Compiler 1

Lecture 7: Context-Free Grammar

BY: Assist Prof. Dr. Ielaf O Abdul Majjed

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- ❖ Deciding the non-terminal which is to be replaced.
- ❖ Deciding the production rule, by which, the non-terminal will be replaced.

***To decide which non-terminal to be replaced with production rule, we can have two options*.**

1. Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called *left-most* derivation..

2. Right-most Derivation

If we scan and replace the input with production rules, from right to left, it is known as *right-most* derivation.

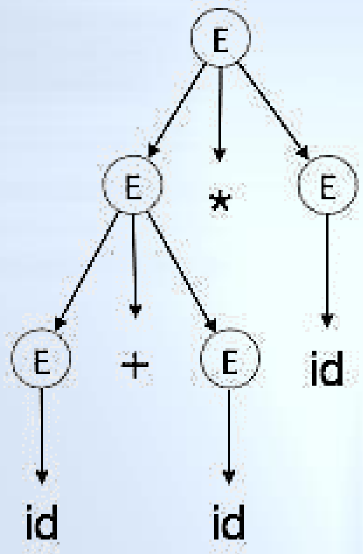
Example : Production rules

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

Input string: id + id * id



The left-most derivation is

$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

The right-most derivation is

$E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

$E \rightarrow id + id * id$

Ambiguity

A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

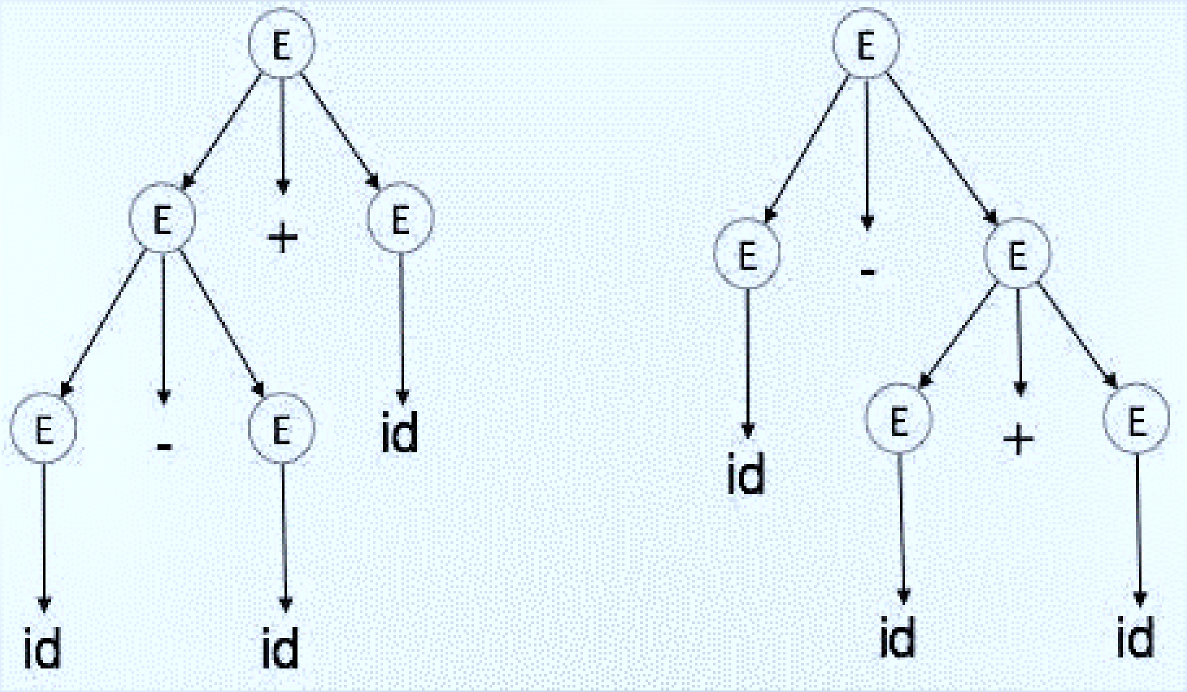
Example

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

For the string id + id – id, the above grammar generates two parse trees:



The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following ***associativity*** and ***precedence*** constraints.

1. solved by associativity

If an operator is right-associative (or left-associative), an operand in between 2 operators is associated to the operator to the right (left).

