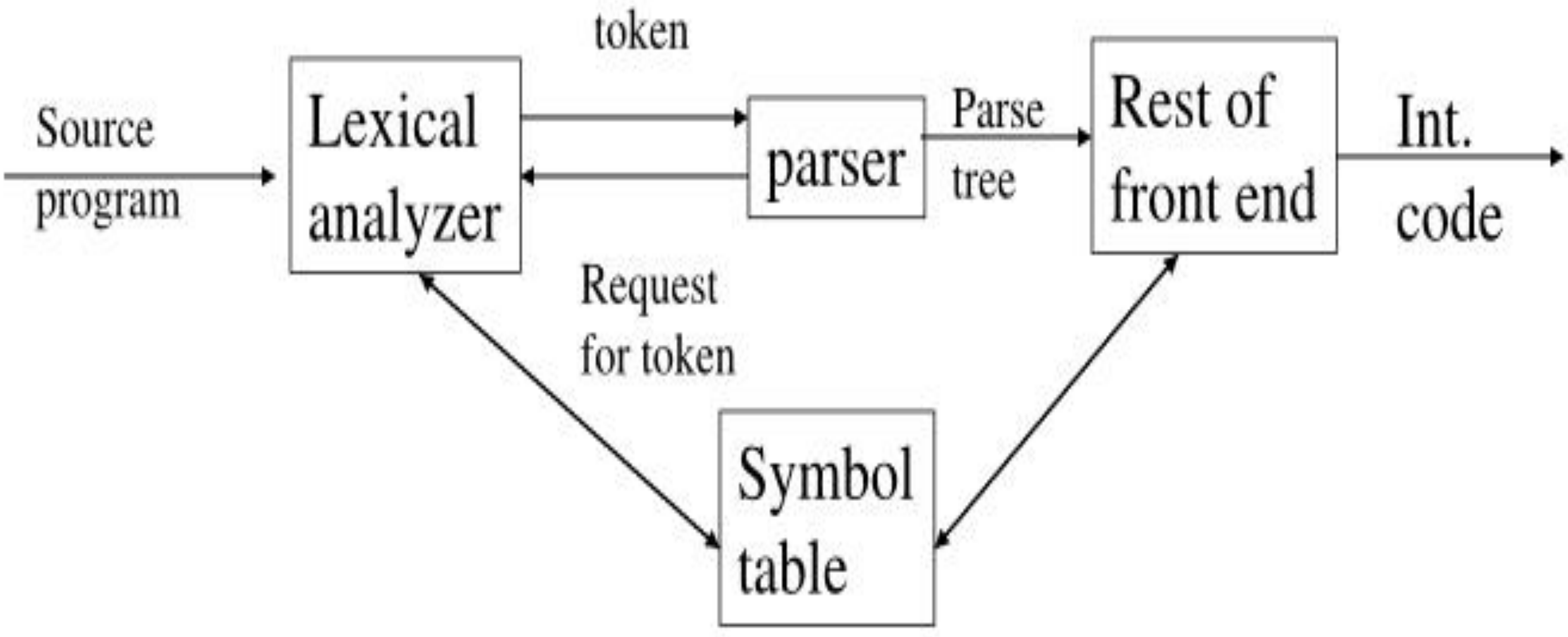
Syntax Analysis

Role of a Parser



Context free grammars

Regular language does not support nested construction

A grammar G = (N, T, P, S) is context free grammar, if each production P is of the form

$A \rightarrow \alpha$,

where $A \in N$ and

$\alpha \in (N \cup T)^*$

N is a finite set of Non-terminals

T is a finite set of Terminals

P is the set of Productions

S is the Start symbol

Example:

exp → exp op exp

exp → (exp)

exp → number

op → +

op → -

op → *

Context free grammars

Regular language does not support nested construction

A grammar G = (N, T, P, S) is context free grammar, if each production P is of the form

