University of sulaimani
College of science
Department of Computer Science

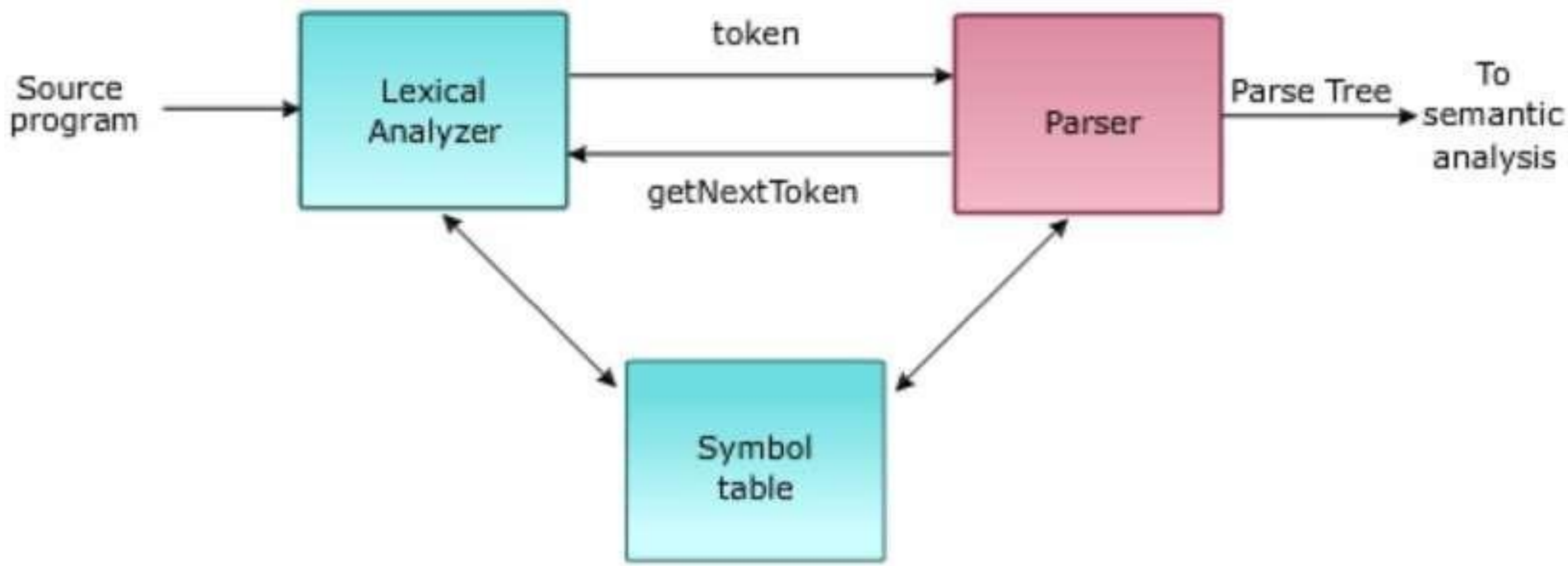
Compiler
Second Phase of the Compiler
Syntax Analyzer

2024-2025

Mzhda Hiwa

Syntax Analyzer

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.



Role of the Syntax Analyzer

- Takes input from the lexical analyzer (a stream of tokens) and organizes them into a hierarchical structure.
- Builds a **parse tree** or **abstract syntax tree (AST)** that represents the syntactic structure of the code. This tree is used in later stages of the compilation process.
- Ensures the source code follows the correct grammatical structure.
- Detects syntax errors like missing semicolons, unmatched brackets, incorrect ordering of tokens, etc. Reports to user where any syntax error in the source code are.

Introduction to Parsing → Syntax Analysis

- **Parsing** is the process of determining how a string of terminals can be generated by a grammar.

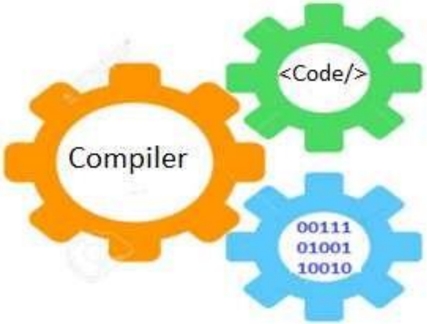
$$F \rightarrow id \mid (E)$$

- A parser must be capable of constructing the tree in principle, or else the translation cannot be guaranteed correct.

