CS4031
Compiler Construction
Lecture 4

Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

Practice: define the lex program for the given code lexems.

```
int main ( )
{
int x, a=2, b=3, c=5;
x = a+b*c;
printf(“the value of x is %d”, x);
return 0;
}
```

Practice: define the lex program for the given code lexems.

```
int main ( )
{
a = b + d;
int a, b, d;
printf(“sum of given is %d”; a);
}
```

Practice: define the lex program for the given code lexems.

```
int main ( )
{
a = b++++++===--;
}
```

Syntax Analysis

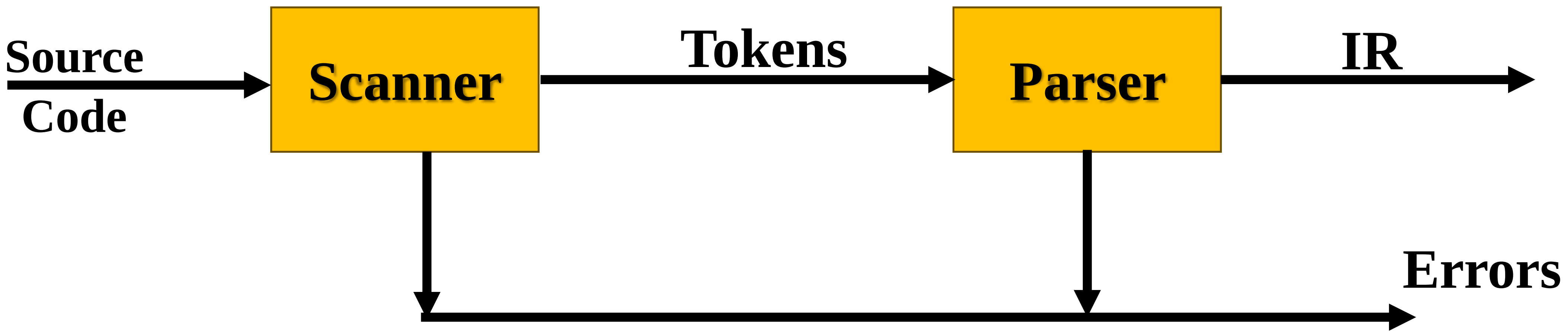
- Syntax Analyzer creates the syntactic structure of the given source program.
- This syntactic structure – parse tree.
- Syntax analyzer is also known as parser
- The syntax analyzer (parser checked whether a given source program satisfies the rules of implied for context free grammar or nor.
- If it satisfies, the parser creates the parse tree of the program.
- Otherwise the parser gives the error messages.

Syntax Trees

- A syntax tree represents the syntactic structure of tokens in a program defined by the grammar of the programming language.

- Id1 = id2 + ide * 60;

Front End Parser



Front End Parser

- Checks the stream of words and **their parts of speech for grammatical correctness**
- Determines if the input is **syntactically well formed**
- Guides **context-sensitive (“semantic”) analysis (type checking)**
- Builds IR for source program

