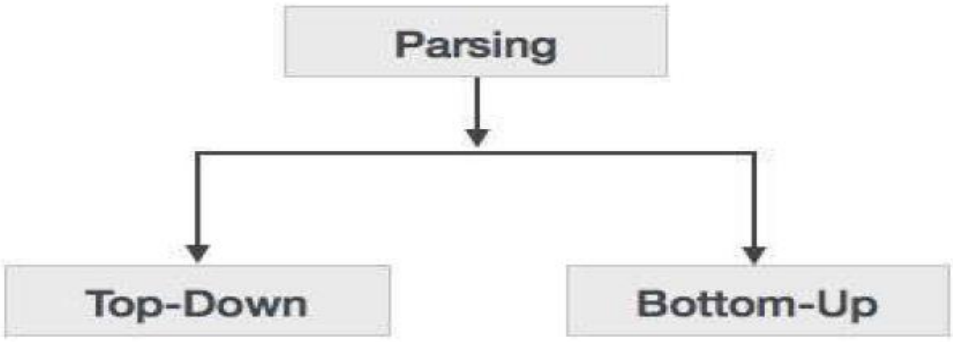
Parsing Techniques

Parsers

A ***parser*** for grammar ***G*** is a program that takes as input a string ***w*** and produces as output either a parse tree for ***w***, if ***w*** is a sentence of ***G***, or an error message indicating that ***w*** is not a sentence of ***G***.

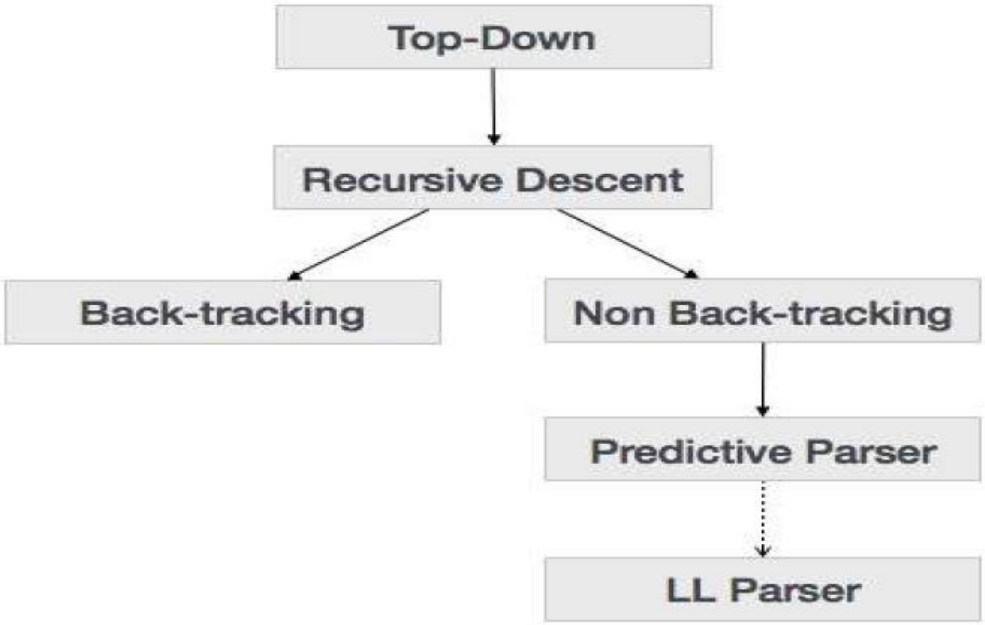
There are two basic types of **parsers** for context-free grammars are **Top-Down** and **Bottom-Up**. As indicated by their names, top-down parsers start with the root and work down to the leaves, while bottom-up parsers build parse trees from the bottom (leaves) to the top (root). In both cases the input to the parser is being scanned from ***left to right***, one symbol at a time.