- A parser scans the input string from left to right and it makes use of production rules for choosing appropriate derivation.

abbcde

- Two parsing techniques:
  - Top-down parsing
  - Bottom-up parsing

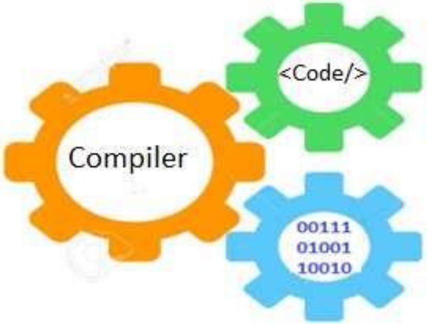


Top-down parsing

- A parser is top-down if it generates a parse tree starting from the root and precedes towards the leaves.
- It is easier to understand and program manually
- A leftmost derivation is applied at each derivation step

➢ Two kinds of top-down parsing techniques will be studied

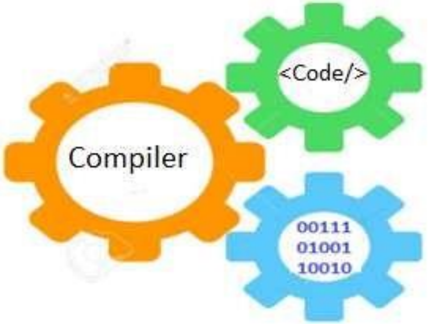
1. Recursive-Decent parser
2. Predictive parser



Bottom-Up Parsing

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

- Bottom-up parsing is more general than top-down parsing.



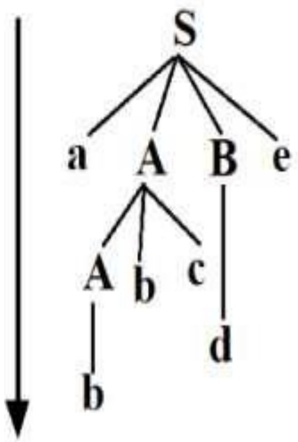
Example

$S \rightarrow aABe$ w=abbcde

$A \rightarrow Abc \mid b$

$B \rightarrow d$

- Top-down LMD



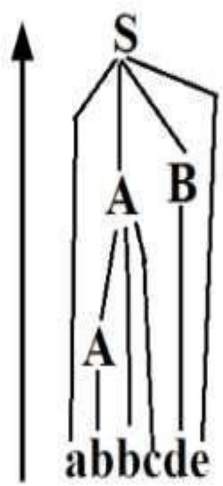
$S \rightarrow aABe$

$S \rightarrow aAbcBe$

$S \rightarrow abbcBe$

$S \rightarrow abbcde$

- Bottom-up RMD



$S \rightarrow aABe$

$S \rightarrow aAde$

$S \rightarrow aAbcde$

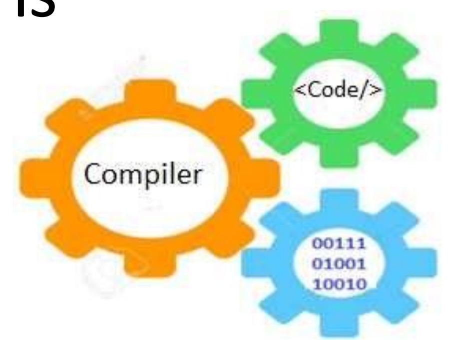
$S \rightarrow abbcde$

1- Top-Down Parsing by Recursive-Descent

- It reads characters from the input stream and matches them with terminals from the grammar.

The operation involved are : (مهم)

- Start from the "Start non-terminal" and select a rule from the production rules (CFG)
- If it was not a correct rule, then backtrack and choose another rule.
- If every production is unsuitable for string match, then parse tree cannot be built, and syntax error is reported.