- Right-associated : W + (X + (Y + Z))
- Left-associated : ((W + X) + Y) + Z

E -> E-E | id → ***E -> E – P | P*** **// Left Recursive**

P -> id

or E -> E-id | id

E -> P - E | P ***// Right Recursive***

P -> id

or E -> id- E | id

2. solved by precedence

- An operator with higher precedence is done before one with lower precedence.
- An operator with higher precedence is placed in a rule (logically) further from the start symbol.

E -> E + E | E * E | id

E -> E + P *// + is at higher level and left associative*

E -> P

P -> P * Q *// * is at lower level and left associative*

P -> Q

Q -> id

(or)

E -> E + P | P

P -> P * Q | Q

Q -> id

H.W.

1- Derive the statement (id + id) * id in both *leftmost and rightmost* derivation using the following grammar:

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

$O \rightarrow + \mid - \mid * \mid /$

2- Is the following grammar ambiguous? If so, eliminate the ambiguity.

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$

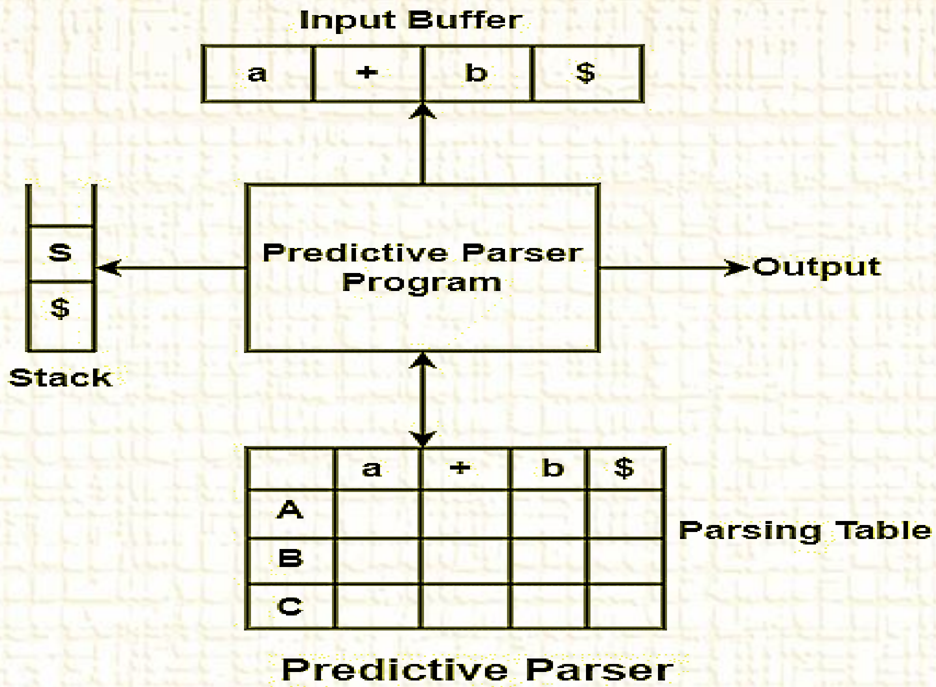
$E \rightarrow id$

Compiler 1

Non Recursive Predictive Parser

BY: Assist Prof. Dr. Ielaf O Abdul Majjed

Non Recursive Predictive Parser



Non-recursive parser model diagram

Predictive Parser is also another method that implements the technique of Top- Down parsing without Backtracking. A predictive parser is an effective technique of executing recursive-descent parsing by managing the stack of activation records, particularly.

Predictive Parsers has the following components:

- **Input Buffer:** The input buffer includes the string to be parsed followed by an end marker $ to denote the end of the string.
- **Stack:** It contains a combination of grammar symbols with $ on the bottom of the stack. At the start of Parsing, the stack contains the start symbol of Grammar followed by $.
- **Parsing Table:** It is a two-dimensional array or Matrix M [A, a] where A is nonterminal and 'a' is a terminal symbol.
- **Parsing Program:** The parsing program performs some action by comparing the symbol on top of the stack and the current input symbol to be read on the input buffer.
- **Actions:** Parsing program takes various actions depending upon the symbol on the top of the stack and the current input symbol. Various Actions taken are given below:

Description	Top of Stack	Current Input Symbol	Action
1. If stack is empty, i.e., it only contains $ and current Input symbol is also $.	$	$	Parsing will be successful and will be halted.
2. If symbol at top of stack and the current input symbol to be read are both terminals and are same.	a $	a b $	Pop a from stack & advance to next input symbol.
3. If both top of stack & current input symbol are terminals and top of stack ≠ current input symbol e.g. a ≠ b.	a $	b $	Error
4. If top of stack is non-terminal & input symbol is terminal.	X $	a $	Refer to entry M [X, a] in Parsing Table. If M[X, a] = X → ABC then Pop X from Stack Push C, B, A onto stack.