$A \rightarrow \alpha$,

where $A \in N$ and

$\alpha \in (N \cup T)^*$

N is a finite set of Non-terminals

T is a finite set of Terminals

P is the set of Productions

S is the Start symbol

Example:

exp → exp op exp | (exp) | number

op → + | - | *

Derivation

Leftmost derivation

exp => exp op exp => exp op exp op exp => number op exp op exp

=> number – exp op exp => number – number op exp

=> number – number * exp => number – number * number

Rightmost derivation

exp => exp op exp => exp op exp op exp => exp op exp op number

=> exp op exp * number ……

Sentence: a string of terminal symbols that can be derived from S

Sentential form: a string of terminals and non-terminals that can be derived from S

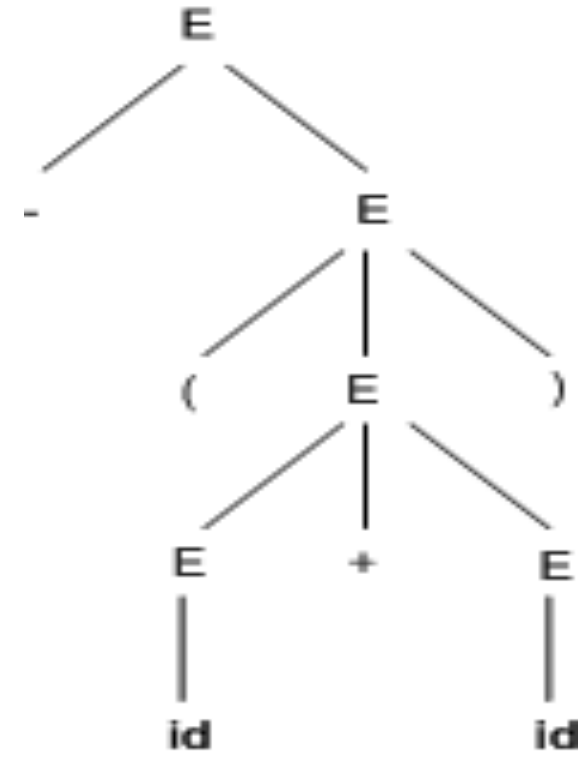
Parse Tree

Grammar :

E -> E + E | E * E | -E | (E) | id

Derivation for –(id+id)

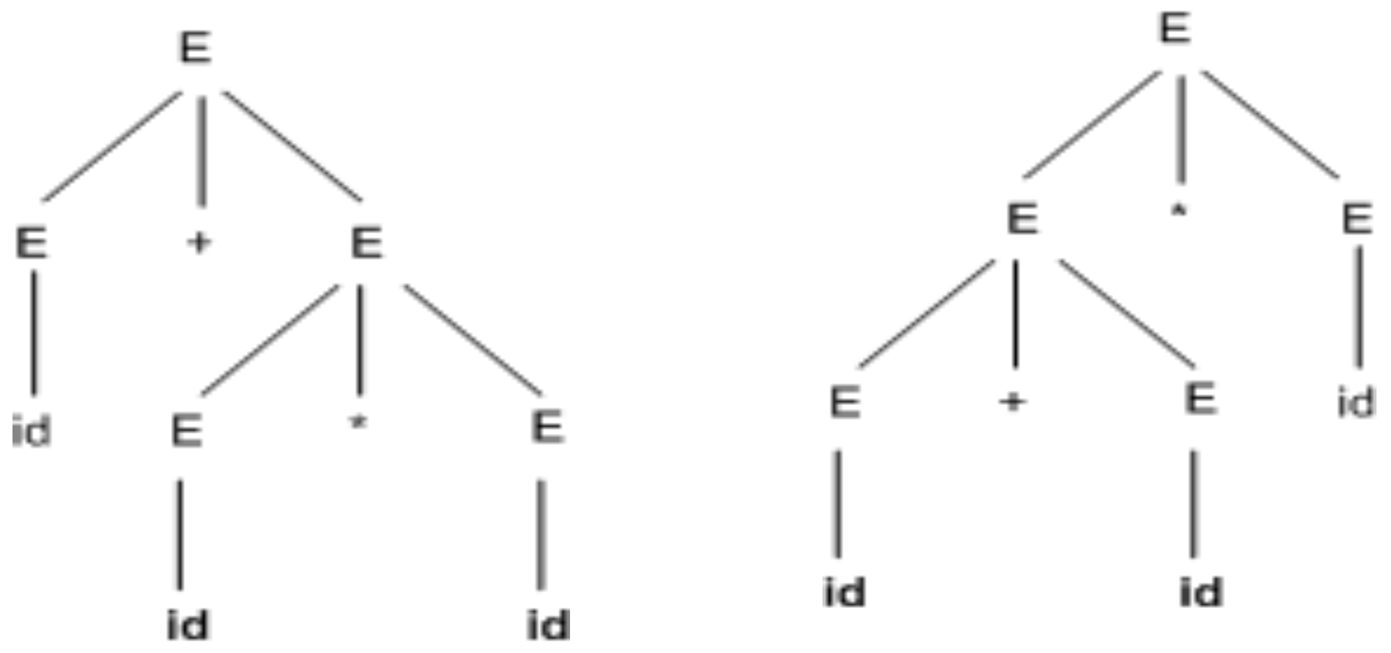
E => -E => -(E) => -(E+E) => -(id+E)=>-(id+id)



