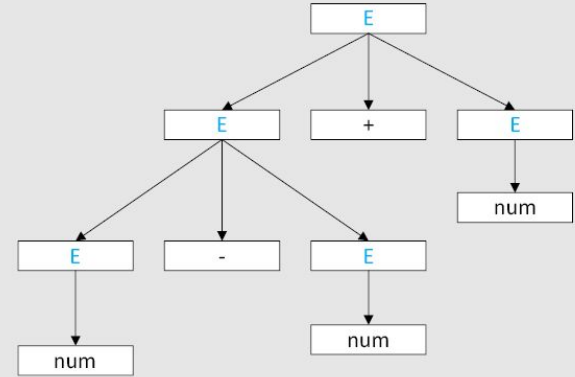
$E \rightarrow E - E$

$5 - 3 + 1$

$w = \text{num} - \text{num} + \text{num}$



$5 - (3 + 1)$



$(5 - 3) + 1$

Posmatrajmo sad sledeću gramatiku:

$E \rightarrow \text{num}$

$E \rightarrow E + E$

$E \rightarrow E * E$

$5 + 3 * 2$

$w = \text{num} + \text{num} * \text{num}$

$E \rightarrow \text{num}$

$E \rightarrow E + E$

$E \rightarrow E * E$

$5 + 3 * 2$

$w = \text{num} + \text{num} * \text{num}$

leftmost + prvo pravilo 2

E

$\Rightarrow E + E$

$\Rightarrow \text{num} + E$

$\Rightarrow \text{num} + E * E$

$\Rightarrow \text{num} + \text{num} * E$

$\Rightarrow \text{num} + \text{num} * \text{num}$

leftmost + prvo pravilo 3

E

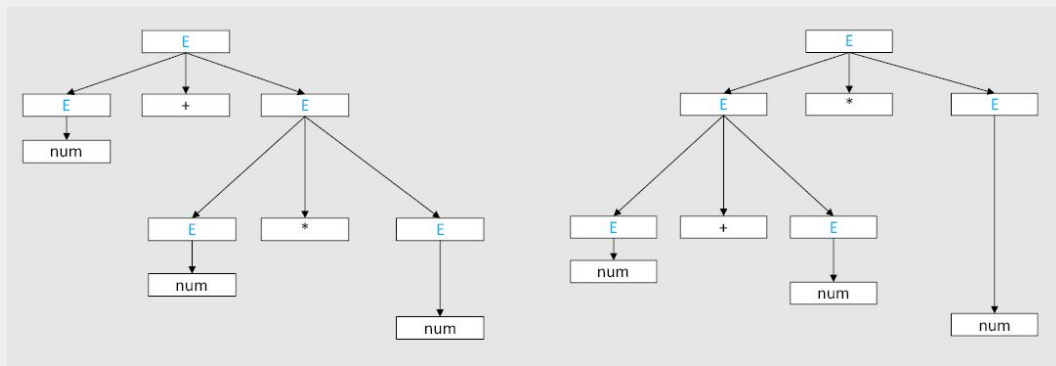
$\Rightarrow E * E$

$\Rightarrow E + E * E$

$\Rightarrow \text{num} + E * E$

$\Rightarrow \text{num} + \text{num} * E$

$\Rightarrow \text{num} + \text{num} * \text{num}$



$5 + (3 * 2)$

$(5 + 3) * 2$

Ako obezbedimo da su osnovni aritmetički operatori levo asocijativni to nam ne rešava ceo problem. Potrebno je modelirati **prioritet operatora**.

Moramo obezbediti asocijativnost.

Moramo obezbediti prioritet.

Različita stabla parsiranja mogu odgovarati istom stringu.

Kažemo da je gramatika **dvosmislena** (višeznačna) ako za neki polazni string postoje bar dva različita stabla parsiranja.

Ako je nedvosmislena ima samo jedno LM i jedno RM za svaki string.

Kako da rešimo i zašto bismo rešavali?

- Možemo promeniti gramatiku
- Koristeći neke mehanizme/ideje specifične za problem koji rešavamo
- Koristeći mogućnosti alata za generisanje parsera (shortcut)

Ne postoji univerzalni algoritam za detekciju dvosmislene gramatike ili njenog pretvaranja u gramatiku koja nije dvosmislena.

$E \rightarrow \text{num}$

$E \rightarrow E + E$

$E \rightarrow \text{num}$

$E \rightarrow E + \text{num}$

$5 + 3 + 1$

$w = \text{num} + \text{num} + \text{num}$

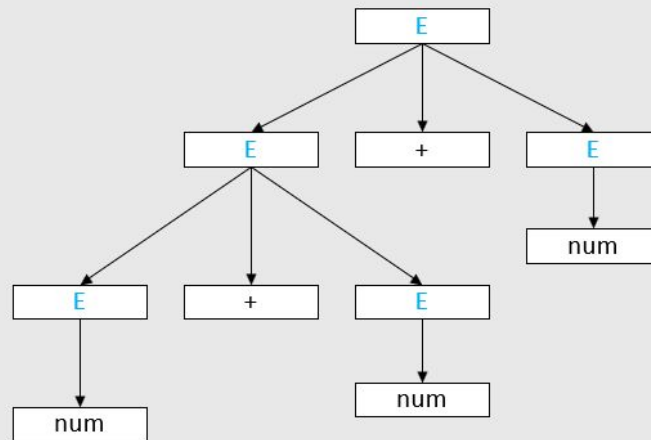
E

$\Rightarrow E + \text{num}$

$\Rightarrow E + \text{num} + \text{num}$

$\Rightarrow \text{num} + \text{num} + \text{num}$

Obratiti pažnju: definisano kao lista



Ideja:

5 + 3 * 2 + 4 * 1 ...