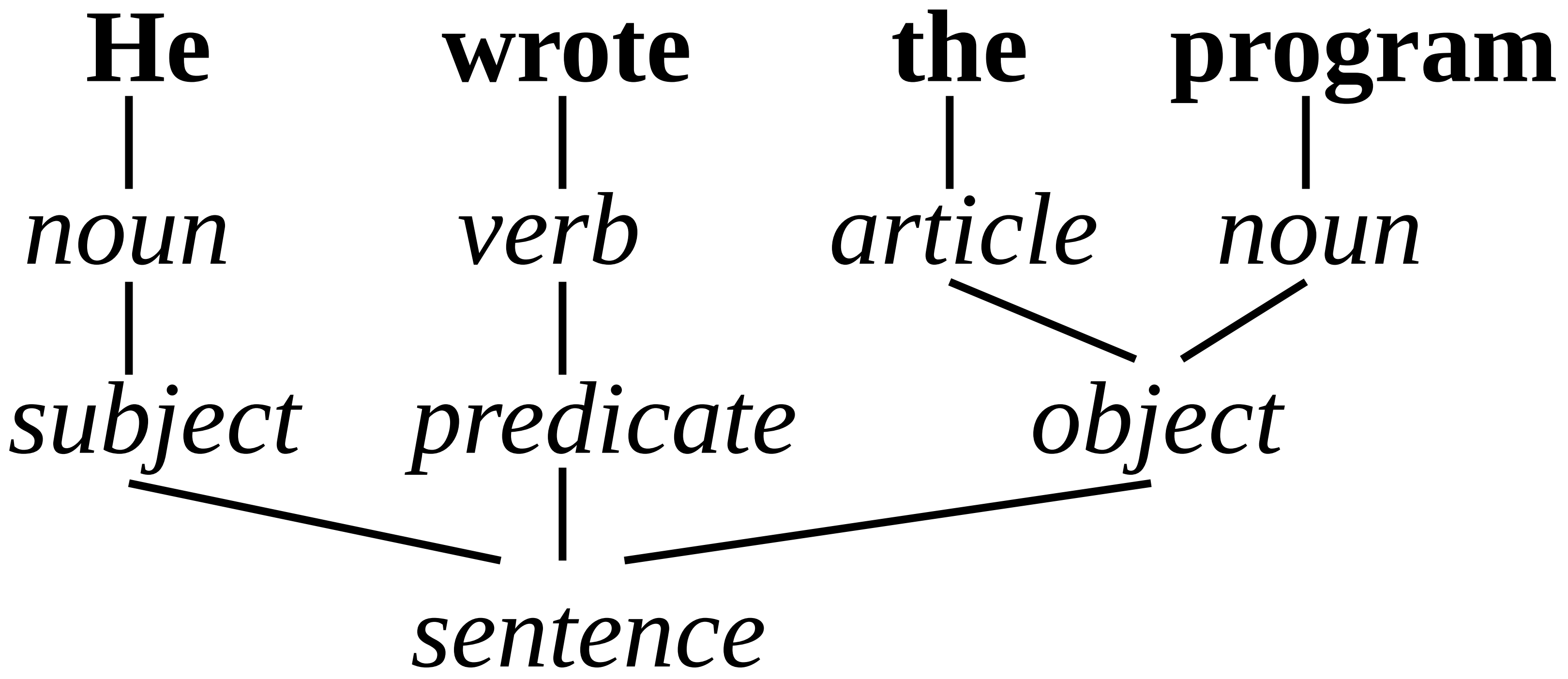
Syntactic Analysis

- Natural language analogy: consider the sentence

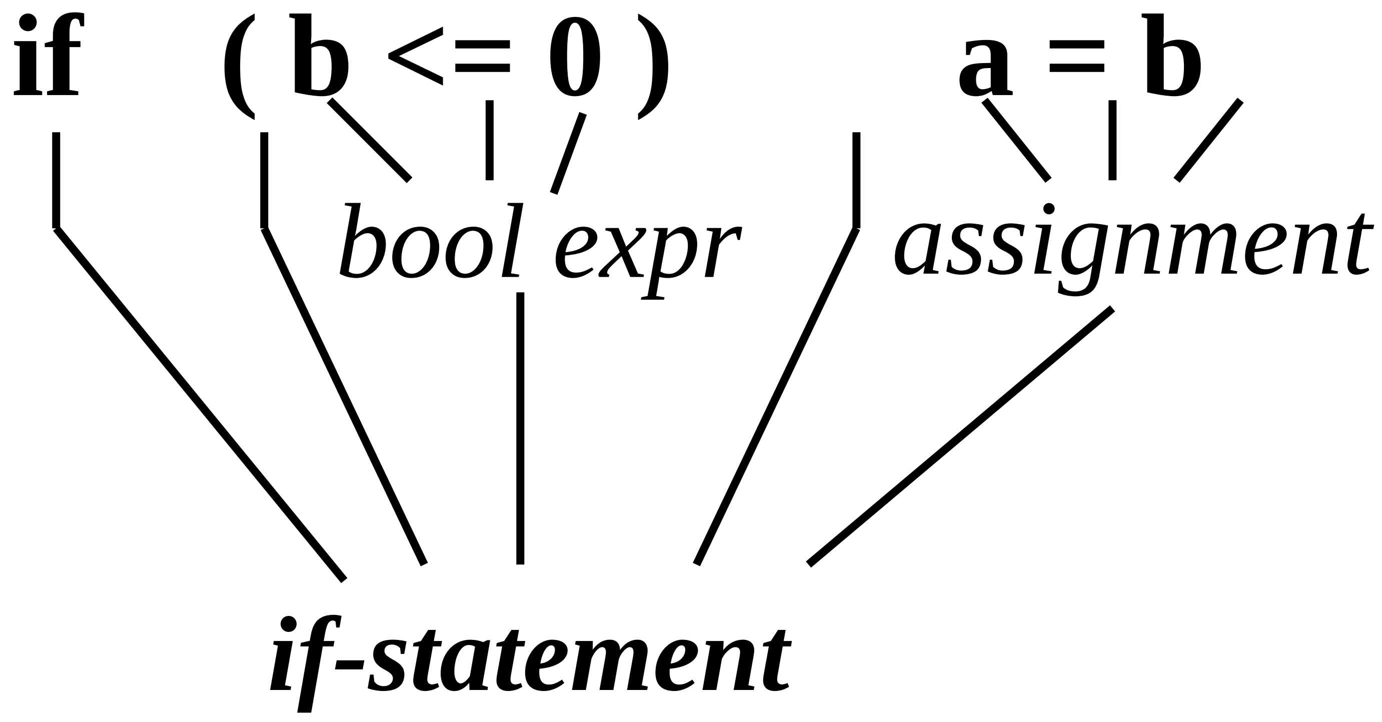
He wrote the program

Syntactic Analysis

- Natural language analogy



Syntax Analysis in Programming Language



Syntactic Analysis

```
Int *compiler(int i, int j))
{
for(k = 0; i   j;)
fi ( i > j)
return j;
}
```

Syntactic Analysis: Identify the error

```
Int *compiler(int i, int j))        Extra Parenthesis
{
for(k = 0; i   j;)  Missing Expression
fi ( i > j)
                Not a keyword
return j;
}
```

Semantic Analysis

```
int* compiler(int i, int j )
{
for(k=0; i<j; j++)
If( i <  j – 2 )
float sum;
sum = sum + i;
return sum;
}
```

Role of Parser

- Not all sequences of tokens are program.
- Parser must distinguish between valid and invalid sequences of tokens.