Ambiguity

More than one leftmost derivation or more than one rightmost derivation.

For example, for the sentence id + id * id



Elimination of ambiguity

exp → exp op exp | (exp) | number

op → + | - | *

Adding precedence

exp → exp addop exp | term

addop → + | -

term → term mulop term | factor

mulop → *

factor → (exp) | number

With left associativity

exp → exp addop term | term

addop → + | -

term → term mulop factor | factor

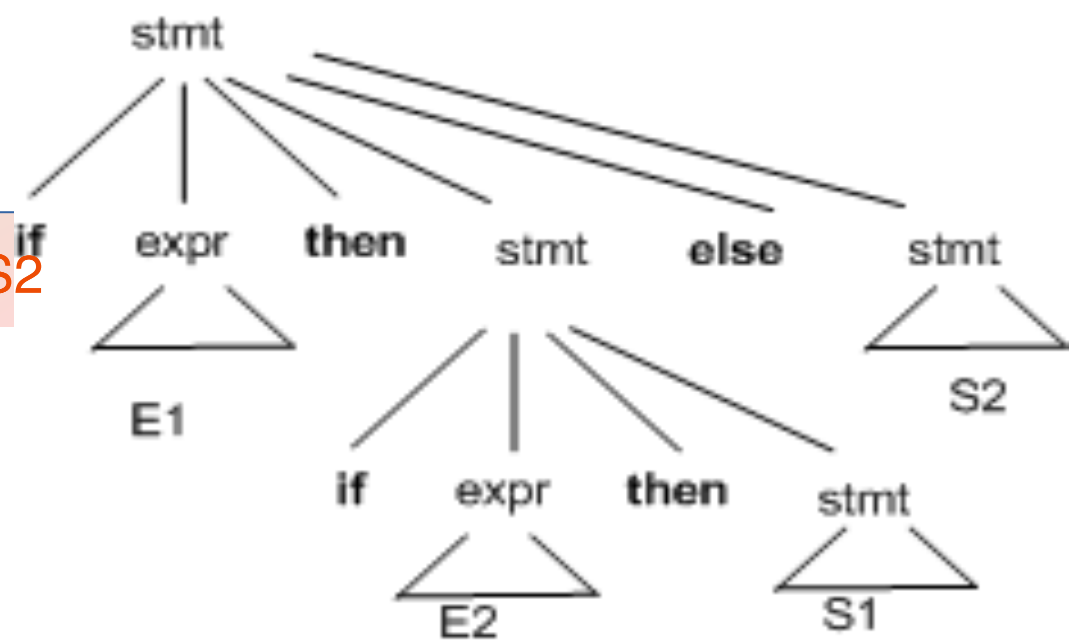
mulop → *

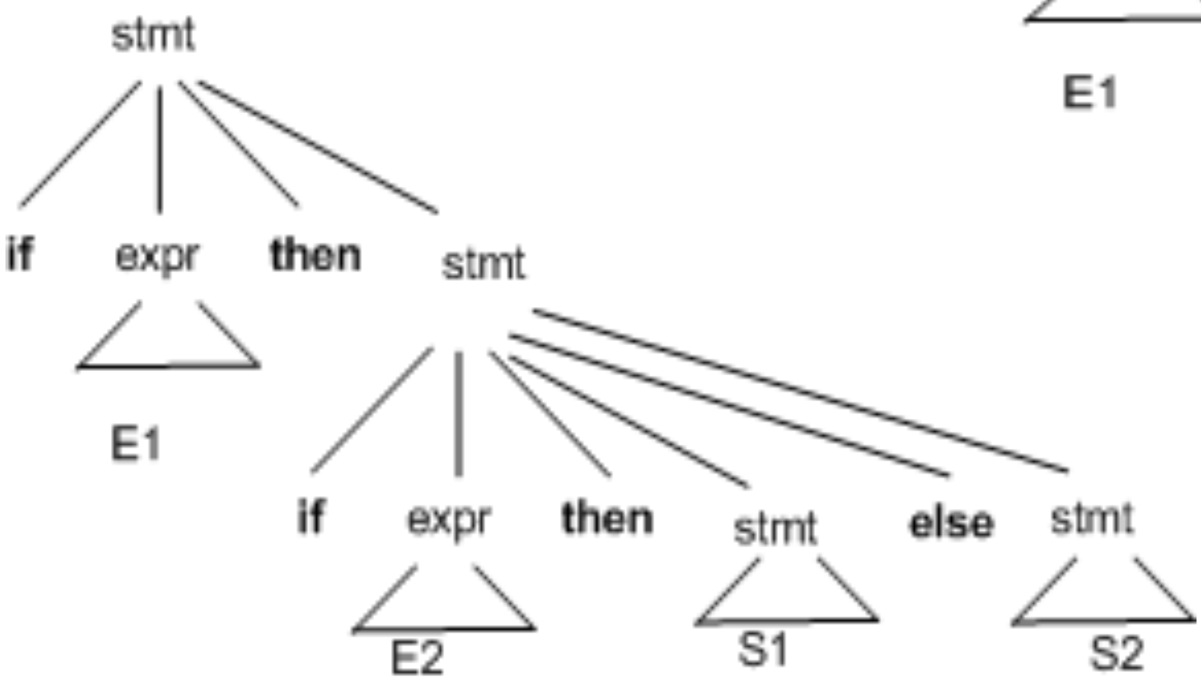
factor → (exp) | number

Dangling Else

stmt ⟶ **If** expr **then** stmt
| **If** expr **then** stmt **else** stmt
| **other**