Posmatramo kao sabirke

1. $E \Rightarrow \text{num}$

2. $E \Rightarrow E + E$

3. $E \Rightarrow E * E$

1. $E \rightarrow E + T$

2. $E \rightarrow T$

3. $T \rightarrow T * \text{num}$

4. $T \rightarrow \text{num}$

5 + 3 * 2

w = num + num * num

E

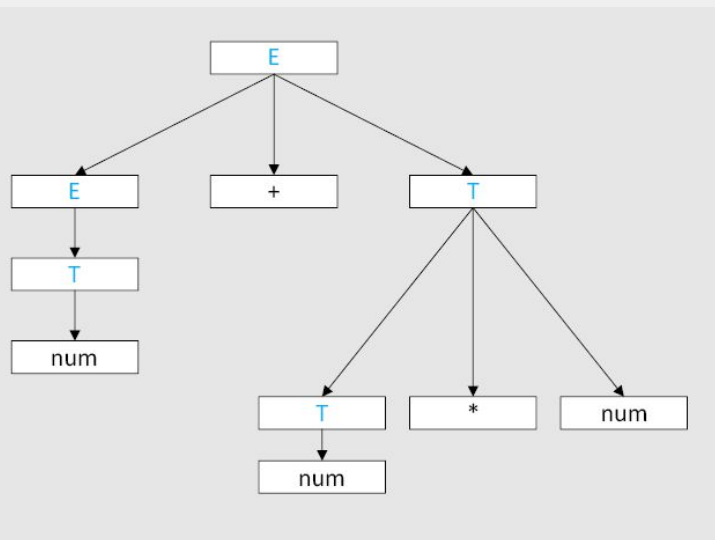
$\Rightarrow E + T$

$\Rightarrow T + T$

$\Rightarrow \text{num} + T$

$\Rightarrow \text{num} + T * \text{num}$

$\Rightarrow \text{num} + \text{num} * \text{num}$



Dvosmislenost je osobina gramatike, ne jezika!

Jezik je (nasledno, izvedeno) dvosmislen ako su sve odgovarajuće gramatike dvosmislene, ako ne postoji nijedna nedvosmislena.

Zadatak

Napisati pravila gramatike sintakse nekog jezika (C++, Java, Python...) za različite konstrukte tog jezika. Na primer: for, while, dodela, deklaracija promenljive, definicija funkcije,...

Kako bi izgledale produkcije u gramatici za if-else u jeziku C?

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

Šta je problem?

Kako uopšte izgledaju gramatike za neki programski jezik?

Kao implementirati parser za neki programski jezik?

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

jednostavnije za direktnu implementaciju // potrebno ili makar poželjno razmotriti osobine CFG

Uzlazno parsiranje (bottom-up)

Pokušavamo da od ulaznog niza simbola redukcijom dobijemo početni pojam.

Akcije: Desna strana pravila se zamenjuje levom

širu klasu gramatika podržavaju // potrebno definisati i razmotriti odgovarajuće automate

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

x = y + 5;

Silazno parsiranje (top-down)

Pokušavamo da od početnog pojma izvedemo ulazni niz simbola prateći pravila produkcije.

Akcije: Leva strana pravila se zamenjuje desnom

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)
- Šta su problemi?

Uzlazno parsiranje (bottom-up)

Pokušavamo da od ulaznog niza simbola redukcijom dobijemo početni pojam.

Akcije: Desna strana pravila se zamenjuje levom

`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

x = y + 5;

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.

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)

Primer

```
text
  → sentence
  → text sentence

sentence
  → words dot

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

pp: capital_word, word i dot su terminali

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

„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

text

→ *sentence*

→ *text sentence*

sentence

→ *words DOT*

words

→ *CAPITAL_WORD*

→ *words WORD*

→ *words CAPITAL_WORD*

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
<i>words</i>

„Ovo je tekst.“

WORD DOT EOF

text

- *sentence*
- *text sentence*

sentence

- *words DOT*

words

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

WORD
<i>words</i>

„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
<i>words</i>

„Ovo je tekst.“

DOT EOF

text

- *sentence*
- *text sentence*

sentence

- *words* DOT

words

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

WORD
<i>words</i>

„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
<i>words</i>

„Ovo je tekst.“

EOF

text

- *sentence*
- *text sentence*

sentence

- *words DOT*

words

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

DOT
<i>words</i>

„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

text

„Ovo je tekst.“

text

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

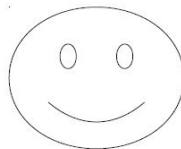
sentence

- ***words DOT***

words

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

EOF
<i>text</i>



program

```
: function_list  
;
```

function_list

```
: function  
| function_list function  
;
```

function

```
: type _ID _LPAREN parameter _RPAREN body  
;
```

...

statement_list

```
: /* empty */  
| statement_list statement  
;
```

statement

```
: compound_statement  
| assignment_statement  
| if_statement  
| return_statement  
;
```

...