2- Left Recursion

A grammar is left recursion if it has a non terminal A, such that there is a derivation $A \longrightarrow \alpha A$ Fer some string α. Top-down parsing methods cannot handle left recursion grammars, so a transformation that eliminates left recursion is needed.

In the following example, we show how that left recursion pair of production

$A \longrightarrow \alpha A|\beta$ could be replaced by the non left recursional productions:

$A \longrightarrow \alpha A|\beta$

$A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A'|\lambda$

Example: Consider the following grammar for arithmetic expressions.

$E \longrightarrow E+T|T$

$T \longrightarrow T*F|F$

$F \longrightarrow (E) | id$

Eliminating the immediate left recursion (productions of the form $A \longrightarrow A\alpha$ to the production for E and then for T, we obtain:

$E \longrightarrow TE'$

$E' \longrightarrow +T E'|\lambda$

$T \longrightarrow FT'$

$T' \longrightarrow *FT' | \lambda$

$F \longrightarrow (E) | id$

No matter how many A productions there are, we can eliminate immediate left recursion from them by the following technique. First we group the A production as

$A \longrightarrow A\alpha 1| A\alpha 2| \ldots| A\alpha n| \beta 1| \beta 2| \ldots \beta n$

where no βi begins with an A. then, we replace the A -productions by

$A \longrightarrow \beta 1A'| \beta 2A'| \ldots| \beta nA'$

$A \longrightarrow \alpha 1A'| \alpha 2A'| \ldots| \alpha nA' | \lambda$

This produce eliminates all immediate left recursion from A and A' production. but it does not eliminate left recursion involving derivation of two or more steps.

S ⟶ Aa | b
A ⟶ Ac | Sd | λ
The non terminal S is left recursion because S ⟶ Aa ⟶ Sda, but is not immediately left recursion.

3- Left Factoring

left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.
For example, if we have the two productions

Stmt ⟶ if Expr then Stmt else Stmt
| if Expr then Stmt

on seeing the input token if, we cannot immediately tell which production to choose to expand stmt. In general, if A ⟶ α β1| α β2 are two A productions, and the input begins with a non-empty string derived from α ,we do not know whether to expand A to α β1 or to α β2. However, we may defer the decision by expanding A to αA', then after seeing the input derived from α, we expand A' to β1 or to β2. that is, left factored, the original production become:

A ⟶ αA'
A' ⟶ β1| β2

Top down parser

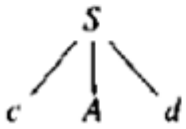
In this section there are basic ideas behind top-down parsing and show how constructs an efficient non- backtracking form of top-down parser called a predictive parser.

Top down parsing can be viewed as attempt to find a left most 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.

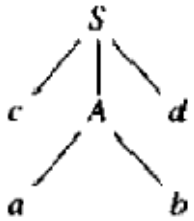
The following grammar requires **backtracking**:

$S \rightarrow cAd$
$A \rightarrow ab \mid a$

the input string $w = cad$. To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled S. 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

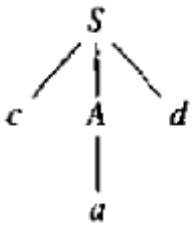


The leftmost leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree



We now have a match for the second input symbol so we 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.

We now try the second alternative for A to obtain the tree:



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.

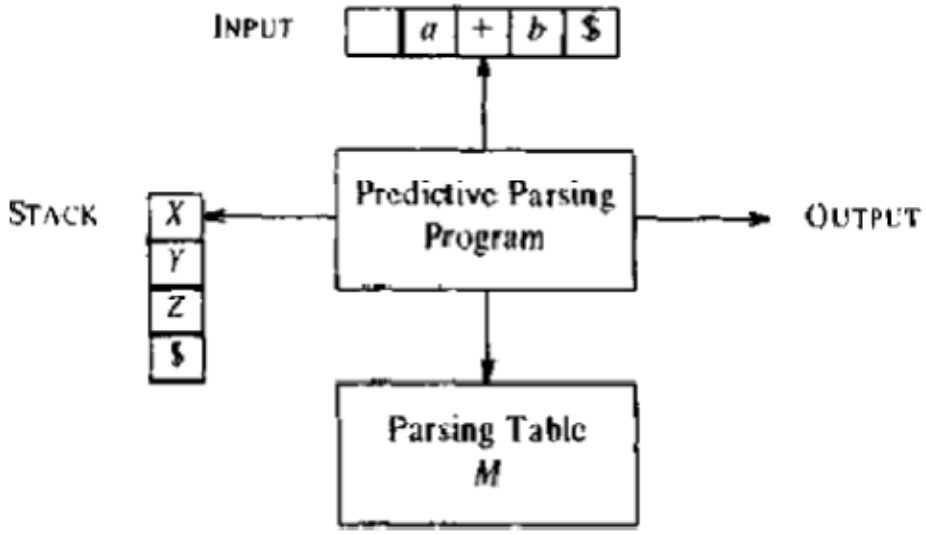
Predictive Parsing Method

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 non backtracking predictive parser.