If E1 then if E2 then S1 else S2



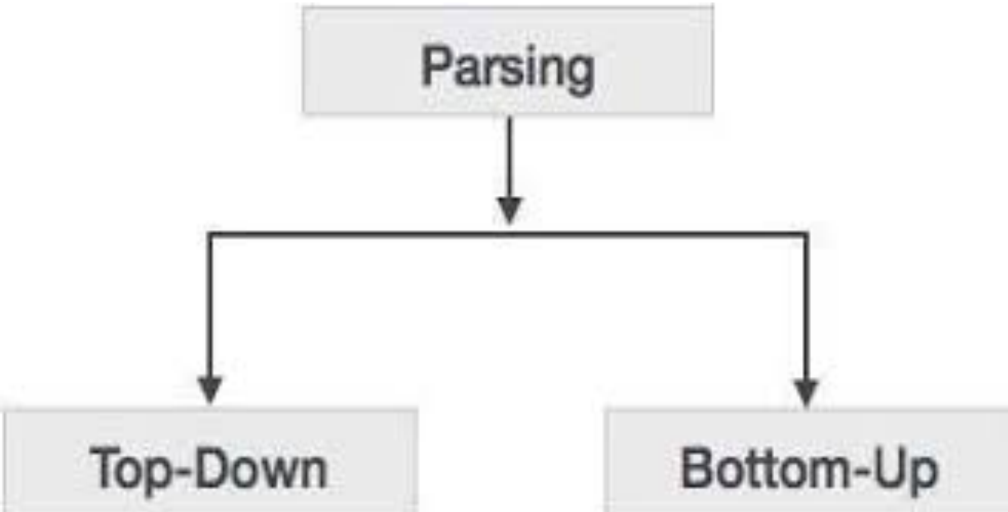


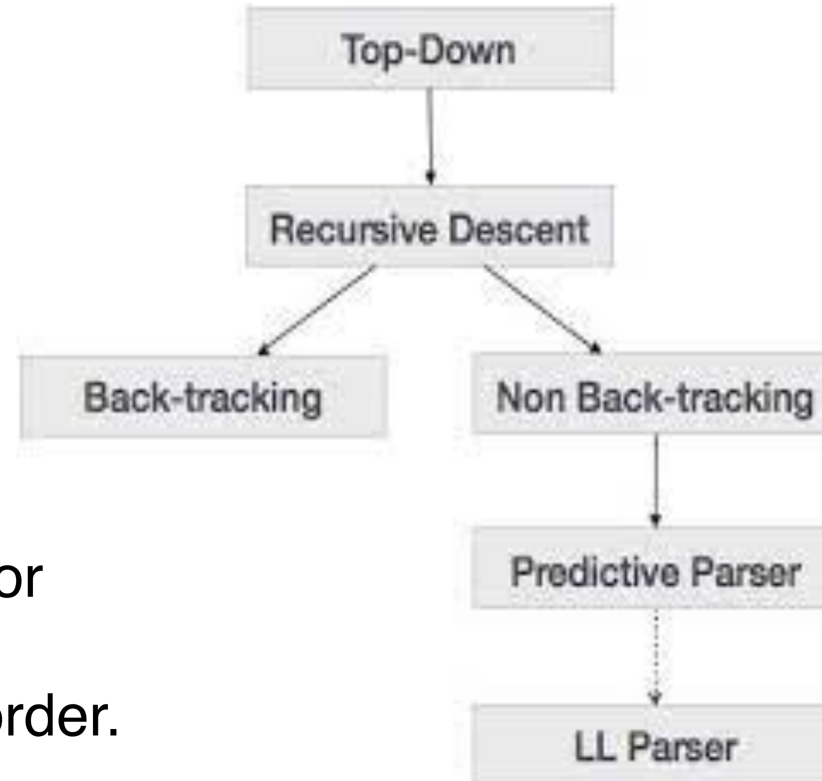
Elimination of ambiguity

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{matched\_stmt} \\
& | & \textit{open\_stmt} \\
\textit{matched\_stmt} & \rightarrow & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{matched\_stmt}\ \textbf{else}\ \textit{matched\_stmt} \\
& | & \textbf{other} \\
\textit{open\_stmt} & \rightarrow & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\
& | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{matched\_stmt}\ \textbf{else}\ \textit{open\_stmt}
\end{array}
$$

Parsing

Parsing is a process of determining if a string of tokes can be generated by a grammar





Top-down parsing constructs a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.

Top-down Parsing

Top-down parsing traces out the steps in a leftmost derivation.

At each step the key problems is

➢ How to choose a production to be applied for a non-terminal *A*

Recursive-descent parsing is a general form of parsing that consists of a set of procedures, one for each non-terminal.

It starts with the start symbol *S*

Recursive-descent parsing may need backtracking

Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Derive id + id * id

Start symbol is E

Which production to be applied first?