Example1

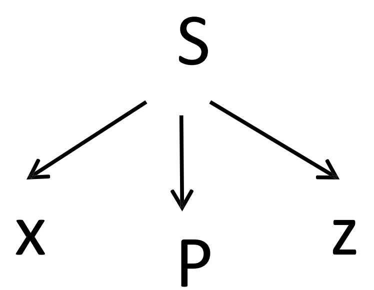
- Consider grammar

  S → xPz

  P → yw | y

  - For Token stream is: **<u>xyz</u>**

1- Select rule S → xPz



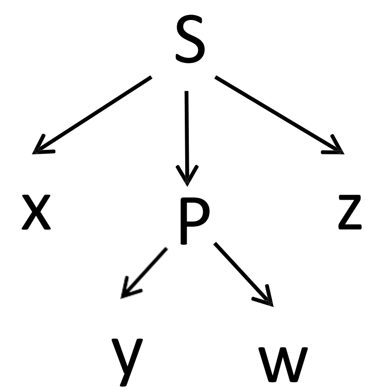
Example1…cont'd

S → xPz

P → yw | y

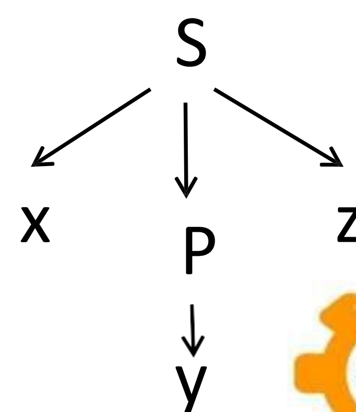
2- Select rule P → yw

- Not correct



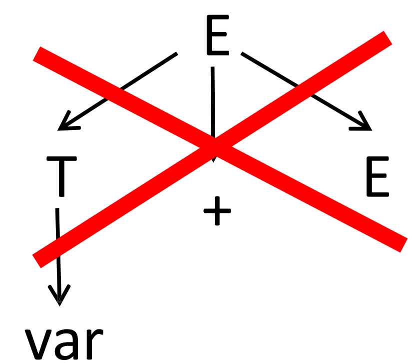
3- Select rule P → y

- Correct



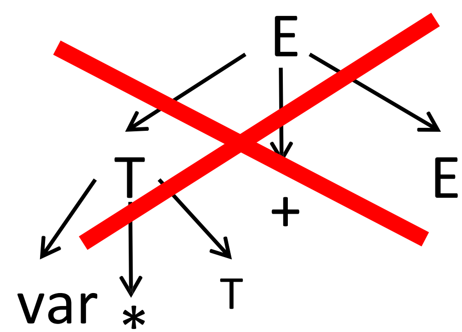
Example2

- Consider the grammar
  - E → T + E | T
  - T → var | var * T
- Token stream is: var*var
- Start with top-level non-terminal E
- Try the rules for E in order
- Try E → T + E
  - Then try a rule for T → var
    - Token matches **var**
    - But **+** after **var** does not match input token *



Example2...cont'd

- Try T → var* T
  - Token matches.
  - This will match but **+** after **T** will be unmatched
- Has exhausted the choices for **E → T + E**

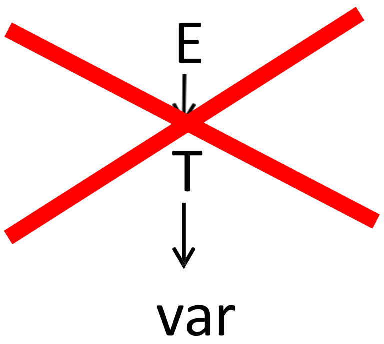


- Backtrack to choice for **E**

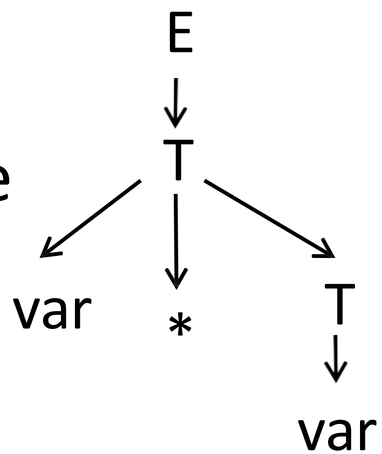
Example2…cont'd

- Try E → T
- Follow same steps as before for T
  - Try a rule for T → var

    Token matches **var**
    - But there is no other token after **var**

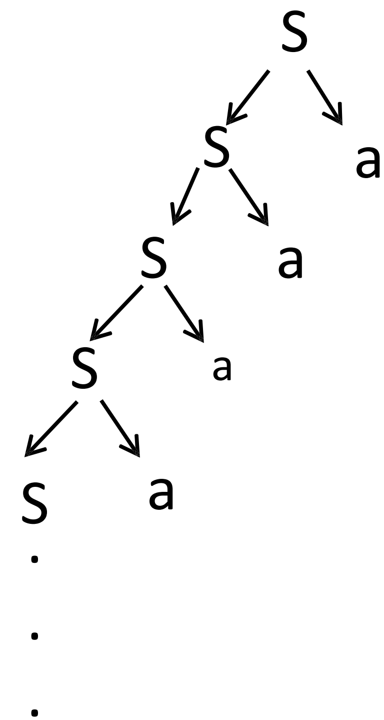


  - Try T → var* T
    - Then try **T→ var**
    - Succeed with the following parse tree



Notes

- Easy to implement by hand.
- But does not always work …
- Consider a production ***S → S a***
- S will get into an infinite loop.
- This case is called **left-recursion**.
- Recursive descent does not work in such cases.