What we need

- An expressive way to describe the syntax
- An acceptor mechanism that determines if input token stream satisfies the syntax

Study of Parsing

- Parsing is the process of discovering **a *derivation*** for some sentence
- Mathematical model of syntax – a grammar ***G*.**
- Algortihm for testing membership in ***L(G)*.**

Limitations of Regular Languages

- Finite Automation can’t remember number of times it has visited a particular state

Grammar

- Grammar is basically defi ned as a set of 4-tuple (V, T, P, S), where
- V is set of nonterminals (variables),
- T is set of terminals (primitive symbols),
- P is set of productions (rules),
- which govern the relationship between nonterminals and terminals, And S is start symbol with which strings in grammar are derived.

Terminals Symbol

- Terminals symbols: these are represented by
- Lower case letters of alphabet like a, c, z, etc.
- Operator symbols like +, –, etc.
- Punctuation symbols like (, } , , etc.
- Digits like 0…9, etc.
- Bold face strings like int, main, if, else etc.

Non-Terminals Symbols

- Nonterminal symbols: these are represented by
- Upper case letters of alphabet like A, C, Z, etc.
- Letter S is the start symbol.
- Lower case strings like expr, stmt, etc.

Types of Grammar – Chmosky Hierarchy

- The Chomsky hierarchy consists of four levels, which are
- Type 0 (unrestricted),
- Type 1 (context-sensitive),
- Type 2 (context-free),
- Type 3 (regular)
- The levels are based on the complexity of the formal grammar needed to generate a language.

Type 0 Grammar

- These grammars include all formal grammars. In URG, all the productions are of the form $\alpha \rightarrow \beta$, where $\alpha\ and\ \beta$ may have any number of terminals and nonterminals.
- No restrictions on either side of productions.
- Every grammar is included in it if it has at least one nonterminal on the left hand side.

Example:

- The language specified by $L(G) = \{a^{2i} \mid I > = 1\}$ is an unrestricted grammar. This grammar is also called phrase structured grammar.
- The grammar is represented by the following productions:

$$\begin{aligned}
S &\rightarrow A\,C\,a\,B \\
C\,a &\rightarrow a\,a\,C \\
C\,B &\rightarrow D\,B \\
C\,B &\rightarrow E \\
a\,D &\rightarrow D\,a \\
A\,D &\rightarrow A\,C \\
a\,E &\rightarrow E\,a \\
A\,E &\rightarrow \varepsilon
\end{aligned}$$

Type 1 Grammar: Context Sensitive grammar

- These grammars define the context sensitive languages. In CSG, all the productions of the form $\alpha \rightarrow \beta$ where $|\alpha| \leq |\beta|$, α and β may have any number of terminals and non terminals.
- These grammars can have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with A as nonterminal and α, β and γ are strings of terminals and non terminals.
- We can replace A by γ, where A lies between α and β. Hence, the name context sensitive grammar. The strings α and β may be empty, but γ must be nonempty.

Example

- The language specified by L(G) = $\{a^n b^n | \ n >= 1\}$ is a context sensitive grammar.

$$\begin{aligned} S &\rightarrow S\,B\,C \mid a\,C \\ B &\rightarrow a \\ C\,B &\rightarrow B\,C \\ Ba &\rightarrow aa \\ C &\rightarrow b \end{aligned}$$

Type 2 Grammar: Context Free Grammar

- These grammars define the context free languages. These are defined by rules of the form $\alpha \rightarrow \beta$ with $|\alpha| \leq |\ \beta$ where $|\alpha| = 1$ and is a nonterminal and β is a string of terminals and non terminals.
- We can replace α by β regardless of where it appears.

Example:

- The language specified by L(G) = $\{a^n b^n | \text{ n} > = 1\}$ is a context sensitive grammar.

$$S \rightarrow aSb \mid \varepsilon$$

Type 3 Grammar: Regular Grammar

- These grammars generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left hand side.
- The right hand side consists of either a single terminal or a string of terminals with a single nonterminal on the left or the right end. Here rules can be of the form $A \rightarrow a\ B \mid a$ or $A \rightarrow Ba \mid a$.

Example:

- Right linear grammar: $A \rightarrow a\ A \mid a$
- Left linear grammar: $A \rightarrow A\ a \mid a$

Backus-Naur Form (BNF)

- BNF stands for **Backus-Naur Form**. It is used to write a formal representation of a context-free grammar. It is also used to describe the syntax of a programming language.
- BNF uses a range of symbols and expressions to create **production rules**. A simple BNF production rule might look like this:

<digit> ::= 0|1|2|3|4|5|6|7|8|9

BNF

- It is a formal mathematical way to describe a language
- It is used describe the syntax of programming language
- It consist of
  - A set of **terminal symbols**
  - A set of **non-terminal symbols**
  - A set of production rules of the form
  - **Left-hand-side ::= right-hand-side**
- The meaning of the production rule is that the **non-terminal on the LHS may be replaced by the expression on the RHS.**