Example:

```
Procedure factor;
begin
    case token of
    ( : match ( ( );
        e;
        match ( ) );
    id :  match (id);
    else error;
    end case;
    end factor;
```

```
procedure match (expectedtoken);
begin
    if token = expectedtoken then
        gettoken;
    else
        error;
    endif;
    end match;
```

Predictive parsing

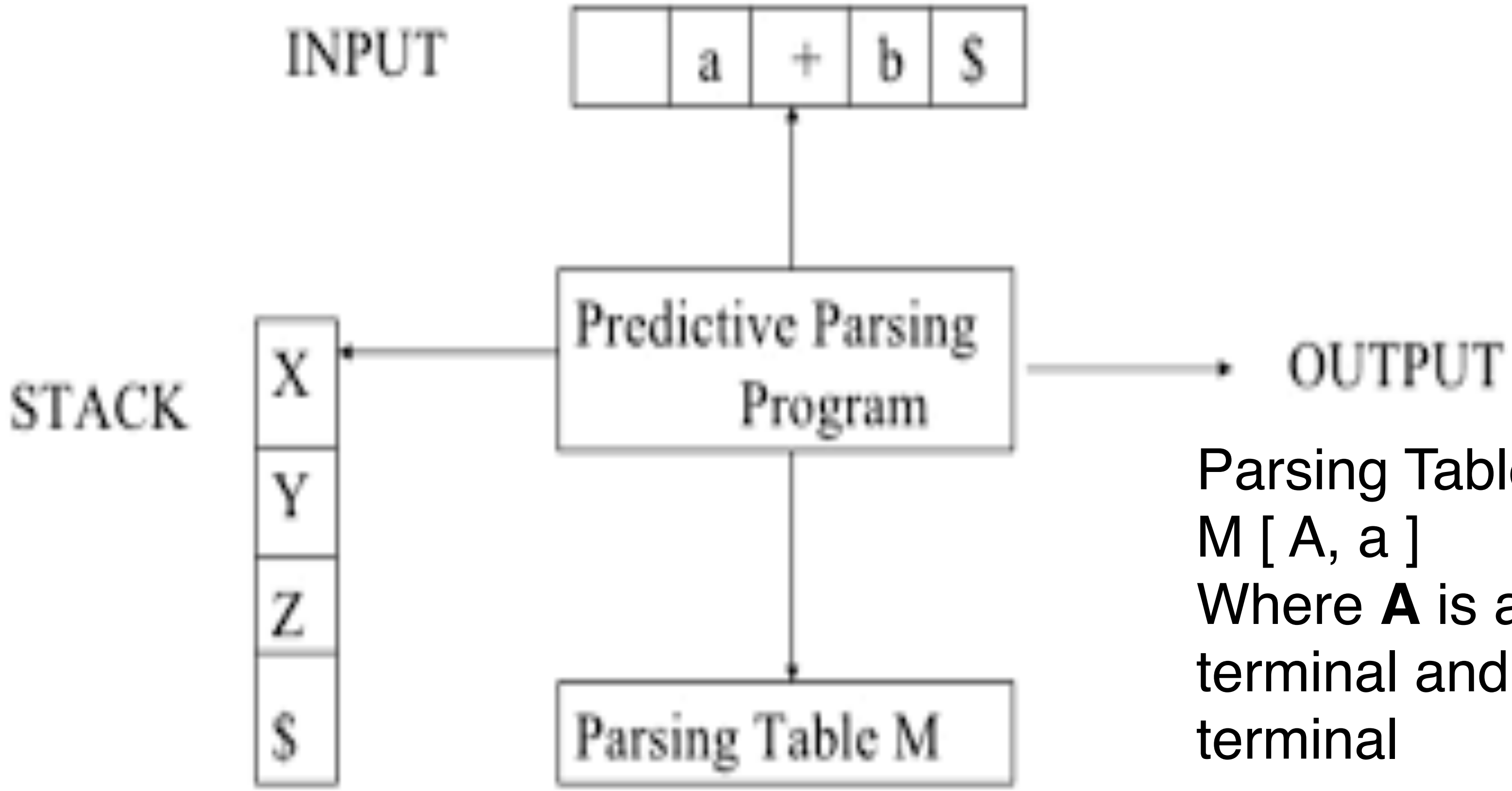
Predictive parsing chooses a correct production by looking ahead at the input a fixed number of symbols

No backtracking is required

Predictive parser can be constructed using a class of grammar known as LL(1)