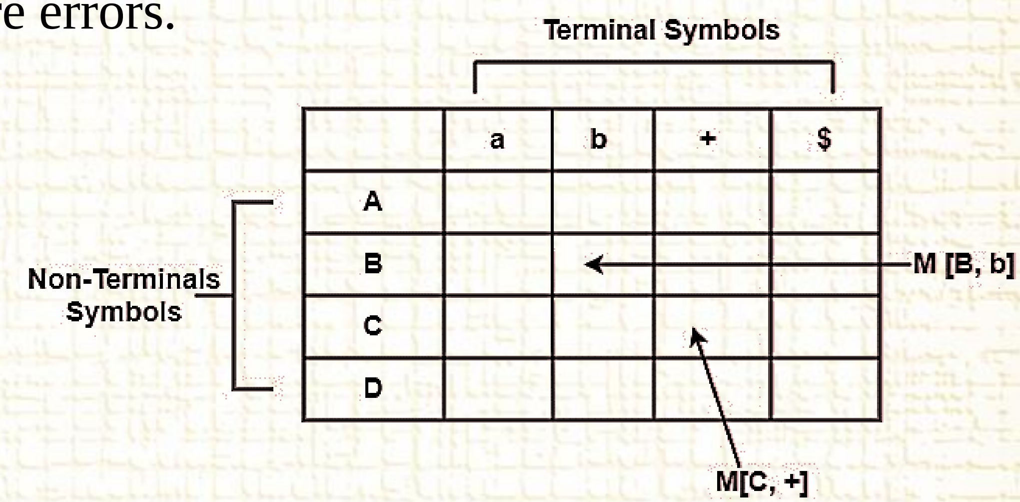
Algorithm to construct Predictive Parsing Table

Input − Context-Free Grammar G

Output − Predictive Parsing Table M

Method − For the production A → α of Grammar G.

- For each terminal, a in FIRST (α) add A → α to M [A, a].
- If ε is in FIRST (α), and b is in FOLLOW (A), then add A → α to M[A, b].
- If ε is in FIRST (α), and $ is in FOLLOW (A), then add A → α to M[A, $].
- All remaining entries in Table M are errors.



LL(1) Grammars

Predictive parsers, needing no backtracking, can be constructed for a class of grammars called LL(1).

The **first** "**L**" in LL(1) stands for scanning the input from left to right, the **second** "**L**" for producing a leftmost derivation, and the "**1**" for using one input symbol of look ahead at each step to make parsing action decisions.

LL(1) grammar has the following conditions:

1. Unambiguous grammar.
2. No Left-recursive.
3. No Left factoring.

Consider the following grammar **:**

E->E+T|T

T->T×F|F

F->(E)|id

After removing left recursion, left factoring

E → TE'

E' → +TE' | ε

T → F T'

T' → × F T' | ε

F → (E) | id

Produces the parsing table

Solution:

First (E) = {(, id}	Follow (E) = {$,)}
First (E') = {+, ε}	Follow (E') = {$,)}
First (T) = {(, id}	Follow (T) = {+, $,)}
First (T') = {×, ε}	Follow (T') = {+, $,)}
First (F) = {(, id}	Follow (F) = {×, +, $,)}

The parsing table:

Nonterminal	Input Symbol					
	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)		

Predictive Parsing for id + id * id

Stack	Input	production
E $	id + id × id $	
TE' $	id + id × id $	E → TE'
FT' E' $	id + id × id $	T → FT'
id T' E' $	id + id × id $	F → id
T' E' $	+ id × id $	pop id
E' $	+ id × id $	T' → ε
+ TE' $	+ id × id $	E' → + TE'
TE' $	id × id $	pop +
FT' E' $	id × id $	T → FT'
id T' E' $	id × id $	F → id
T' E' $	× id $	pop id
× FT' E' $	× id $	T' → × FT'
FT' E' $	id $	pop ×
id T' E' $	id $	F' → id
T' E' $	$	pop id
E' $	$	T' → ε
$	$	E' → ε