We can build a predictive parser by maintaining a stack. The key problem during predictive parser is that of determining the production to be applied for a nonterminal. The nonrecursive parser looks up the production to be applied in a parsing table.

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, (a symbol used as a right endmarker to indicate the end of the input string). 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

the top of $. 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.

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. 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, $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU (with U on top).

If M[X,a]= error, the parser calls an error recovery routine.

Algorithm nonrecusive predictive parser

Input. A string w and a parsing table M for grammar G.

Output. If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

Method. Initially, the parser is in a configuration in which it has \$$S$ on the stack with S, the start symbol of G on top, and w\$ in the input buffer.

```
– set ip to point to the first symbol of w$;
  repeat
       let X be the top stack symbol and a the symbol pointed to by ip;
       if X is a terminal or $ then
            if X = a then
                 pop X from the stack and advance ip
            else error()
       else      /* X is a nonterminal */
            if M[X, a] = X → Y1Y2 · · · Yk then begin
                 pop X from the stack;
                 push Yk, Yk-1, . . . , Y1 onto the stack, with Y1 on top;
                 output the production X → Y1Y2 · · · Yk
            end
            else error()
  until X = $    /* stack is empty */
```

Example:

Parse the input **id * id + id** in the grammar:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | 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 parser 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

FIRST and FOLLOW:

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible.

Define the FIRST(α) to be the set of terminals that begin the strings derived from α, and the FOLLOW(A) for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form.

To compute FIRST(x) for all grammar symbols x, apply the following rules until no more terminals or **ε** can be added to any first set.

1- If x is terminal, then FIRST(x) is {x}.

2- If X→ a ; is a production, then add a to FIRST(X) and

If X → **ε** ; is a production, then add **ε** to FIRST(X).

3- If X is nonterminal and X→Y1,Y2…Yi ; is a production, then add FIRST(Y1) to FIRST(X).

4- a- for (i = 1; if Y_i can derive epsilon **ε**; i++)
b- add First(Y_{i+1}) to First(X)
If Y1 does not derive **ε** , then we add nothing more to FIRST(X), but if

Y1→ **ε** , then we add FIRST(Y2) and so on .

First function example

1- FIRST (terminal) = {terminal }

S → aSb | ba | ε

FIRST (a) ={a}

FIRST (b) ={b}

2- FIRST(non terminal) = FIRST (first char)

FIRST (S)= {a,b, **ε** }

To compute FOLLOW(A) for all non terminals A, is the set of terminals that can appear immediately to the right of A in some sentential form S → aAxB... To compute Follow, apply these rules to all nonterminals in the grammar:

1- Place $ in FOLLOW(S) , where S is the start symbol and $ is the input right end marker.

FOLLOW(START) = {$}

2- If there is a production X→ α Aβ , then everything in FIRST(β) except for **ε**

is placed in FOLLOW(A).

i.e. FOLLOW(A) = FIRST(β)

3- If there is a production X→ α A, or a production X→ α Aβ , where FIRST(β)

Contains ε (β → **ε**), then everything in FOLLOW(X) is in FOLLOW(A).

i.e. : FOLLOW(A)= FOLLOW(X)

Follow function examples:

Example 1:

S → aSb | X

X → cXb | b

X → bXZ

Z → **n**

	First	**Follow**
S	a , c , b	$, b
X	c , b	b , n , $
Z	n	b , n , $

Example 2:

S → bXY

X → **b | c**

Y → **b | ε**

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

Example 3:

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

Example 4:

X → ABC │ nX

A → bA │ bb │ ε

B → bA │CA

C → ccC │CA │ cc

	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.

S → bSX | Y

X → bbX' **X' → CX'| ε**

Y → b | bY **C → ccC | CX | cc**

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

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→∈ . In this case, we should again expand A by α if the current input symbol is in FOLLOW(A).

Algorithm construction of predictive parsing table:

Input. Grammar G.

Output. Parsing table M.

Method.

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be **error**.

Example:

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:

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\}$.