Top-Down Parsing

Top-down parsing can be viewed as an attempt to find a ***leftmost derivation*** for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

The general form of top-down parsing, called recursive descent, the recursive descent can be divided to two cases. First case that may involve ***Backtracking***, which is, making repeated scans of the input and second case, is ***No Backtracking*** (***Predictive Parser***). The types of top-down parsing are depicted below:



Backtracking: It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

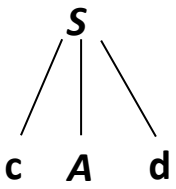
Example: Consider the grammar

$$S \longrightarrow \mathbf{c}A\mathbf{d}$$

$$A \longrightarrow \mathbf{ab} \mid \mathbf{a}$$

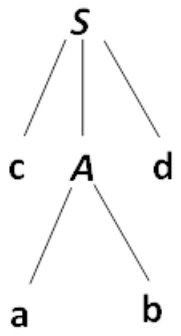
and the input string ***w*** = **cad**. To construct a parse tree for this string top-down.

1) Create a tree consisting of a single node labeled ***S***.
2) An input pointer points to **c**, the first symbol of ***w***. We then use the first production for ***S*** to expand the tree and obtain the tree of Figure below.



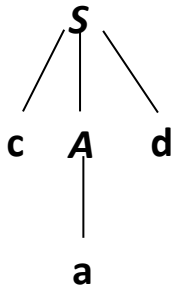
The leftmost leaf, labeled **c**, matches the first symbol of ***w***.

3) Advance the input pointer to **a**, the second symbol of ***w***, and consider the next leaf, labeled **A**. We can then expand **A** using the first alternative for **A** to obtain the tree of Figure below. We now have a match for the second input symbol.



4) Advance the input pointer to **d**, the third input symbol, and compare **d** against the next leaf, labeled **b**. Since **b** does not match **d**, we report failure and **go back to *A*** to see whether there is another alternative for ***A*** that we have not tried but that might produce a match.

5) In going back to ***A***, we must reset the input pointer to position **2**, the position it had when we first came to ***A***, we now try the second alternative for ***A*** to obtain the tree of figure below.



The leaf a matches the second symbol of w and the leaf d matches the third symbol. Since we have produced a parse tree for ***w***, we halt and announce successful completion of parsing.

Note: A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop. That is, when we try to expand ***A***, we may eventually find ourselves again trying to expand ***A*** without having consumed any input.

Example2: Consider the following CFG:

S → rXd | rZd

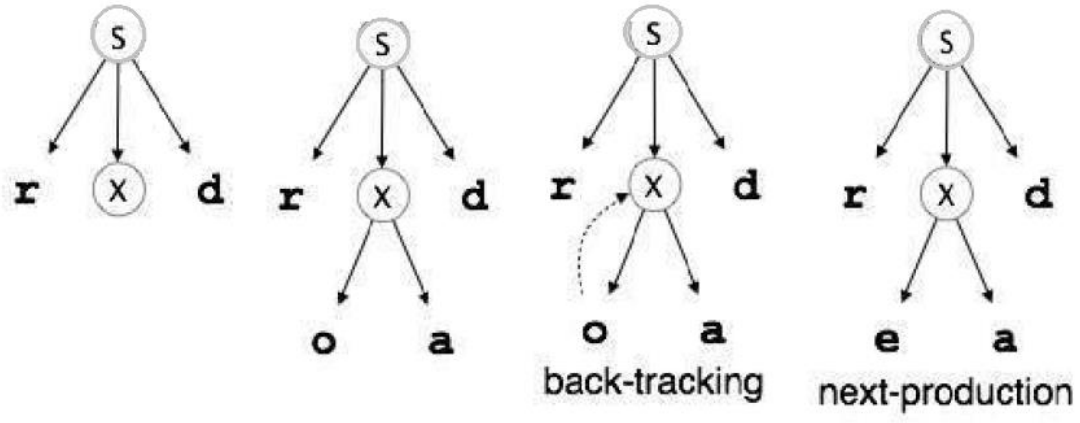
X → oa | ea

Z → ai

For an input string: read, a top-down parser will behave like this:

It will start with **S** from the production rules and will match its yield to the left-most letter of the input, i.e. '**r**'. The very production of ***S → rXd*** matches with it. So the top-down parser advances to the next input letter *i. e.* '***e***′. The parser tries to expand non-terminal '**X**' and checks its production from the left **X → oa**. It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of **X**, ***X → ea***.