Example

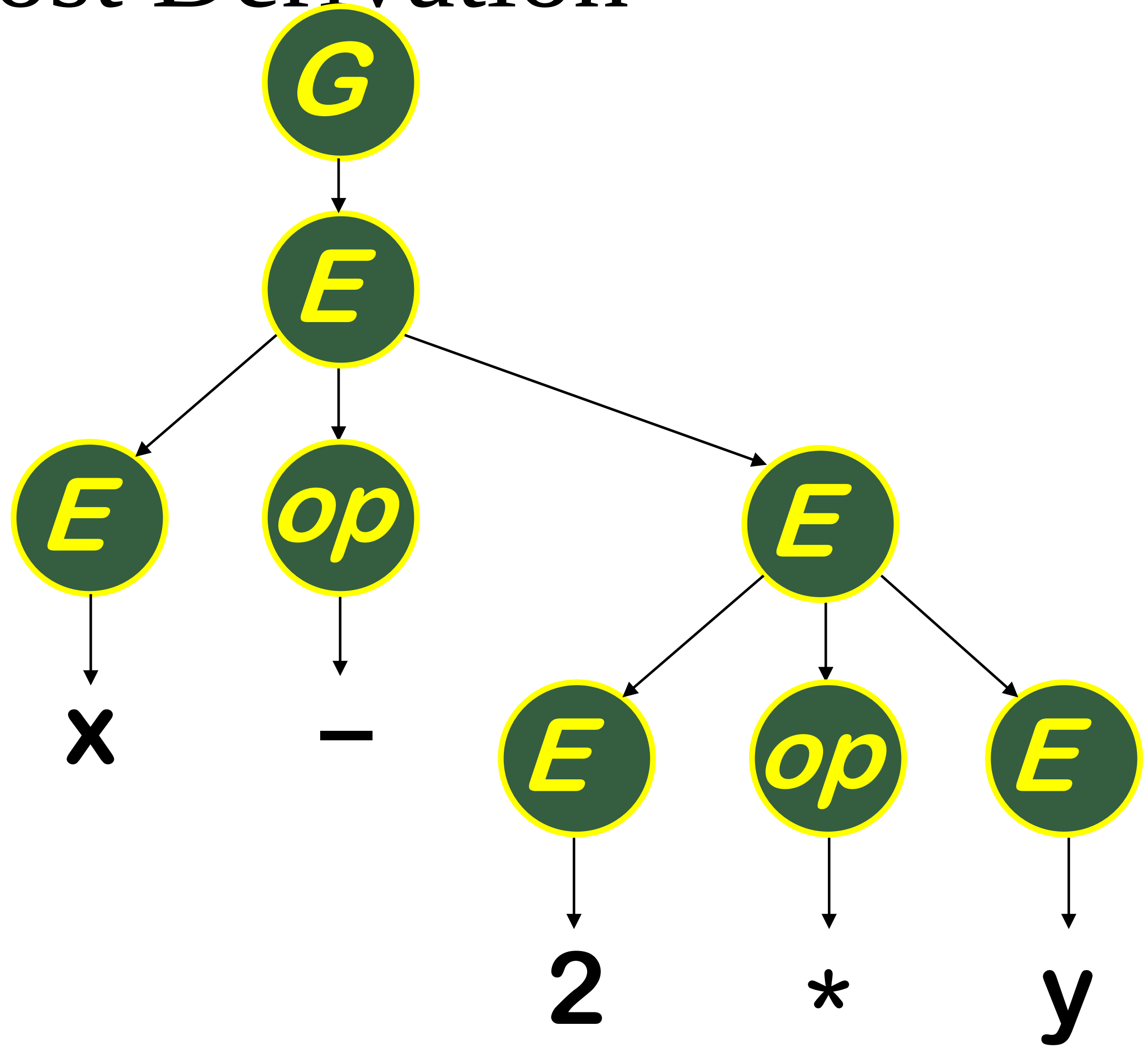
- Let us use the expression grammar to derive the sentence

