CS4031
Compiler Construction
Lecture 5

Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

Ambiguity Removal

- The ambiguity removal is important in the grammar which consist of

1. Logical Operators
2. Arithmetic Operators
3. If else statement
4. Regular expressions operators

Arithmetic Operators

- The two basic Properties of ambiguity removals are

1. Precedence
2. Associativity

Precedence	
+, -	4
*, /	3
^	2
Id/terminals	1

Associativity		
+, -	Left to right	Keep left recursion
*, /	Left to right	Keep left recursion
^	Right to left	Keep right recursion
id/terminal	-	

Logical Operators

- The two basic Properties of ambiguity removals are

1. Precedence
2. Associativity

Precedence	
∨	**4**
∧	**3**
~	**2**
Id/terminals	**1**

Associativity		
∨	**Left to right**	**Keep left recursion**
∧	**Left to right**	**Keep left recursion**
~	**Right to left**	**Keep right recursion**
id/terminal	**-**	**-**

Example

Ambiguous Grammar

$$\boldsymbol{E \rightarrow E \wedge E \mid E \vee E \mid \sim E \mid (E) \mid id}$$

Unambiguous Grammar

$$\boldsymbol{E \rightarrow E \vee E_1 \mid E_1}$$
$$\boldsymbol{E_1 \rightarrow E_1 \wedge E_2 \mid E_2}$$
$$\boldsymbol{E_2 \rightarrow \ \sim E_2 \mid E_3}$$
$$\boldsymbol{E_3 \rightarrow (E) \mid id}$$

Regular Expression Operators

- The two basic Properties of ambiguity removals are

1. Precedence
2. Associativity

Precedence	
Union	**4**
Concatenation	**3**
Kleene star	**2**
Id/terminals	**1**

Associativity		
Union	**Left to right**	**Keep left recursion**
Concatenation	**Left to right**	**Keep left recursion**
Kleene star	**-**	**-**
id/terminal	**-**	**-**

Regular Expression Operators Example

Ambiguous Grammar

$$E \rightarrow E + E \mid EE \mid E^* \mid id$$

Unambiguous Grammar

$$E \rightarrow E + E_1 | E_1$$
$$E_1 \rightarrow E_1 \ E_2 | E_2$$
$$E_2 \rightarrow E_2^* | E_3$$
$$E_3 \rightarrow id$$

If- else Problem

- Consider the following Grammar

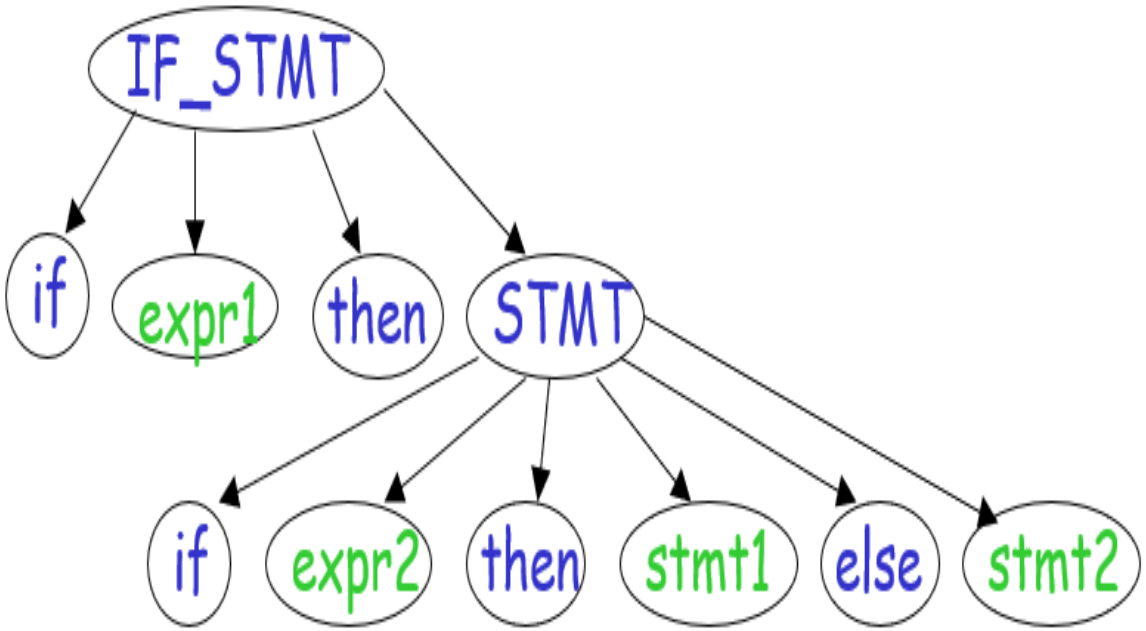
$$S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{other}$$