Left Recursion

- A production of grammar is said to have **left recursion** if the leftmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having left recursion is called as Left Recursive Grammar.

Elimination of Left Recursion

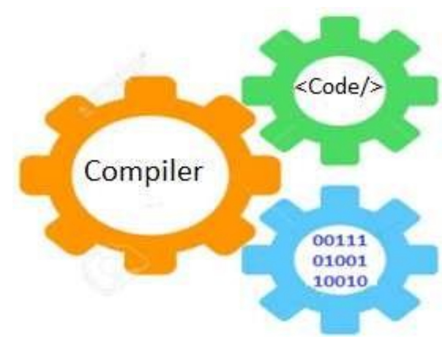
If we have the left-recursive pair of productions-

A → Aα / β

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

A → βA'

A' → αA' / ∈



Example of Elimination of Left Recursion

$E \rightarrow E + T \mid T$

Eliminate immediate left recursion from the Grammar

Solution

Comparing $E \rightarrow E + T \mid T$ with $A \rightarrow A\,\alpha \mid \beta$

$A = E$, $\alpha = +T$, $\beta = T$

$A \rightarrow A\,\alpha \mid \beta$ is changed to $A \rightarrow \beta A'$ and $A' \rightarrow \alpha\, A' \mid \varepsilon$

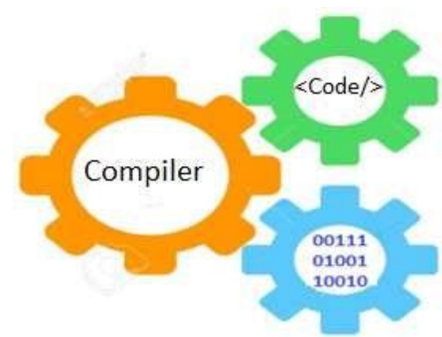
$A \rightarrow \beta A'$ means $E \rightarrow TE'$

$A' \rightarrow \alpha\, A' \mid \varepsilon$ means $E' \rightarrow +TE' \mid \varepsilon$

Result

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$



2- Top-Down Parsing by Predictive parser

- Consider Grammar

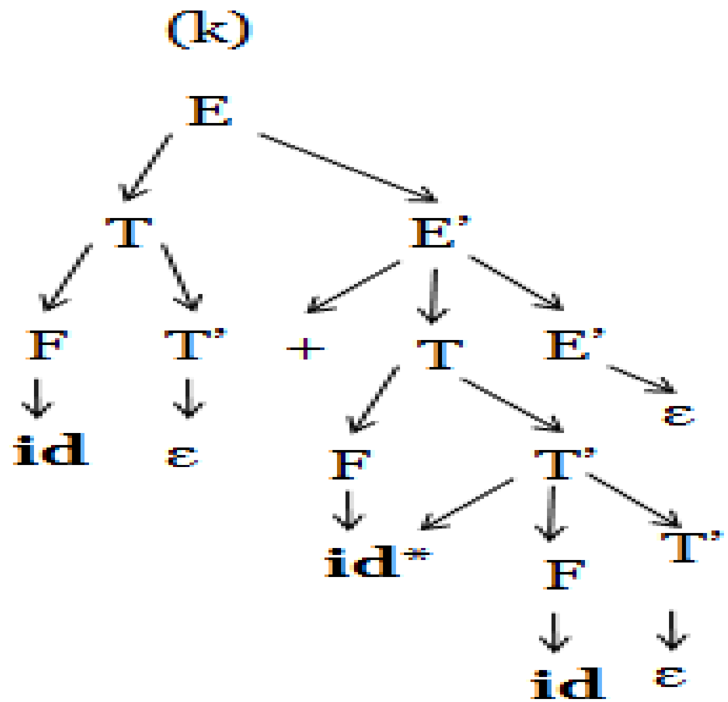
  E → T E’

  E’ → **+** T E’ | ε

  T → F T’