Now the parser matches all the input letters in an ordered manner. The string is accepted.



Predictive Parser

In many cases, by carefully writing a grammar, ***eliminating left recursion*** from it, and ***left factoring*** the resulting grammar, we can obtain a grammar that can be parsed by a **recursive-descent parser** that needs **no backtracking**, i.e., a **predictive parser**.

Transition Diagrams for Predictive Parser

Transition diagrams are useful for visualizing predictive parsers. Several differences between the transition diagrams for a lexical analyzer and a predictive parser are immediately apparent.

In the case of the parser, there is one diagram for each **nonterminal**. The labels of edges are **tokens** (**terminal**) and **nonterminals**. A transition on a token (**terminal**) means we should take that transition if that token is the next input symbol. A transition on a nonterminal, ***A*** is a call of the procedure for ***A***.

*To construct the transition diagram of a predictive parser from a grammar, first **eliminate left recursion** from the grammar, and then **left factor** the grammar. Then for each **nonterminal A** do the following:*

1) Create an **initial** and **final** (return) state.

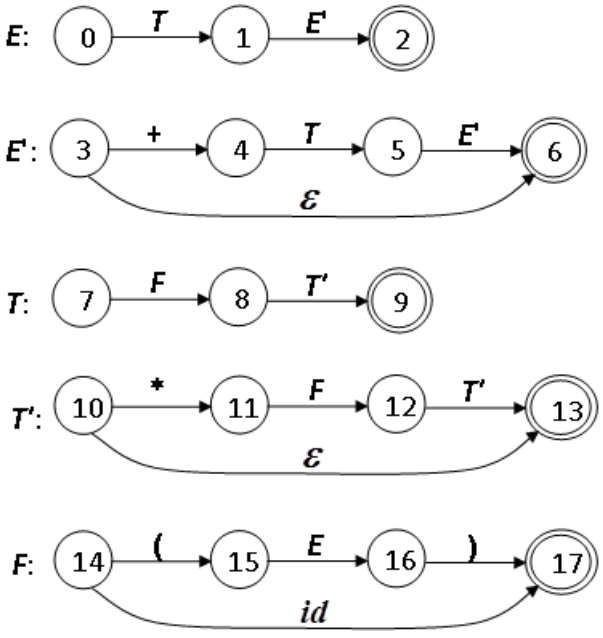
2) For each production $A \longrightarrow X_1X_2 \ldots X_n$, create a **path** from the **initial** to the **final** state, with edges labeled $X_1X_2 \ldots X_n$.

The predictive parser working off the transition diagrams behaves as follows. It begins in the start state for the start symbol. If after some actions it is in state **s** with an edge labeled by terminal **a** to state **t**, and if the next input symbol is **a**, then the parser moves the input cursor one position right and goes to state **t**. If, on the other hand, the edge is labeled

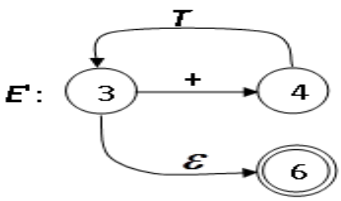
by a nonterminal **A**, the parser instead goes to the start state for **A**, without moving the input cursor. If it ever reaches the final state for **A**, it immediately goes to state **t**, in effect having "read" **A** from the input during the time it moved from state **s** to **t**. Finally, if there is an edge from **s** to **t** labeled **ϵ**, then from state **s** the parser immediately goes to state **t**, without advancing the input.

Example: Design the transition diagram of predictive parser for the following grammar:

$$E \longrightarrow TE'$$
$$E' \longrightarrow +TE' \mid \in$$
$$T \longrightarrow FT'$$
$$T' \longrightarrow *FT' \mid \in$$
$$F \longrightarrow (E) \mid \mathbf{id}$$



The figure in below shows an equivalent transition diagram for ***E'***.

