$E \rightarrow T\bar{E}$
$\bar{E} \rightarrow +T\bar{E} \mid \varepsilon$
$T \rightarrow F\bar{T}$
$T' \rightarrow *F\bar{T} \mid \varepsilon$
$F \rightarrow (E) \mid id$

The parse table M for the grammar

NONTER-MINAL	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)		

The moves made by predictive parse on input **id + id * id**

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ϵ
$E'T+	+ id * id$	E' → +TE'
$E'T	id * id$	
$E'T'F	id * id$	T → FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	
$E'T'F*	* id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ϵ
$	$	E' → ϵ

https://www.youtube.com/watch?v=_uSlP91jmTM

2.6.5.1 First and Follow

- Left Recursion & Left Factoring يجب التخلص من مشكلة

بعد التخلص من مشكلة Left Recursion & Left Factoring
ان وجدت في السؤال توجد خطوات يجب الاعتماد عليها في الحل وهي :-
1- First لكل حرف صغير هو الحرف نفسه (العمليات الرياضية +،ـ ، / ،* تعتبر احرف صغيره و First لها نفسه
وجميع احرف اللغة الانكليزية الصغير الغير a,b,c… ال First لها يبقى نفسه) .
2- First لكل حرف كبير هو توليد هذا الحرف وصولا الى اول حرف صغيرونتوقف .

Ex:- 1
A → abc | def | ghi

A→ abc
A→ def
A→ ghi

First A = a,d,g

Ex:- 2
S→ AB | b | c
A→ a

First A = a
First S = b,c,a

Ex:- 3
S→ AB
A→ a
B→ b| €

First A = a
First B =b,€
First S = a, b,€

https://www.youtube.com/watch?v=n_ktF-PoPoQ
https://www.youtube.com/watch?v=HLNSUkp8HHA

Example:-

E→ E+T | T
T → T*F | F
F → (E) | id

يجب التخلص من مشكلة Left Recursion & Left Factoring

- Left Recursion

E → T$\bar{E}$
$\bar{E}$ → +T$\bar{E}$ | λ
T → F$\bar{T}$
$\bar{T}$ → *F$\bar{T}$ | λ
F → (E) | id

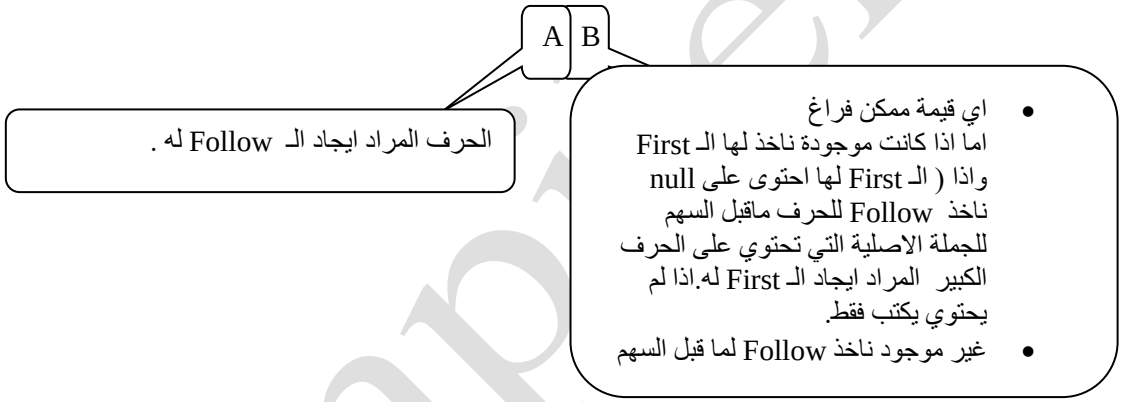
لاتحتوي على Left Factoring

- First

- First (E)= { (, id }
- First (E̅)= { + , € }
- First (T)= { (, id }
- First (T̅)= { * , € }
- First (F)= {(, id }

2.6.5.2 Follow

ايضا توجد عدة خطوات يجب اتباعها في الحل وهي :-



Ex:-

S1→ S#
S → qABC
A → a | bbD
B → a | €
C → b | €
D → c | €

First **Follow**

Sol :-
S1→ { q }
S → { q }
A → { a , b }
B → { a , € }
C → { b , € }
D → { c , € }