  T’ → ***** F T’ | ε

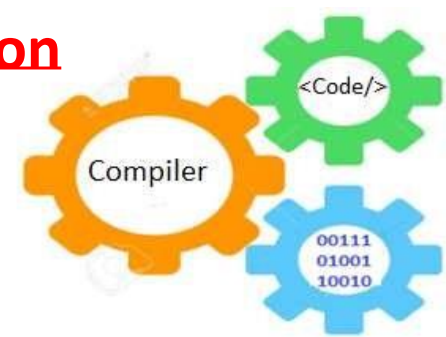
  F → (E) | **id**



- String **id + id * id**

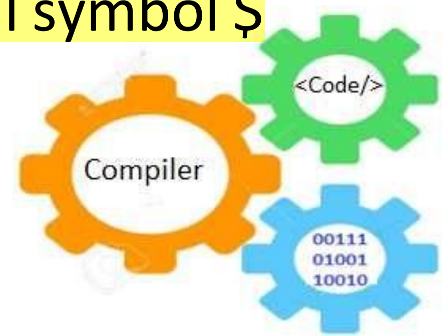
2- Top-Down Parsing by Predictive parser

- Predicts which production to use
  - By looking at the next few tokens, using "lookahead" variable
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means "Left-to-Right" scan of input
  - L means "Leftmost derivation"
  - k means "predict based on k tokens of lookahead"
  - In practice, LL(1) is used
  - LL(k) grammar must be unambiguous
  - LL(k) grammar must not include any **left-recursion**