$$x - 2 \ * y$$

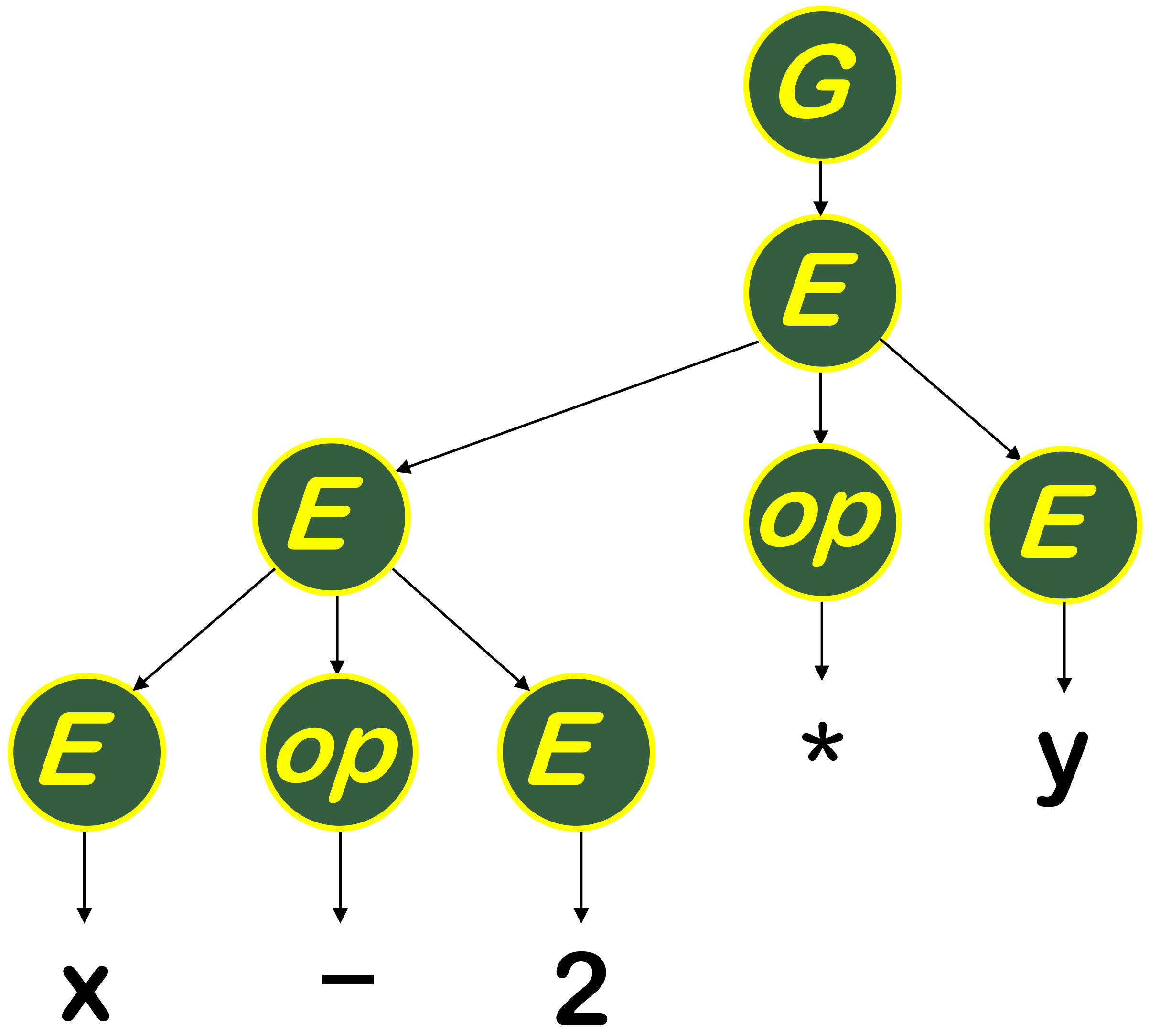
Solution

```
<expression>  ::= <term> | <expression> “-” <term>
<term>        ::= <factor> | <term> “*” <factor>
<factor>      ::=  “x” | “y” | <number>
<number>      ::= “2”
```

Left Most Derivation



Right Most Derivation



Problems in Grammars

- Left Factoring
- Left Recursion
- Right Recursion
- Left Most Derivation
- Right Most Derivation

Left Factoring

- Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.
- Example:

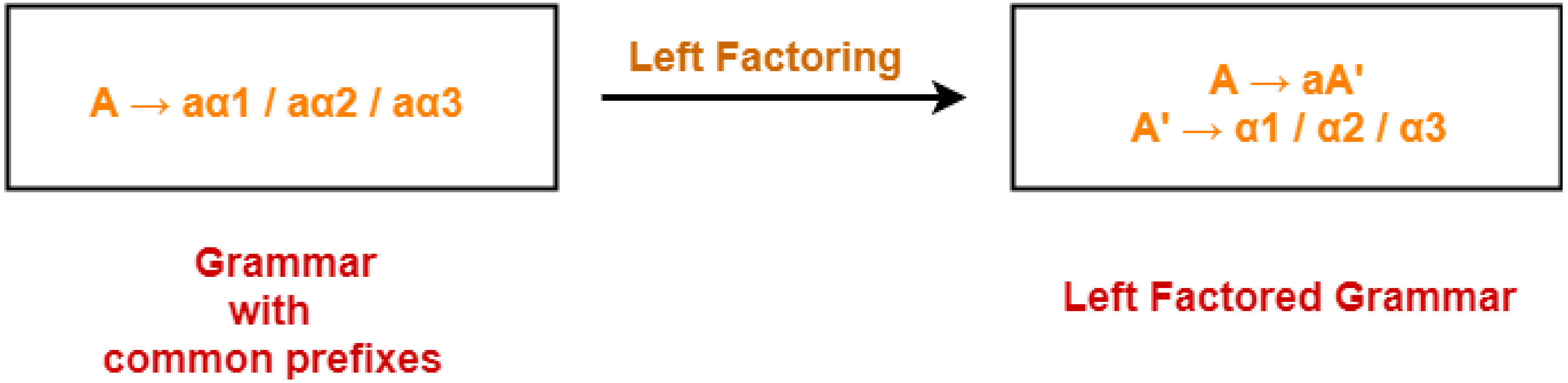
$$A \rightarrow \alpha\beta1 \ / \ \alpha\beta2 \ / \ \alpha\beta3$$

This kind of grammar creates a problematic situation for Top down parsers.