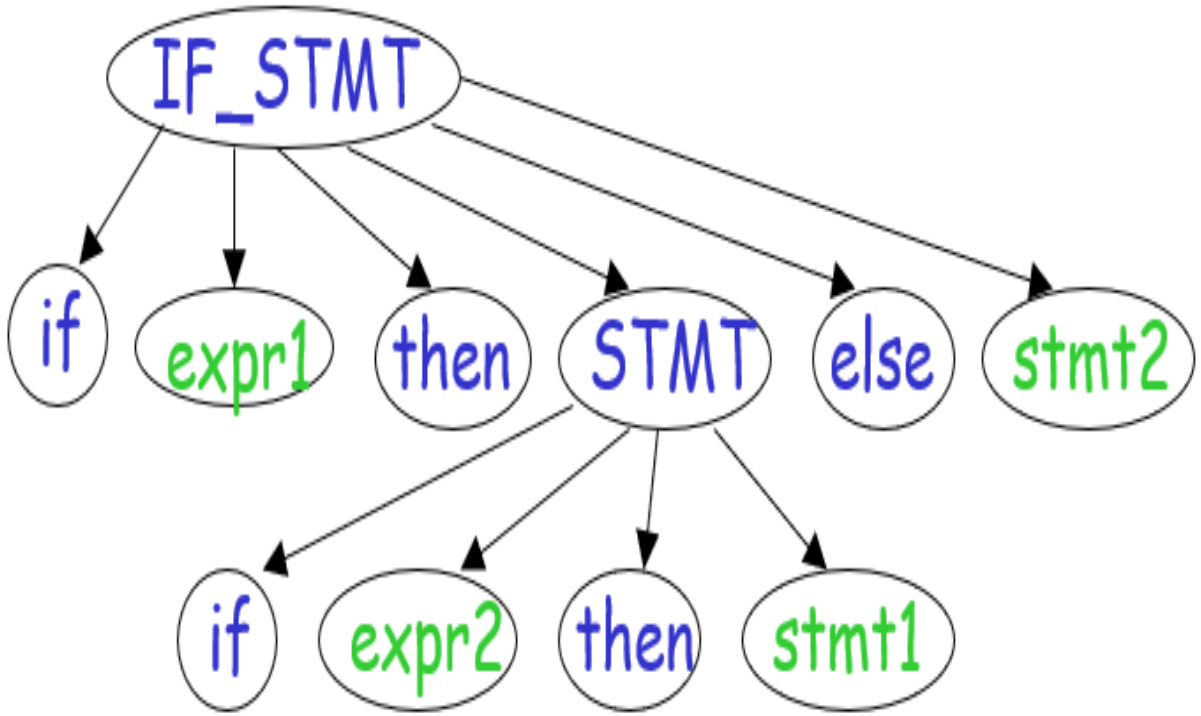
Where

E represent the Expression

S represent the Statement

Parse Tree


IF_STMT
if
expr1
then
STMT
if
expr2
then
stmt1
else
stmt2


IF_STMT
if
expr1
then
STMT
else
stmt2
if
expr2
then
stmt1

If else Statements

Matched statements

It is either a non-if statement, or a complete if-then-else statement.

Open statements

It is either a if-then statement (without else), or it is a if-then-else statement but the else-statement is an open statement.