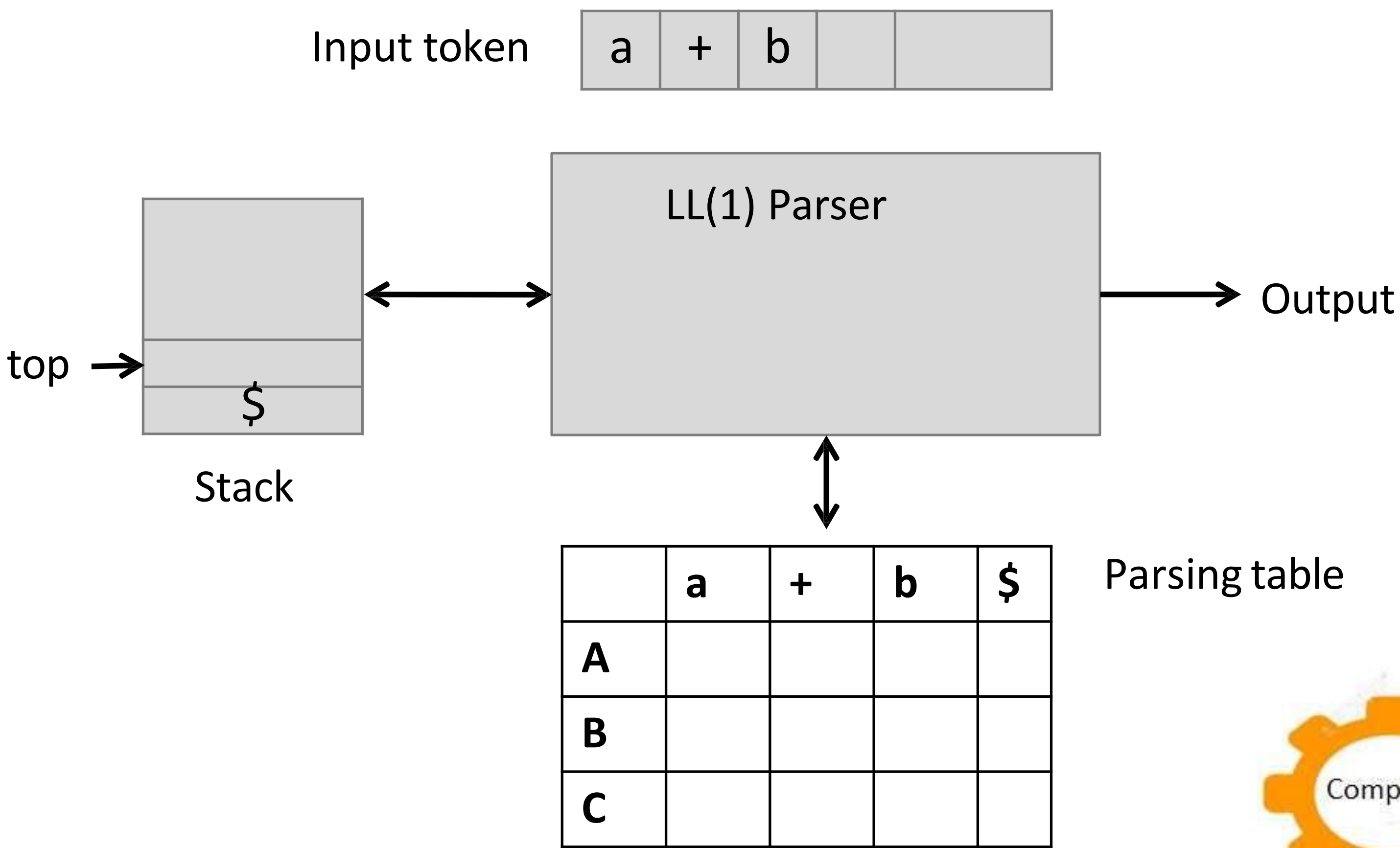
LL(1) Parser

- **input buffer** : The string to be parsed
- **Output:** A production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.
- **Stack**: keeps the grammar symbols, initially contains $.
- The symbols in RHS of rule are pushed into the stack in reverse order i.e. from right to left
- **parsing table**:
  - a two-dimensional array
  - each row is a non-terminal symbol
  - each column is a terminal symbol or the special symbol $
  - each entry holds a production rule.



LL(1) Parser Model

LL(1) Parser

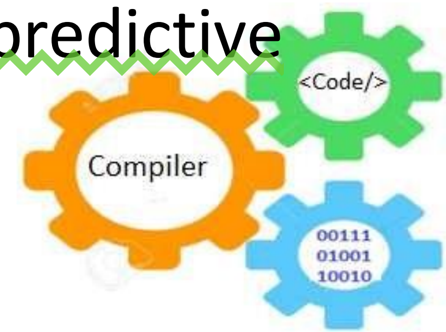


Building Predictive Parser

Three steps

1. Compute FIRST and FOLLOW
2. Construct the predictive parsing table
3. Parse the input string

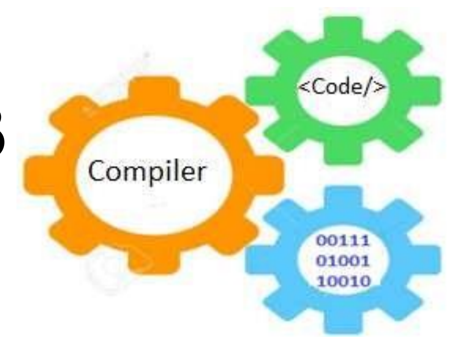
- Note: the grammar must be unambiguous and Left Recursion must be eliminated
- First and follow are used to construct the predictive parsing table



Computing First and Follow

- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α . Where α is any string of grammar (terminals and non-terminals).
- if α derives to ε, then ε is also in FIRST(α) .

- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal A*. If the strings derived from the starting symbol.
  - _ $ is in FOLLOW(A) if $S \Rightarrow \alpha A$
  - a terminal **a** is in FOLLOW(A) if $S \Rightarrow \alpha A a \beta$



Computing FIRST

- FIRST(*a*) = {*a*}   if $a \in T$
- FIRST(ε) = {ε}
- FIRST (X) for a non-terminal X

  If there is production $X \rightarrow Y_1Y_2....Y_k$ then

  - $FIRST(X) = FIRST(Y_1) - \{\varepsilon\}$.
  - But, if $\varepsilon \in FIRST(Y_1)$,then add $FIRST(Y_2)-\{\varepsilon\}$
  - And, if $\varepsilon \in FIRST(Y_2)$,…

Example 1

- E → TE`
  E`→ +TE` | ε
  T → FT`
  T`→ *FT` | ε
  F → id | (E)

- FIRST(E) = FIRST(T) =FIRST(F)= { id, (}
- FIRST(E`) = {+, ε}
- FIRST(T) = FIRST(F)= { id, (}
- FIRST(T`) = {*, ε}
- FIRST(F) = { id, (}

Example 2

- *type* → *simple*

  | **^ id**

  | **array [** *simple* **] of** *type*
- *simple* → **integer**

  | **char**

  | **num dot num**

- FIRST(*simple*) = { **integer**, **char**, **num** }
- FIRST(*type*) = { **integer**, **char**, **num**, **^**, **array** }

Exercise

Find FIRST for the following grammar

- $S \rightarrow ACB \mid CbB \mid Ba$

  $A \rightarrow da \mid BC$

  $B \rightarrow g \mid \varepsilon$

  $C \rightarrow h \mid \varepsilon$

مدام وصول الى العنصر ا وما اكون نشوف يلي بعده

g موجود اصلا في f(s)