Top down parsers can not decide which production must be chosen to parse the string in hand.

To remove this confusion, we use left factoring.

Example:


A → aα1 / aα2 / aα3
Left Factoring
A → aA'
A' → α1 / α2 / α3
Grammar with common prefixes
Left Factored Grammar

Left Recursion

$$A \rightarrow A\alpha \,/\, \beta$$

Left recursion is considered to be a problematic situation for Top down parsers.

Therefore, left recursion has to be eliminated from the grammar.

Left-Recursion Elimination

we can eliminate left recursion by replacing the pair of productions with-

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \,/\, \in$

Ambiguous to Unambiguous Grammar

- Any grammar which consist of more than one derivation tree is called ambiguous grammar.

Removal Ambiguity

- Two approaches are used to remove the ambiguity

1. Associativity Constraint
2. Precedence Constraint

Removal Ambiguity

Precedence Constraint

The precedence constraint is implemented using the following rules-

- The level at which the production is present defines the priority of the operator contained in it.
- The higher the level of the production, the lower the priority of operator.
- The lower the level of the production, the higher the priority of operator.

Associativity Constraint Examples:

Consider the following Grammar

$$E \longrightarrow E + T \mid T$$
$$T \longrightarrow T * F \mid F$$
$$F \longrightarrow G \wedge F \mid G$$
$$G \longrightarrow id$$

Operator in Grammar	**Precedence**
+	3
*	2
^	1

Removal Ambiguity

- Associativity Constraint
  - If the operator is left associative, induce left recursion in its production.
  - If the operator is right associative, induce right recursion in its production.

- Associativity of operators

• +, -	left to right
• *, /	left to right
• ^	right to left

Associativity Constraint Examples:

Consider the following Grammar

$$E \longrightarrow E + T \mid T$$
$$T \longrightarrow T * F \mid F$$
$$F \longrightarrow G \mathbin{\char`\^} F \mid G$$
$$G \longrightarrow id$$

Operator in Grammar	Associativity either left or right
+	Left Associativity
*	Left Associativity
^	Right Associativity

Example

- $E \rightarrow E + E \mid E * E \mid id$

First and Follow

First and follow sets are needed so that the parser can properly apply the needed production rule at the correct position.

First

First(α) is a set of terminal symbols that begin in strings derived from α.

Consider the production rule-

$$A \rightarrow abc \,/\, def \,/\, ghi$$

Then, we have- First(A) = { a , d , g }

Follow

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

- For the start symbol S, place $ in Follow(S).
- For any production rule $A \longrightarrow \alpha B$,

  $$\text{Follow}(B) = \text{Follow}(A)$$

- For any production rule $A \longrightarrow \alpha B \beta$,

  If $\in \notin \text{First}(\beta)$, then $\text{Follow}(B) = \text{First}(\beta)$

  If $\in \in \text{First}(\beta)$, then $\text{Follow}(B) = \{ \text{First}(\beta) - \in \} \cup \text{Follow}(A)$

Example:

- Consider the following Grammar

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow id \mid (E)$$

Example:

- Consider the following Grammar

$$E \rightarrow E{+}T \mid T$$

$$T \rightarrow T{*}F \mid F$$

$$F \rightarrow id \mid (E)$$

Remove the left recursion

$$E \rightarrow T\,E'$$

$$E' \rightarrow +\,T\,E' \mid \varepsilon$$

$$T \rightarrow F\,T'$$

$$T' \rightarrow {*}F\,T' \mid \varepsilon$$

$$F \rightarrow id \mid (E)$$

Now find the first and follow

Grammar Productions	First	Follow
$E \rightarrow TE'$	{id, (}	
$E' \rightarrow +TE' \mid \epsilon$	$\{+, \epsilon\}$	
$T \rightarrow FT'$	{id, (}	
$T' \rightarrow * FT' \mid \epsilon$	$\{*, \epsilon\}$	
$F \rightarrow id \mid (E)$	{id, (}	

Now find the first and follow

Grammar Productions	First	Follow
$E \rightarrow TE'$	{id, (}	{$,)}
$E' \rightarrow +TE' \mid \epsilon$	{+, ϵ}	{
$T \rightarrow FT'$	{id, (}	
$T' \rightarrow * FT' \mid \epsilon$	{*, ϵ}	
$F \rightarrow id \mid (E)$	{id, (}	

Now find the first and follow

Grammar Productions	**First**	**Follow**
$E \rightarrow TE'$	{id, (}	{$,)}
$E' \rightarrow +TE' \mid \epsilon$	{+, ϵ}	{$,)}
$T \rightarrow FT'$	{id, (}	{+, $,)}
$T' \rightarrow * FT' \mid \epsilon$	{*, ϵ}	{+, $,)}
$F \rightarrow id \mid (E)$	{id, (}	{*, +, $,)}