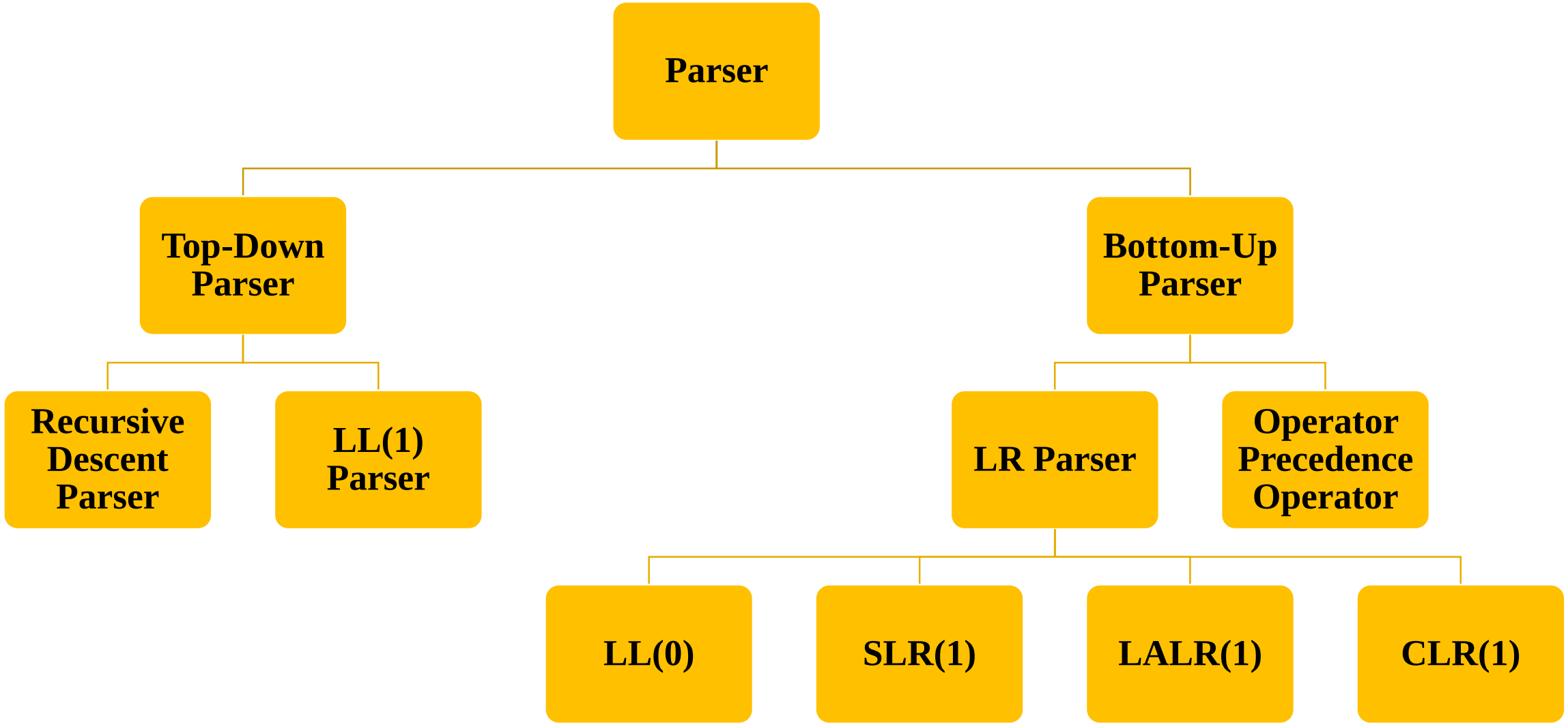
Unambiguous grammar

$$
\begin{aligned}
\text{stmt} &: \text{matchedStmt} \\
&\mid \text{openStmt} \\
\text{matchedStmt} &: \textbf{if } E \textbf{ then } \text{matchedStmt } \textbf{else } \text{matchedStmt} \\
&\mid \textit{others} \ldots \\
\text{openStmt} &: \textbf{if } E \textbf{ then } \text{stmt} \\
&\mid \textbf{if } E \textbf{ then } \text{matchedStmt } \textbf{else } \text{openStmt}
\end{aligned}
$$

Example

- if E1 then if E2 then S1 else S2
- Can you construct two distinct parse tree for it?