$\text{FIRST}(E') = \{+, \epsilon\}$

$\text{FIRST}(T') = \{*, \epsilon\}$

$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{), \$\}$

$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{+,), \$\}$

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

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

For example, **id** and left parenthesis are added to FIRST(F) by rule (3) in the definition of FIRST with $i = 1$ in each case, since FIRST(**id**) = {**id**} and FIRST('(') = { (} by rule (1). Then by rule (3) with $i = 1$, the production $T \to FT'$ implies that **id** and left parenthesis are in FIRST(T) as well. As another example, ϵ is in FIRST(E') by rule (2).

To compute FOLLOW sets, we put $ in FOLLOW(E) by rule (1) for FOLLOW. By rule (2) applied to production $F \to (E)$, the right parenthesis is also in FOLLOW(E). By rule (3) applied to production $E \to TE'$, $ and right parenthesis are in FOLLOW(E'). Since $E' \stackrel{*}{\Rightarrow} \epsilon$, they are also in FOLLOW(T). For a last example of how the FOLLOW rules are applied, the production $E \to TE'$ implies, by rule (2), that everything other than ϵ in FIRST(E') must be placed in FOLLOW(T). We have already seen that $ is in FOLLOW(T). □

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'→ ∈ causes M[E',)] and M[E',$] to acquire E'→ ∈ since FOLLOW(E') = {),$}. 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)		

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.

Example:

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

FIRST(S) = { i, a} FOLLOW(S) = { $, e }

FIRST(S') = { e, ε} FOLLOW(S') = { $, e }

FIRST(E) = { b } FOLLOW(E) = { t }

So the parsing table for our grammar is:

NONTER-	INPUT SYMBOL					
MINAL	*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:

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

The parsing table for this grammar is repeated with synchronizing tokens

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

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.

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'	$	
$	$	

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".

Example: consider the grammar

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

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

abbcde
aAbcde
aAde
aABe
S

Definition: a *handle* is a substring that

1- matches a right hand side of a production rule in the grammar and
2- Whose reduction to the nonterminal 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.**

An easy to implement form of this parsing, called **operator precedence parsing**.

A much more general method of shift reduce parsing called **LR parsing** , used in a number of automatic parsing generators.

Shift reduces parsing attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (the top).

Shift Reduce Parsing Method

There are two problems that must be solved if we are to parse by handle pruning. The first is to determine the handel, and the second is to determine what production to choose in case there is more than one production with that substring on the right side. A convenient way to implement a shift reduce parser is to use stack to hold grammar symbols and an input buffer to hold the string (W) to be parsed. Use $ to mark the bottom of the stack and also the right end of the input. Initially the stack is empty and the string (W) is on the input, as follows:

Stack	**Input**
$	**W $**

The parser operates by shifting zero or more input symbol onto the stack until a handle β is on top of the stack. The parser then reduces β to the left side of the appropriate production. The parser repeat this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

Stack	**Input**
$ S	**$**

After entering this configuration the parser halts and announces successful completion of parsing.

At each step, the parser performs one of the following actions.

1- Shift one symbol from the input onto the parse stack
2- Reduce one handle on the top of the parse stack. The symbols from the right hand side of a grammar rule are popped of the stack, and the nonterminal symbol is pushed on the stack in their place.
3- Accept is the operation performed when the start symbol is alone on the parse stack and the input is empty.
4- Error actions occur when no successful parse is possible.

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

E → E+E

E → E*E

E → (E)

E → id

	STACK	INPUT	ACTION
(1)	$	$id_1 + id_2 * id_3$$	shift
(2)	$id_1	+ $id_2 * id_3$$	reduce by $E \rightarrow id$
(3)	$E	+ $id_2 * id_3$$	shift
(4)	$E +	$id_2 * id_3$$	shift
(5)	$E + id_2	* $id_3$$	reduce by $E \rightarrow id$
(6)	$E + E	* $id_3$$	shift
(7)	$E + E *	$id_3$$	shift
(8)	$E + E * id_3	$$	reduce by $E \rightarrow id$
(9)	$E + E * E	$$	reduce by $E \rightarrow E * E$
(10)	$E + E	$$	reduce by $E \rightarrow E + E$
(11)	$E	$$	accept

Example 2: parse the input id +* id for the same grammar

Action	Input	Stack
Shift	**id1 + * id2 $**	**$**
Reduce by E → id	**+ * id2 $**	**$ id1**
Shift	*** id2 $**	**$ E+**
Shift	**id2 $**	**$ E+***
Shift	**$**	**$ E +* id**
Reduce by E→ id	**$**	**$ E +*E**
Not Accept	**$**	**$ E +*E**

H.W. : For this grammar

E → E+T | T T

→ T*F | F F →

id | (E)

Parse the input **id * id + id**