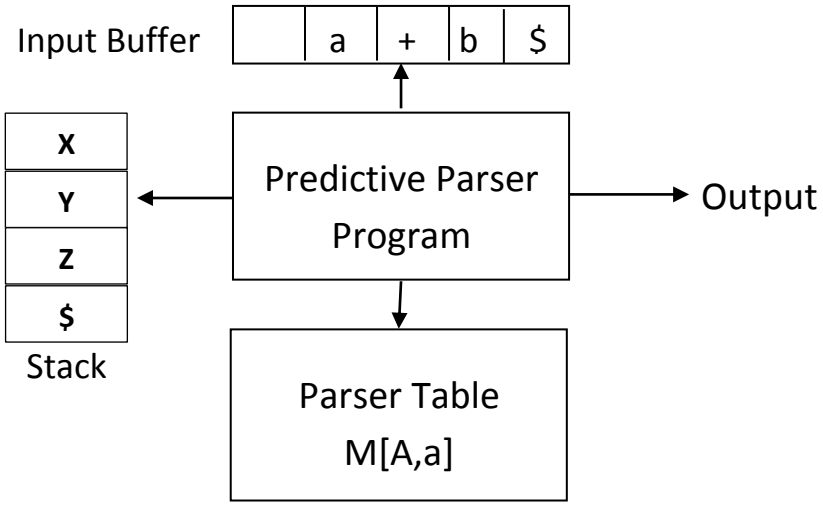


Simplified Transition diagram

Predictive parser is a recursive descent parser, which has the capability to predict which ***production*** is to be used to replace the input string. The predictive parser does ***not suffer from backtracking***. To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(*k)* grammar.

Components of Predictive Parser

Predictive parser has an input buffer, a stack, a parsing table, and an output stream. The model of predictive parser is shown the in figure below:



Model of Predictive Parser

1) The input buffer contains the string to be parsed, followed by **$**, a symbol used as a right end-marker to indicate the end of the input string.

2) The stack contains a sequence of grammar symbols with **$** on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of **$**.

3) The parsing table is a two-dimensional array **M [A, a]**, where **A** is a nonterminal, and **a** is a terminal or the symbol **$**.

The parser is controlled by a program that behaves as follows. The program considers ***X***, the symbol on top of the stack, and ***a***, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

a) If ***X* = a = $**, the parser halts and announces successful completion of parsing.

b) If ***X* = a ≠ $**, the parser pops ***X*** off the stack and advances the input pointer to the next input symbol.

c) If ***X*** is a nonterminal, the program consults entry ***M[X, a]*** of the parsing table ***M***. This entry will be either an ***X***-production of the grammar or an error entry. If, for example, ***M[X, a]*** = {***X*** $\longrightarrow$ ***UVW***}, the parser replaces ***X*** on top of the stack by ***WVU*** (with ***U*** on top).

4) As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X, a] = error, the parser calls error recovery routine.

Construction of Predictive Parsing Tables

The following algorithm can be used to construct a predictive parsing table for a grammar ***G***.

Algorithm: Construction of a predictive parsing table.

Input: Grammar ***G***.

Output: Parsing table ***M***.

Method:

1. For each production ***A*** $\longrightarrow$ **α** of the grammar, do steps 2 and 3.
2. For each terminal ***a*** in FIRST (**α**), add ***A*** $\longrightarrow$ **α** to ***M [A, a]***.

3. If ϵ is in FIRST (**α**), add ***A*** ⟶ **α** to ***M [A, b]*** for each terminal **b** in FOLLOW (***A***). If ϵ is in FIRST (**α**) and $ is in FOLLOW (***A***), add ***A*** ⟶ **α** to ***M [A, $]***.