Parsing Technique



Parser

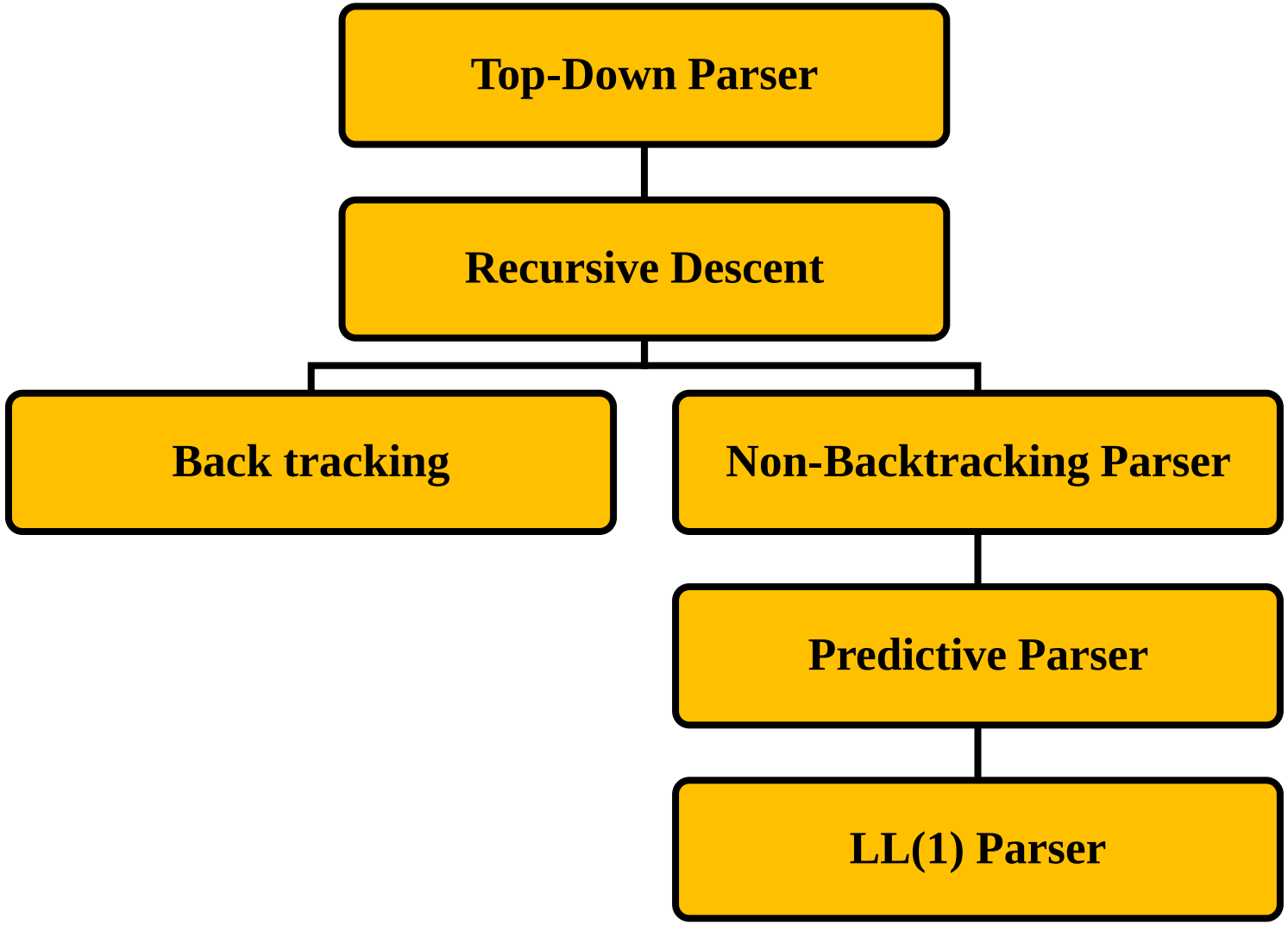
Top-Down Parser

Top- Down parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

Bottom-Up Parser

bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Top-Down Parser



Recursive Descent Parser

- Recursive Descent Parsing is a top-down method of syntax analysis in which s set of recursive procedures to process the input is executed.
- A procedure is a associated with each nonterminal of a grammar.
- Top-down parsing can be viewed as an attempt to find a leftmost derivations for an input string.
- Recursive descent parsing involves backtracking.

Example

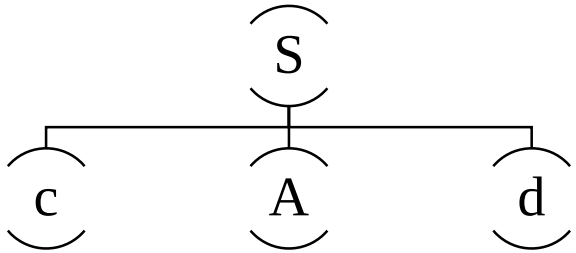
- Consider the following grammar

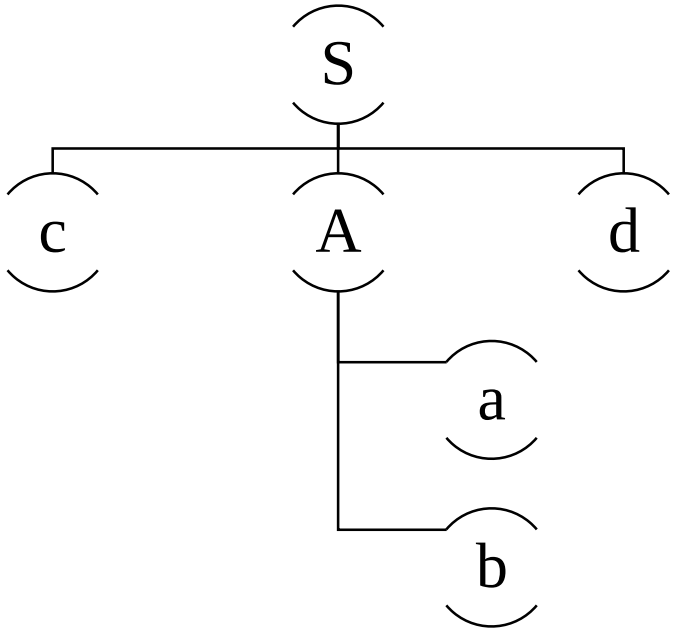
$S \rightarrow cAd$

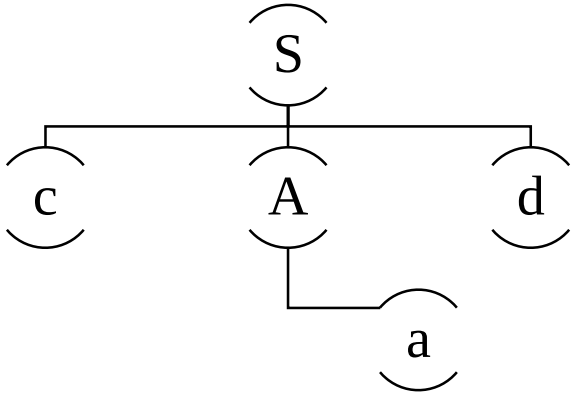
$A \rightarrow ab \mid a$

And the input string w = cad.