ε اذن لازم نشوف الي بعده

$F(S) = \{d, g, h, \varepsilon, b, a\}$

$F(A) = \{d, g, h, \varepsilon\}$

$F(B) = \{g, \varepsilon\}$

$F(C) = \{h, \varepsilon\}$

- $S \rightarrow Aa$

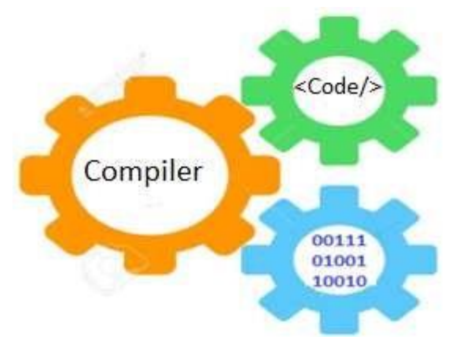
  $A \rightarrow bdZ \mid eZ$

  $Z \rightarrow cZ \mid adZ \mid \varepsilon$

$f(Z) = \{c, a, \varepsilon\}$

$f(A) = \{b, e\}$

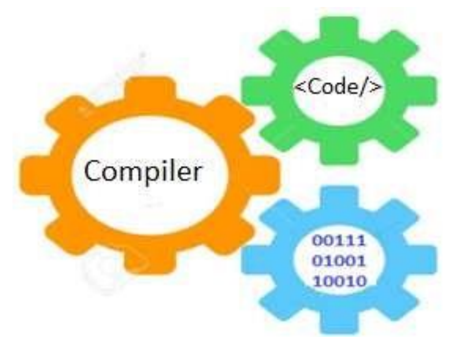
$f(S) = f(A) = \{b, e\}$



Computing FOLLOW

- If S is the start symbol ➔ \$ is in FOLLOW(S)
- if $A \rightarrow \alpha B \beta$ is a production rule
  ➔ everything in FIRST(β) is FOLLOW(B) except ε
- If ($A \rightarrow \alpha B$ is a production rule) or
  ($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))
  ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set



Example 1

E → T E’

E’→ + T E’ | ε

T → F T’

T’→ * F T’ | ε

F → (E) | id

- FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

  FIRST(E’) = {+, ε}

  FIRST(T’) = {*, ε}

- FOLLOW(E) = {) , $}

  FOLLOW(E’) = {) , $}

  FOLLOW(T) = {+,), $}

  FOLLOW(T’) = {+,), $}

  FOLLOW(F) = {*, +,), $}

Example 2

E → T E'

E' → + T E' | - T E' | ε

T → F T'

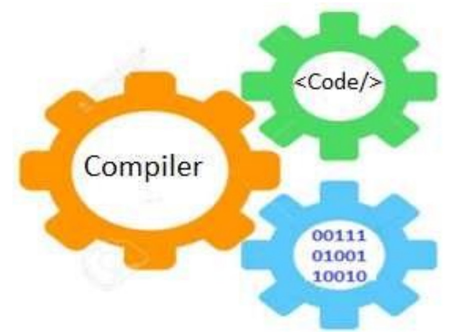
T' → * F T' | / F T' | ε

F → num | id

	First	Follow
E	{num , id}	{$}
E'	{+, -, ε}	{$}
T	{num , id}	{+, -, $}
T'	{*, /, ε}	{+, -, $}
F	{num , id}	{*, /, +, -, $}

Notes

- To compute FIRST(*A*) you must look for A on a production's **left-hand** side.
- To compute FOLLOW(*A*) you must look for A on a production's **right-hand** side.
- FIRST sets are always sets of **terminals** (plus, perhaps epsilon).
- FOLLOW sets are always sets of **terminals** (plus, perhaps $).
- Nonterminals are ***never*** in a FIRST or a FOLLOW set.
- epsilon is ***never*** in a FOLLOW set.



Constructing the Parse Table

- Parse table summarizes the applicable RHS for each terminal/non-terminal combination.
- Construct a parsing table T for CFG :
- For each production $X \rightarrow \alpha$
  - Add $\rightarrow \alpha$ to the X row for each symbol in FIRST(α)
  - If α is nullable, add $\rightarrow \alpha$ for each symbol in FOLLOW(X)
  - Entry for [S, \$] is ACCEPT
  - All other undefined entries of the parsing table are error entries.