Calculate the first and follow of the following Grammar

$S \to aBDh$

$B \to cC$

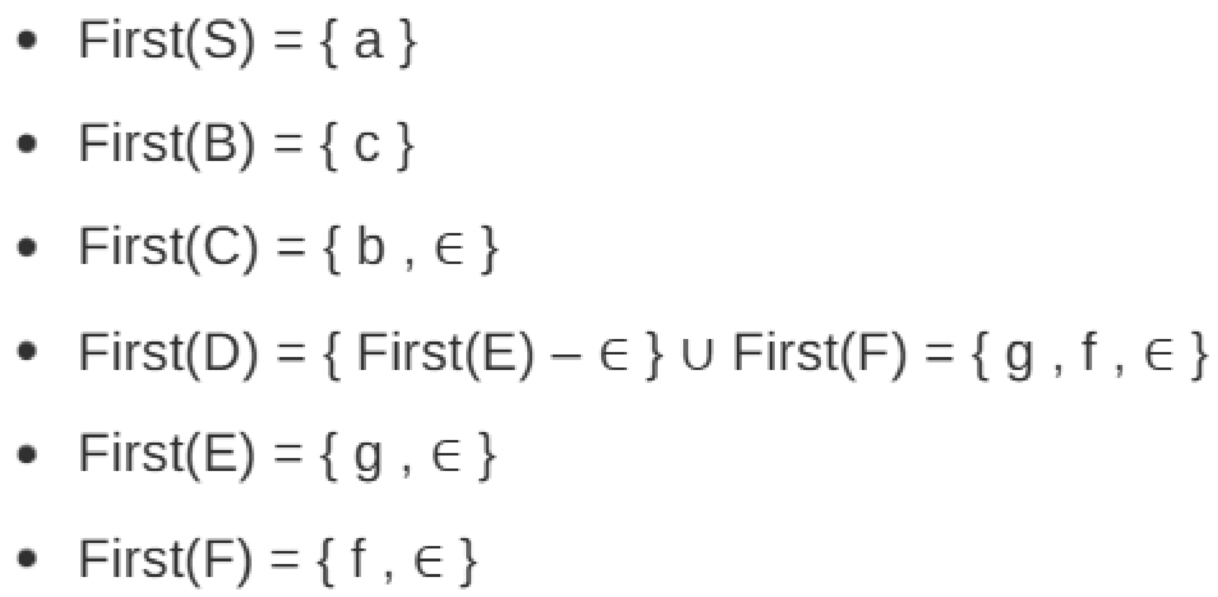
$C \to bC \, / \in$

$D \to EF$

$E \to g \, / \in$

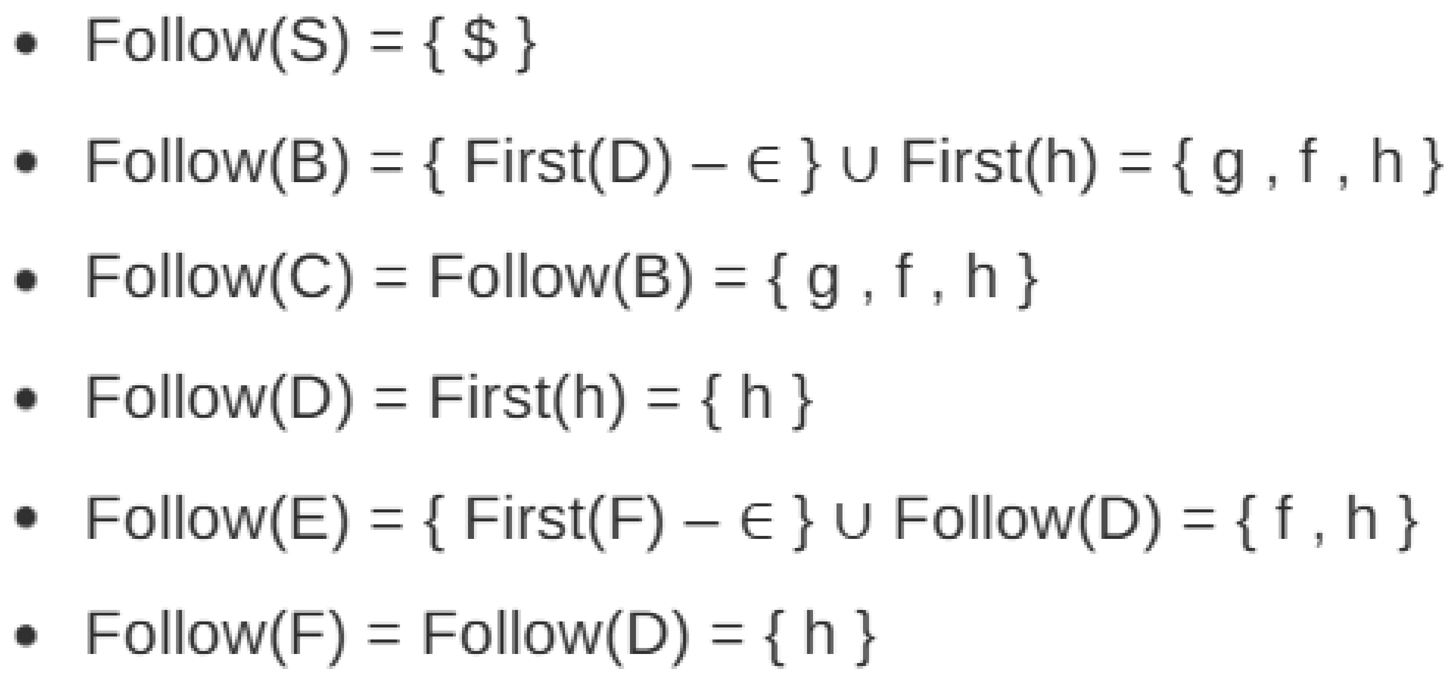
$F \to f \, / \in$

Solution: First



- First(S) = { a }
- First(B) = { c }
- First(C) = { b , ∈ }
- First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
- First(E) = { g , ∈ }
- First(F) = { f , ∈ }