Solution







Top Down Parser

- In top down parser we have two types of parser

1. Recursive Descent
2. LL(1) Parser
    1. First L represent the direction of look a head pointer.
    2. L represent the type of derivation.
    3. 1 means how many symbol we read from the string.
    4. LL means we used left to right derivation and making the left most derivation tree.

Condition of LL(1) Parser

- To construct a working LL(1) parsing table, a grammar must satisfy these conditions:
  - No Left Recursion: Avoid recursive definitions like A -> A + b.
  - Unambiguous Grammar: Ensure each string can be derived in only one way.
  - Left Factoring: Make the grammar deterministic, so the parser can proceed without guessing.

LL(1) Parser

- The rules for LL(1) parsers are

1. Remove left Recursion
2. Remove Left Factoring
3. Find the first and follow of the grammar
4. Create Parse Table
5. Create Parse tree

Example: Check whether the grammar is LL(1) or not.

- E → TE'
- E' → +TE' | ε
- T → FT'
- T' → *FT' | ε
- F → id | (E)

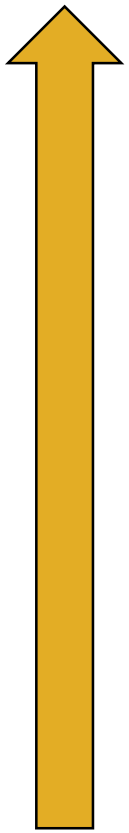
- *ε denotes epsilon

First of Grammar

- E → TE'
- E' → +TE' | ε
- T → FT'
- T' → *FT' | ε
- F → id | (E)

Grammar Production Rule	Non-Terminals	First
E → TE'	E	
E' → +TE' | ε	E`	
T → FT'	T	
T' → *FT' | ε	T`	
F → id | (E)	F	

First of Grammar



Grammar Production Rule	Non-Terminals	First
E → TE'	E	{id,(}
E' → +TE' | ε	E`	{+, ε}
T → FT'	T	{id, (}
T' → *FT' | ε	T`	{*, ε}
F → id | (E)	F	{id, (}

Follow of the Grammar

Grammar Production Rule	Non-Terminals	Follow
E → TE'	**Follow of (E)**	**($,) }**
E' → +TE' | ε	**E`**	
T → FT'	**T**	
T' → *FT' | ε	**T`**	
F → id | (E)	**F**	

Must Remember the follow of first symbol must have $

Follow of the Grammar

Grammar Production Rule	Non-Terminals	Follow
E → TE'	Follow of (E)	{$,) }
E' → +TE' | ε	Follow of E`	{$,) }
T → FT'	Follow of T = First of E`	{+, $,)}
T' → *FT' | ε	Follow of T` = follow of T	{+, $,)}
F → id | (E)	Follow of F = first of T`	{*,+, $,)}

Must Remember the follow of first symbol must have $

Parsing Table

Add all follow symbols in column
Add the variable symbols in row

Grammar Production Rule	First	Follow
E → TE'	{id,(}	{$,) }
E' → +TE' | ε	{+, ε}	{$,) }
T → FT'	{id, (}	{+, $,)}
T' → *FT' | ε	{*, ε}	{+, $,)}
F → id | (E)	{id, (}	{*,+, $,)}

	id	+	*	(	)	$
E	E → TE’			E → TE'		
E`		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T`		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

If the first any variable is ε then move towards the follow of that variable.