S1→ { $ }
S → { # }
A → { a , # }
B → { b , # }
C → { # }
D → { a , # }

https://www.youtube.com/watch?v=HLNSUkp8HHA

Ex:-

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

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

➢ **First**

- First (E)= { (, id }
- First ($\overline{E}$)= { + , € }
- First (T)= { (, id }
- First ($\overline{T}$)= { * , € }
- First (F)= {(, id }

نبحث عن الحرف المراد ايجاد الـ Follow لها بعد السهم وليس قبل السهم .

➢ **Follow**

تكتب الـ $ في بداية جملة الحل فقط .

- Follow (E) = { $,) }
- Follow ($\overline{E}$) = { $,) }
- Follow (T) = { + , $,) }
- Follow ($\overline{T}$) = { + , $,) }
- Follow (F) = {* , + , $,) }

كتابة الملاحظات الموجوده في صفحه 15 من الملاحظات

Example:-

S→ aSb | X
X→ cXb | b
X→ bXZ | X
Z→ n

Sol :-

➢ **First**

- First (S) = a , c , b
- First (X) = c , b
- First (Z) = n

Follow

Follow (S) = {$, b }
Follow (X) = {$, b , n }
Follow (Z) = {$, b , n }

كتابة الملاحظات في 16

Ex:-

S → bXY
X → b | c
Y → b | ε

Sol:

	First	Follow
S	b	$
X	b , c	b , $
Y	b , ε	$

Ex:

S → ABb | bc
A → ε | abAB
B → bc | cBS

	First	Follow
S	**b , a , c**	**$, b , c , a**
A	**ε , a**	**b , c**
B	**b , c**	**b , c , a**

Ex:

X → ABC | n X
A → bA | bb | ε
B → bA | CA
C → ccC | CA | cc

Sol:

	First	Follow
X	n , b , c	$
A	b , ε	b , c , $
B	b , c	c
C	c	b , $

H.W:

S → bSX | Y
X → XC | bb
Y → b | bY
C → ccC | CX | cc

Note: there is a left recursion problem here trying to solve this Problem and find the first and follow for this grammar.

Ex:

S → bSX | Y
X → bbX' X' → CX'| ε
Y → b | bY C → ccC | CX | cc

	First	Follow
S	b	$, b
X	b	$, b , c
X'	c , ε	$, b , c
Y	b	$, b
C	c	$, b , c

2.6.6 Construction of predictive parsing tables

Construction of predictive parsing tables: The following algorithm can be used to construct a predictive parsing table for a grammar G. The idea behind the algorithm is the following:

Suppose A→ α is a production with a in FIRST(α). Then the parser will expand A by α when the current input symbol is a. The only complication occurs when α→ ε . In this case, we should again expand A by α if the current input symbol is in FOLLOW(A).

Ex:

E→ E+T | T
T → T*F | F
F → (E) | id

Parse the input id * id + id by using predictive parsing:

1- We must solve the left recursion and left factoring if it founded in the grammar

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

2- We must find the first and follow to the grammar:

	First	Follow
E	(, id	$,)
E’	+ , ε	$,)
T	(, id	+ , $,)
T’	* , ε	+, $,)
F	(, id	+ , *, $,)

3- We must find or construct now the predictive parsing table
Since FIRST(TE’) = FIRST(T) = {(,id}, production E→TE’ causes M[E,( ] and M[E,id] to acquire the entry E→TE’.
Production E’→+TE’ causes m[E’,+] to acquire E’→+TE’. Production E’→ ε causes M[E’,)] and M[E’,$] to acquire E’→ ε since FOLLOW(E’) = {),$}. So the parsing table produced by the previous algorithm.

NONTER-MINAL	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)		

Ex:
https://www.youtube.com/watch?v=WgFi2ssIR3A

2.6.7 LL (1) grammars:

The previous algorithm can be applied to any grammar G to produce a parsing table M. For some grammars, M may have some entries that are multiply defined. If G is left recursive or ambiguous, then M will have at least one multiply-defined entry.