Follow



- Follow(S) = { $ }
- Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
- Follow(C) = Follow(B) = { g , f , h }
- Follow(D) = First(h) = { h }
- Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }
- Follow(F) = Follow(D) = { h }

Example 2:

- S -> ACB|Cbb|Ba
- A -> da|BC
- B-> g|Є
- C-> h| Є

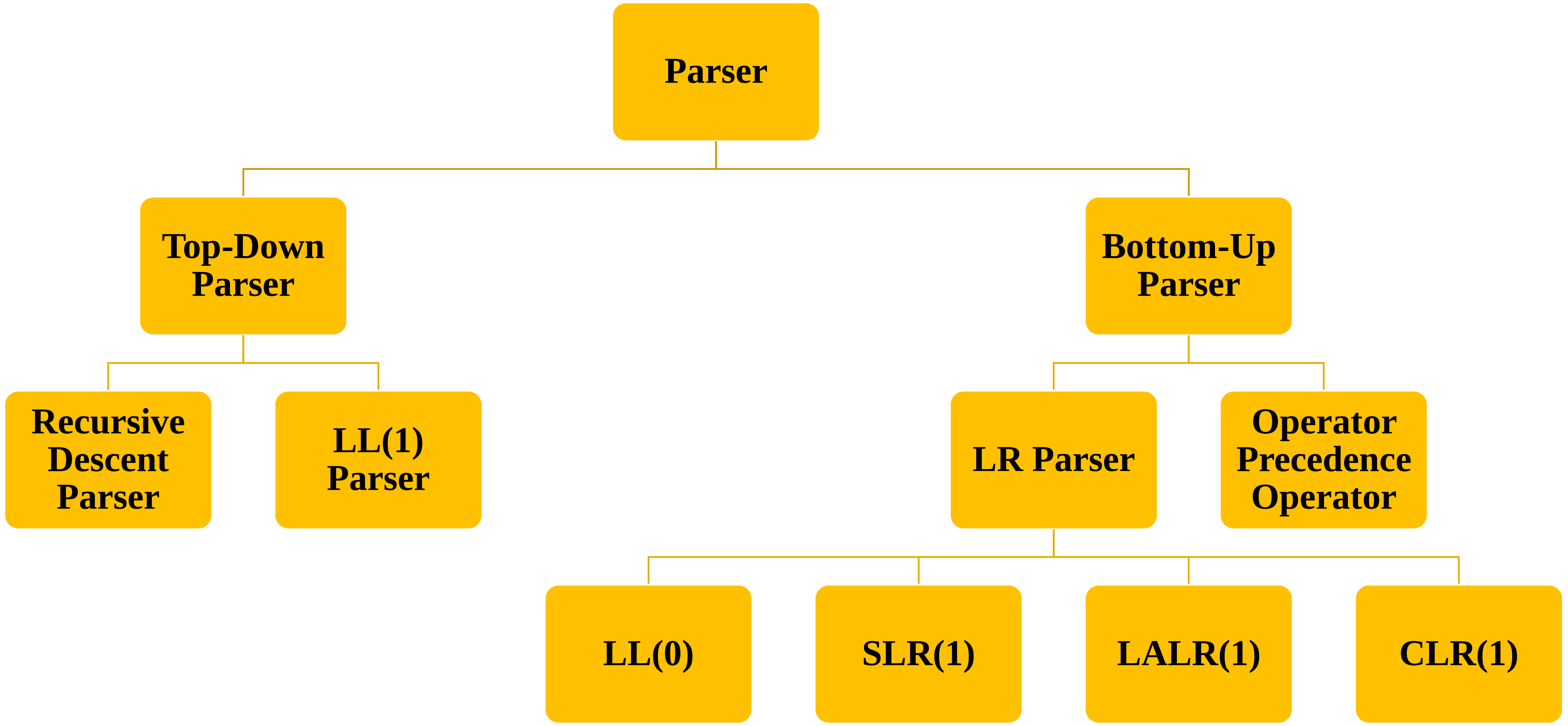
Example 2: First Solution

- FIRST(S) = { d, g, h, Є, b, a}
- FIRST(A) = { d, g, h, Є }
- FIRST(B) = { g, Є }
- FIRST(C) = { h, Є }

Follow

- FOLLOW(S) = { $ }
- FOLLOW(A) = { h, g, $ }
- FOLLOW(B) = { a, $, h, g }
- FOLLOW(C) = { b, g, $, h }

Parsing Technique


Parser
Top-Down Parser
Bottom-Up Parser
Recursive Descent Parser
LL(1) Parser
LR Parser
Operator Precedence Operator
LL(0)
SLR(1)
LALR(1)
CLR(1)