Create the Parser Tree of the following

id+ id * id

Stack Implementation

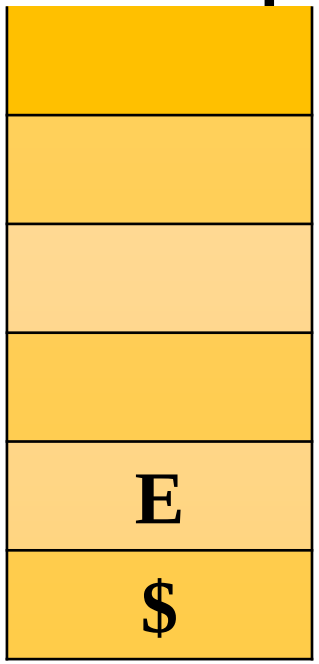
- Given Expression

id + id * id $

- Given Grammar

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow id \mid (E)$$

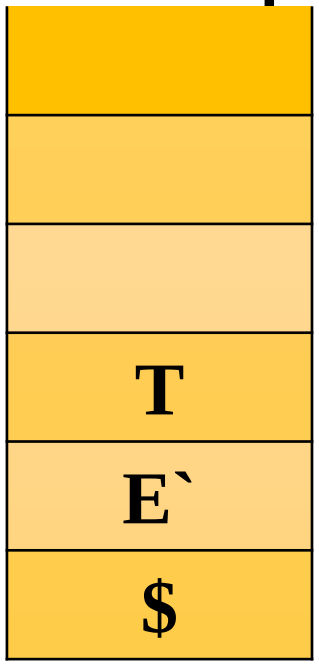
Stack Implementation



id + id * id $

	id	+	*	(	)	$
E	**E → TE'**			**E → TE'**		
E`		**E' → +TE'**			**E' → ε**	**E' → ε**
T	**T → FT'**			**T → FT'**		
T`		**T' → ε**	**T' → *FT'**		**T' → ε**	**T' → ε**
F	**F → id**			**F → (E)**		

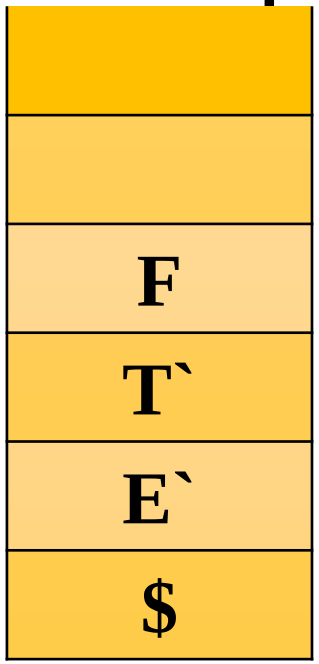
Stack Implementation



id + id * id $

	id	+	*	(	)	$
E	E → TE’			E → TE'		
E`		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T`		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

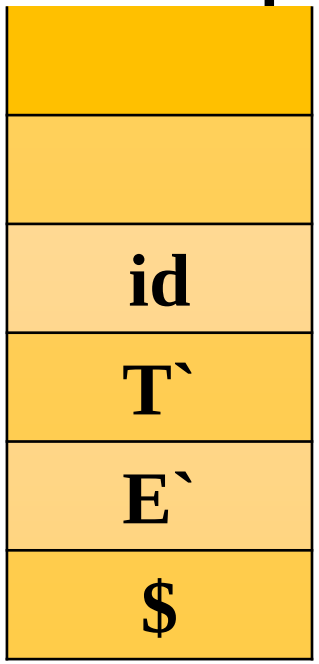
Stack Implementation



id + id * id $

	id	+	*	(	)	$
E	E → TE’			E → TE'		
E`		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T`		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack Implementation

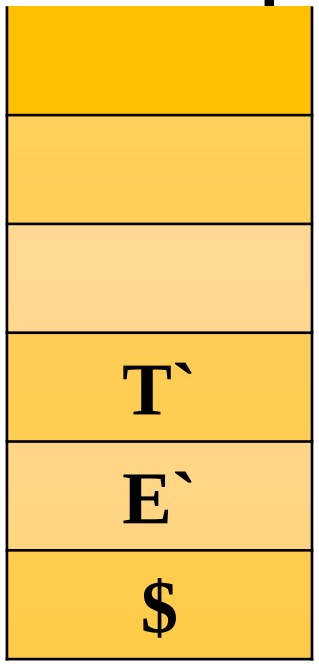


id + id * id $

id

	id	+	*	(	)	$
E	**E → TE'**			**E → TE'**		
E`		**E' → +TE'**			**E' → ε**	**E' → ε**
T	**T → FT'**			**T → FT'**		
T`		**T' → ε**	**T' → *FT'**		**T' → ε**	**T' → ε**
F	**F → id**			**F → (E)**		

Stack Implementation

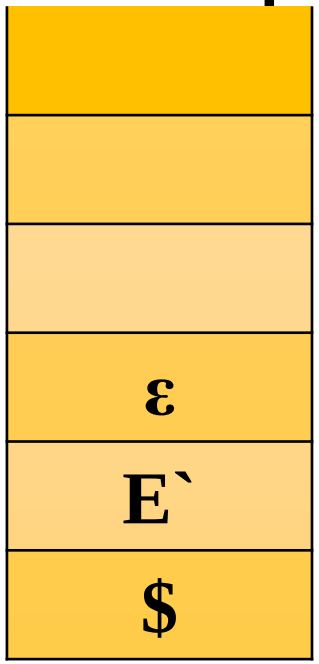


id + id * id $

id

	id	+	*	(	)	$
E	**E → TE'**			**E → TE'**		
E`		**E' → +TE'**			**E' → ε**	**E' → ε**
T	**T → FT'**			**T → FT'**		
T`		**T' → ε**	**T' → *FT'**		**T' → ε**	**T' → ε**
F	**F → id**			**F → (E)**		