4. Make each undefined entry of ***M*** be ***error***.

Example: Parse the string **id + id * id** by using predictive parser for the following grammar: ***E -> E + T | T ; T -> T * F | F ; F -> (E) | id***

E ⟶ ***TE'***
E' ⟶ ***+TE'*** | ∈
T ⟶ ***FT'***
T' ⟶ ****FT'*** | ∈
F ⟶ **(*E*) | id**

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

FIRST (***E'***) = {**+,** ∈**}**

FIRST (***T'***) = {***,** ∈**}**

FOLLOW (***E***) = FOLLOW (***E'***) = {), $}

FOLLOW (***T***) = FOLLOW (***T'***) = {+,), $}

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

Predictive Parsing Table *M* For Above Grammar

i/p Symb. / NonTerm.	id	+	*	(	)	$
E	***E -> TE'***			***E -> TE'***		
E'		***E' -> +TE'***			***E' -> ∈***	***E' -> ∈***
T	***T -> FT'***			***T -> FT'***		
T'		***T' -> ∈***	***T' -> *FT'***		***T' -> ∈***	***T' -> ∈***
F	***F ⟶ id***			***F ⟶ (E)***		

Note*: Blanks are error entries**; ***non-blanks indicate a production with which to expand the top nonterminal on the stack.

Moves made by predictive parser on the input id + id * id is:

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*

Predictive parser accepts the given input string. We can notice that **$** in ***input*** and ***stuck***, i.e., both are ***empty***, hence **accepted**.

Note: In ***recursive descent parsing***, the parser may have more than one production to choose from for a single instance of input, whereas in ***predictive parser***, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

LL (1) Grammars

Algorithm construction of a predictive parsing table can be applied to any grammar ***G*** to produce a parsing table ***M***. For some grammars, however, ***M*** may have some entries that are **multiply-defined**. For example: if ***G*** is **left recursive** or **ambiguous**, then ***M*** will have at least one **multiply-defined** entry.

Example: Let us consider the following grammar:

$S \longrightarrow iEtSS' \mid a$
$S' \longrightarrow eS \mid \in$
$E \longrightarrow b$

NonTerminal	First	Follow
S	**{i, a}**	**{e**, $}
S'	**{e,** ∈**}**	**{e**, $}
E	**{b}**	{t}

Predictive Parsing Table **M** For the above Grammar is

NONTER-MINALS	INPUT SYMBOL					
	a	b	e	i	t	$
S	$S \longrightarrow a$			$S \longrightarrow iEtSS'$		
S'			$S' \longrightarrow \in$ $S' \longrightarrow eS$			$S' \longrightarrow \in$
E		$E \longrightarrow b$				

From the **M**- table, we can see:

The entry for M[S',e] contains both $S' \longrightarrow eS$ and $S' \longrightarrow \varepsilon$, since FOLLOW(S') = {e, $}. The grammar is ***ambiguous*** and the ***ambiguity*** is manifested (تجلى) by a choice in what production to use when an e is seen. Therefore this grammar is **not LL (1)**.

Definition of LL (1):

A grammar whose parsing table has no multiply-defined entries is said to be 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 lookahead at each step to make parsing action decisions. LL (1) grammars have several distinctive properties. **No ambiguous** or **left- recursive** grammar can be LL (1).

The grammar is LL (1) if satisfy the following Conditions :

For all productions $A \longrightarrow \alpha_1 \mid \alpha_2 \mid \ldots\ldots \mid \alpha_n$

1. FIRST (α_i) ∩ FIRST (α_j) = ϕ for all $i \neq j$ and
2. If $\alpha_i \xrightarrow{*} \in$, Then FIRST ($\alpha_j$) ∩ FOLLOW($A$)= ϕ for all $i \neq j$

Example1: Is the following grammar LL (1)?

$A \longrightarrow iBte$
$B \longrightarrow SB \mid \in$
$S \longrightarrow [ec] \mid \bullet i$

Solution:

Rule 1:

$B \longrightarrow SB \mid \in$

FIRST (SB) ∩ FIRST ($\in$) = {[, •} ∩ {$\in$} = ϕ

$S \longrightarrow [ec] \mid \bullet i$

FIRST ([ec]) ∩ FIRST (•i) = {[} ∩ {•} = ϕ

Rule 2:

$B \longrightarrow SB \mid \in$

FIRST (SB) ∩ FOLLOW (B) = {[, •} ∩ {t} = ϕ

This grammar is LL (1).

Example2: Is the following grammar LL (1)?