Ex:

First(S) = {i, a} follow(S) = {$, e}
First($\bar{S}$) = {e, ε} follow($\bar{S}$) = { $, e }
First (E) = {b} follow (E) = {t}

So the parsing table for our grammar is:

النص String لهذه العباره ليس **LL(1)** ذلك لوجود الـ λ e في حالة First ووجود الـ $ ، e في حالة الـ follow مما هو الذي ادى الى ذلك.

NONTER-MINAL	INPUT SYMBOL					
	a	*b*	*e*	*i*	*t*	$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

The entry for M[S',e] contains both S'→ eS and S'→ ε, since FOLLOW(S') ={e, $}. The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an e (else) is seen. We can resolve the ambiguity if we choose S'→ eS. Note that the choice S'→ ε would prevent e from ever being put on the stack or removed from the input, and is therefore surely wrong.

A grammar whose parsing table has no multiply-defined entries is said to be LL(1). The first "L" in LL(1) indicates the reading direction (left-to-right), the second "L" indicates the derivation order (left), and the "1" indicates that there is a one-symbol or lookahead at each step to make parsing action decisions.

- **Error Detection and Reporting**

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry M[A,a] is empty. Error recovery is based on the idea of skipping symbols on the input until a token in selected set of synchronizing tokens appears. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

We can place all symbols in Follow (A) into the synchronizing set for nonterminal A. If we skip tokens until an element of Follow (A) is seen and pop A from the stack, it is likely that parsing can continue.

Example:

Using Follow symbols as synchronizing tokens works reasonably well when expressions are parsed according to the grammar:

$E \rightarrow T\bar{E}$
$\bar{E} \rightarrow +T\,\bar{E} \mid \lambda$
$T \rightarrow F\bar{T}$
$\bar{T} \rightarrow *F\bar{T} \mid \lambda$
$F \rightarrow (E) \mid id$

The parsing table for this grammar is repeated with synchronizing tokens If the parser looks up entry M[A,a] and finds that it is blank, then the input symbol a is skipped. If the

entry is synchronize, then the nonterminal on top of the stack is popped in an attempt to resume parsing.

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id	synch	synch	$F \to (E)$	synch	synch

Synchتعني في حالة وجود خطأ اعمل skip اي بمعنى احذف الخطأ.

STACK	INPUT	REMARK
$E	) id * + id $	error, skip)
$E	id * + id $	id is in FIRST(E)
$E'T	id * + id $	
$E'T'F	id * + id $	
$E'T'id	id * + id $	
$E'T'	* + id $	
$E'T'F*	* + id $	
$E'T'F	+ id $	error, M[F, +] = synch
$E'T'	+ id $	F has been popped
$E'	+ id $	
$E'T+	+ id $	
$E'T	id $	
$E'T'F	id $	
$E'T'id	id $	
$E'T'	$	
$E'	$	
$	$	

2.6.8 Bottom – Up Parsing

Bottom up parsers start from the sequence of terminal symbols and work their way back up to the start symbol by repeatedly replacing grammar rules' right hand sides by the corresponding non-terminal. This is the reverse of the derivation process, and is called "reduction".

Ex:1 consider the grammar

S→ aABe

A→ Abc|b

B→ d

The sentence **abbcde** can be reduced to S by the following steps:

Sol:
abbcde
aAbcde
aAde
aABe
S

S→ aABe
S→ aAbcBe
S→ abbcBe
S→ abbcde

Example:2 consider the grammar
S→ aABe
A→ Abc|bc
B→ dd

The sentence **abcbcdde** can be reduced to S by the following steps:

Sol:
abcbcdde
aAbcdde
aAdde
aABe
S

Definition: a handle is a substring that
1- Matches a right hand side of a production rule in the grammar
2- Whose reduction to the non-terminal on the left hand side of that grammar rule is a step along the reverse of a rightmost derivation.

There is a general style of bottom-up syntax analysis, known as **shift reduces parsing**.

Example 1:
parse the input **id +id *id** for this grammar

E → E+E
E → E*E
E → (E)
E → id

يجب مراعاة اسبقية العمليات الرياضية اثناء الحل. واعتماد الـ shift &reduce