Stack Implementation

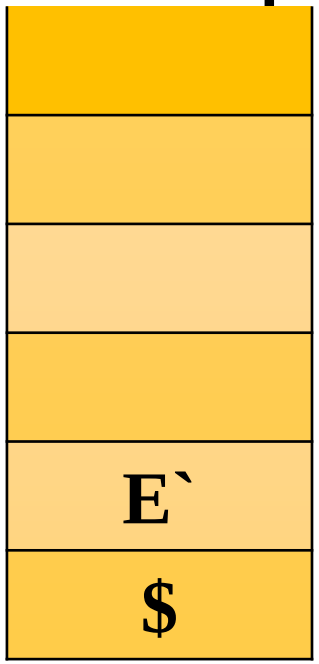


id + id * id $

id

	id	+	*	(	)	$
E	E → TE’			E → TE'		
E`		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T`		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack Implementation

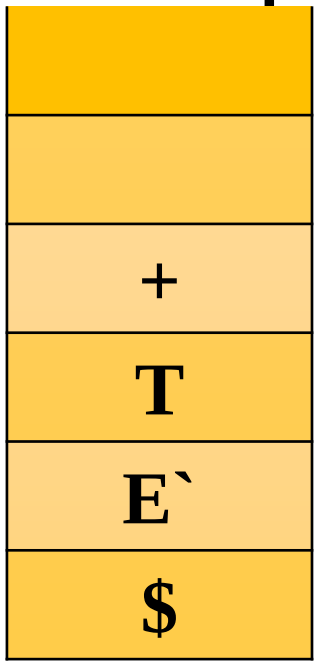


id + id * id $

id

	id	+	*	(	)	$
E	E → TE’			E → TE'		
E`		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T`		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack Implementation

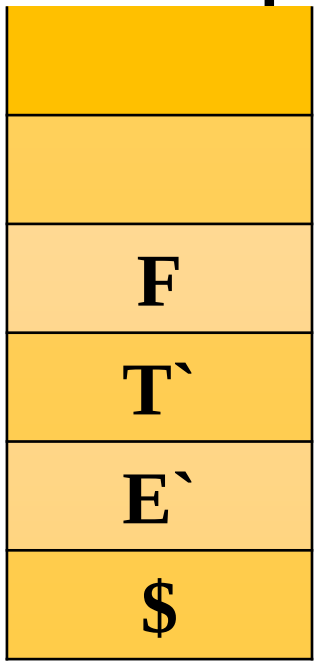


id + id * id $

id

	id	+	*	(	)	$
E	**E → TE'**			**E → TE'**		
E`		**E' → +TE'**			**E' → ε**	**E' → ε**
T	**T → FT'**			**T → FT'**		
T`		**T' → ε**	**T' → *FT'**		**T' → ε**	**T' → ε**
F	**F → id**			**F → (E)**		

Stack Implementation



id + id * id $

id +

	id	+	*	(	)	$
E	E → TE’			E → TE'		
E`		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T`		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