Parse table Example (1)

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'\ |\ \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'\ |\ \varepsilon$

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

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

Parse table Example 1

- Create table with:
  - Put each terminals in the columns
  - Put each non-terminals to rows

	+	*	(	)	id	$
E						
E’						
T						
T’						
F						

Parse table Example 1

- For production E → T E’
  - Add E → T E’ to the *E* row for each symbol in FIRST(E)

	+	*	(	)	id	$
E			E → T E’		E → T E’	
E’						
T						
T’						
F						

Parse table Example 1

- For production E’→ + T E’ | ε
  - E’→ + T E’, Add → + T E’ to the *E’* row for each symbol in FIRST(E’)
  - E’ → ε, add → ε for each symbol in FOLLOW(*E’*)

	+	*****	**(**	**)**	**id**	**$**
E			E → T E’		E → T E’	
E’	E’→ + T E’			E’ → ε		E’ → ε
T						
T’						
F						

Parse table Example 1

- For production T → F T'
  - Add T → F T' to the *T* row for each symbol in FIRST(T)

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

Parse table Example 1

- For production T’→ * F T’ | ε
  - Add T’→ * F T’ to the *T’* row for each symbol in FIRST(*T’*)
  - Add T’ → ε for each symbol in FOLLOW(*T’*)

	+	*	(	)	id	$
E			E → T E’		E → T E’	
E’	E’→ + T E’			E’ → ε		E’ → ε
T			T → F T’		T → F T’	
T’	T’ → ε	T’→ * F T’		T’ → ε		T’ → ε
F						

Parse table Example 1

- For production F → (E) | id
  - Add F → (E) to the *F* row and symbol **(**
  - Add F → id to the *F* row and symbol **id**

	+	*****	**(**	**)**	**id**	**$**
E			E → T E’		E → T E’	
E’	E’→ + T E’			E’ → ε		E’ → ε
T			T → F T’		T → F T’	
T’	T’ → ε	T’→ * F T’		T’ → ε		T’ → ε
F			F → (E)		F → id	

Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.

- **There are four possible parser actions.**

1. If X and a are \$ ➔ parser halts (successful completion)

2. If X and a are the same terminal symbol (different from \$)
   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.

3. If X is a non-terminal
   ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$ to represent a step of the derivation.

4. none of the above ➔ error
   - all empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.

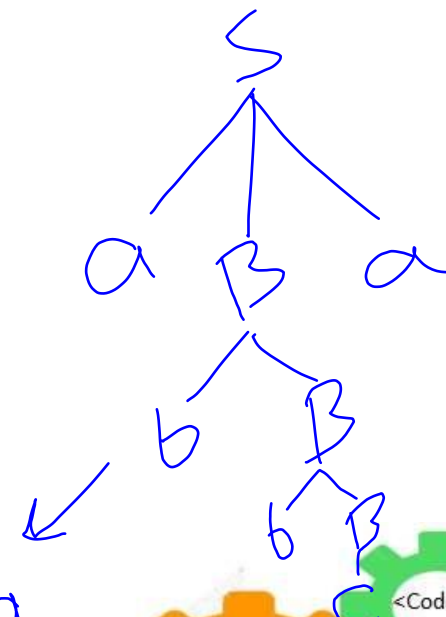
LL(1)Parser Example

$S \rightarrow aBa$

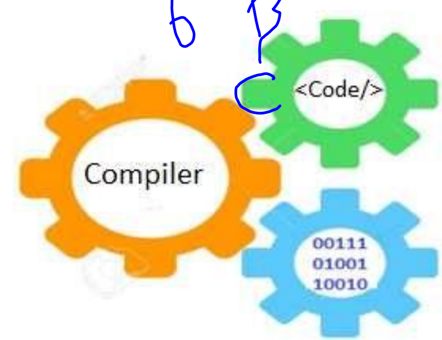
$B \rightarrow bB \mid \varepsilon$

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

stack	input	output
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	c bba$ → synactical error	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion



abba



LL(1) Parser – Example2

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

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

LL(1) Parser – Example2…cont’d

Input is id+id

stack	input	output
$E	id+id$	$E \to TE'$
$E′T	id+id$	$T \to FT'$
$E′ T′F	id+id$	$F \to id$
$ E′ T′id	id+id$	
$ E′ T′	+id$	$T' \to \varepsilon$
$ E′	+id$	$E' \to +\cancel{FT'}$ TE′
$ E′ T+	+id$	
$ E′ T	id$	$T \to FT'$
$ E′ T′ F	id$	$F \to id$
$ E′ T′id	id$	
$ E′ T′	$	$T' \to \varepsilon$
$ E′	$	$E' \to \varepsilon$
$	$	accept