$S \longrightarrow XS \mid aY$
$X \longrightarrow a \mid b$
$Y \longrightarrow (S)$

Sol:
Rule 1:

$S \longrightarrow XS \mid aY$

FIRST (XS) ∩ FIRST (aY) = {**a, b**} ∩ {**a**} = {**a**}

This grammar is **not LL (1)**. And it is not suitable for constructing parser table.

Example3: Is the following grammar LL (1)?

$S \longrightarrow Aa \mid bB$
$A \longrightarrow aBmS \mid C$
$B \longrightarrow (S)$
$C \longrightarrow \in$

Sol:
Rule 1:

$S \longrightarrow Aa \mid bB$

FIRST (Aa) ∩ FIRST (bB) = {**a,** ∈**}** ∩ {**b**} = ϕ

$A \longrightarrow aBmS \mid C$

FIRST ($aBmS$) ∩ FIRST (C) = {**a**} ∩ {∈} = ϕ

Rule 2:

$A \longrightarrow aBmS \mid C$ Since $C \longrightarrow \in$, Then

FIRST ($aBmS$) and FOLLOW (A) must be disjoint.

FIRST ($aBmS$) ∩ FOLLOW (A) = {**a**} ∩ {**a**} = {**a**} ≠ ∅

This grammar is not LL (1).

H.W: Which of the following grammars is LL (1) or not?

1) S → aSbS | bSaS | ϵ

2) S → aABb ; A → c | ϵ ; B → d | ϵ

3) S → A | a ; A → a

4) S → aB | ϵ ; B → bC | ϵ ; C → cS | ϵ

5) S → AB ; A → a | ϵ ; b | ϵ

6) S → aSA | ϵ ; A → c | ϵ

7) S → A ; A → Bb | Cd ; B → aB | ϵ ; C → cC | ϵ

8) S → aA | a | ϵ ; A → abS | ϵ

9) S → iEtSS' ; S' → eS | ϵ ; E → b

Error recovery in predictive parsing

- ***An error may occur in the predictive parsing (LL(1) parsing)***
  - if the ***terminal symbol on the top of stack*** does not match with the current ***input symbol***.
  - if the ***top of stack is a non-terminal A***, the current ***input symbol*** is ***a,*** and the parsing table entry ***M[A,a] is empty***.
- ***What should the parser do in an error case?***
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should be recovered from that error case, and it should be able to continue the parsing with the rest of the input.
  - ***Panic-mode error recovery*** is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens.
  - ***Synchronizing token***: All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.

- "synch" indicating synchronizing tokens obtained from FOLLOW set of the nonterminal in question.
- How the error- recovery in LL (1) Parsing work?
  - ✓ If the parser looks up entry ***M [A, a]*** and finds that it is blank, the input symbol ***a*** is skipped.
  - ✓ If the entry is ***synch***, then the ***nonterminal*** on top of the stack is popped.
  - ✓ If a token on top of the stack does not match the input symbol, then we pop the token from the stack.

Example:

Consider the following

First (E) = First (T) = First (F) = {(, id}

First (E') = {+, ϵ}

First (T') = {*, ϵ}

Follow (E) = Follow (E') = {), $}

Follow (T) = Follow (T') = {+,), $}

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

Parsing Table with Synchronizing Tokens

	id	+	*	(	)	$
E	***E->TE'***			***E->TE'***	***synch***	***synch***
E'		***E'->+TE'***			***E'->ϵ***	***E'->ϵ***
T	***T->FT'***	***synch***		***T->FT'***	***synch***	***synch***
T'		***T'->ϵ***	***T'->*FT'***		***T'->ϵ***	***T'->ϵ***
F	***F->id***	***synch***	***synch***	***F->(E)***	***synch***	***synch***

Consider the following input symbols:

Input symbols=) id * + id $

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'	$	
$	$	

Parsing and error recovery moves made by predictive parser

H.W

1) Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

S → iEtSS' | a

S' → eS | ϵ

E → b

2) Construct the FIRST and FOLLOW and predictive parse table for the grammar:

S → AC$; C →c | ϵ; A → aBCd | BQ | ϵ ; B → bB | d ; Q → q

Then, check whether the given input string (abdcdc$) is accepted by the predictive parser or not?