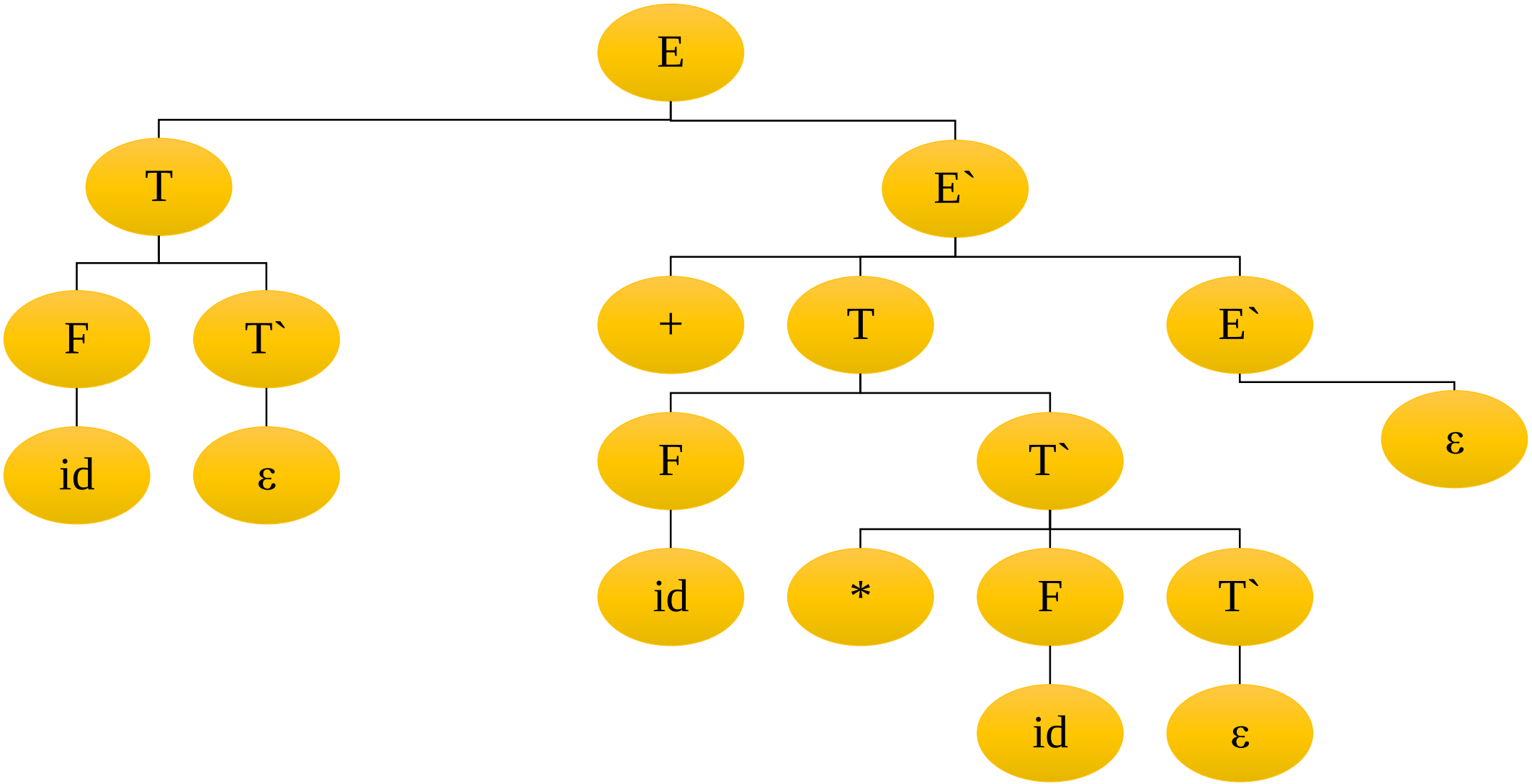
Stack Implementation (Complete Table)

MATCHED	STACK	INPUT	ACTION
	E$	id + id * id$	**E → TE'**
	TE`$	id + id * id$	**T → FT'**
	FTE`$	id + id * id$	F→ **id**
	ididT`E$	id + id * id$	Match id
id	T`E$	+ id * id$	T` → **epsilon**
id	E`$	+ id * id$	E` → **+TE`**
id	+TE`$	+ id * id$	Match +
id+	TE`$	id * id$	T → **FT`**
id+	FT`E`$	id * id$	F → **id**
id+	idT`E`$	id * id$	Match id
id+ id	T`E`$	* id$	**T' → *FT`**
id+ id	FT`E$	* id$	Match *
id+ id *	FT`E`$	id$	F → **id**

Stack Implementation

MATCHED	STACK	INPUT	ACTION
id+ id *	Id T`E`$	Id $	F → **id**
id+ id * id	T`E`$	$	**Match id**
id+ id * id	E`$	$	T` → **epsilon**
id+ id * id	$	$	E` → **epsilon**

Parse Tree



Example 2: Draw the LL(1) parsing table for the given grammar ?

- **S** → iEtSS’ | a
- S' → eS | Ɛ
- E → b

	First	Follow
S	**{i , a}**	**{$, e, ∈}**
S`	**{e, ∈}**	**{$, e, ∈}**
E	**{ b }**	**{t}**

Solution:

S → iEtSS’ | a
S' → eS | Ɛ
E → b

	First	Follow
S	{i , a}	{$, e}
S`	{e, ∈}	{$, e}
E	{ b }	{t}

	a	b	e	i	t	$
S	S → a			S → iEtSS		
S`			S' → eS S' → ∈			S' → ∈
E		E → b				

This grammar is not feasible for LL(1) Parser.

Example 3: Draw the LL(1) parsing table for the given grammar ?

- **S → aABb**
- **A → c | ε**
- **B → d | ε**

Example 4: Draw the LL(1) parsing table for the given grammar ?

- **S → W**
- **W → ZXY**
- **Y → c | ε**
- **Z → a | d**
- **X → Xb | ε**