- ➔ Left to right parsing
- ➔ Leftmost Derivation
- ➔ 1 token lookahead

Predictive parsers



Parsing Table:
M [A, a]
Where **A** is a non-terminal and **a** is a terminal

Left Recursive Grammars

Example:

E → E+T | T

T → T*F | F

F → (E) | id

<u>A general form</u>

A → A α | β ------ immediate left recursion

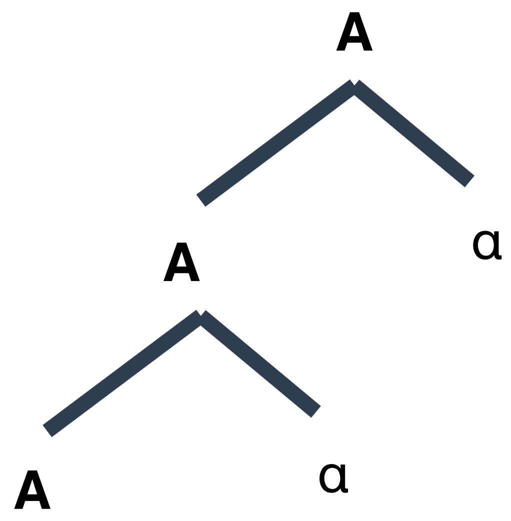
OR

S → A α |b

A → Ac | Sd | ε ------- non-immediate left recursion

Left Recursive Grammars

A left recursive grammar can cause a Top down parser to go into an infinite loop.



Removing Left Recursion

Rule for removing immediate left recursion

$A \rightarrow A\alpha \mid \beta$

To be changed as

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Example:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Removing Left Recursion

Rule for removing immediate left recursion

$A \rightarrow A\alpha \mid \beta$

To be changed as

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Example:

$E \rightarrow T E'$

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

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid \epsilon$

$F \rightarrow (E) \mid id$

Removing non-immediate left recursion

Algorithm:
Arrange the nonterminals in some order A1,A2,…,An

For (each i from 1 to n) {

For (each j from 1 to i-1) {

Replace each production of the form Ai → Aj γ
by the production Ai → δ1 γ | δ2 γ | … |δk γ
where Aj → δ1 | δ2 | … |δk are all current Aj productions

}

Eliminate left recursion among the Ai-productions

}

Removing non-immediate left recursion

Example 1:

S → A ɑ lb

A → Ac l Sd l ε

Example 2:

A → Ba l Aa l c

B → Bb l Ab l d

Left Factoring

- **What happens if we are unable to decide now?**
  - **Delay the decision, wait for further input**

Consider following grammar:

Stmt -> if expr then stmt else stmt

| if expr then stmt

On seeing input 'if' it is not clear for the parser which production to use

We can easily perform left factoring:

If we have $A \rightarrow \alpha\beta 1 \mid \alpha\beta 2$ then we replace it with

$A \rightarrow \alpha A'$

$A' \rightarrow \beta 1 \mid \beta 2$

Left Factoring

Algorithm

For each non-terminal A, find the longest prefix α common to two or more of its alternatives.

If α<> ε, then replace all of A-productions

$A \rightarrow \alpha\beta1 \mid \alpha\beta2 \mid \ldots \mid \alpha\beta n \mid \gamma$ by

$A \rightarrow \alpha A' \mid \gamma$

$A' \rightarrow \beta1 \mid \beta2 \mid \ldots \mid \beta n$

Construction of Parsing Table

Step 1: Compute First and Follow sets

First set

First set for any symbol in the grammar contains the terminal symbols which can be found at the beginning of any string which can be generated from the symbol.

Construction of Parsing Table

Step 1: Compute First and Follow sets

First set

First set for any symbol in the grammar contains the terminal symbols which can be found at the beginning of any string which can be generated from the symbol.

Follow set

Follow(A), for non-terminal *A*, is the set of all terminals *a* that can appear immediately to the right of *A* in some setential form.

Construction of Parsing Table

Computing FIRST(X)

1. If X is a terminal, then FIRST (X) is X
2. If X → ε is a production, then add ε to FIRST(X)
3. If X is non-terminal and X → Y1 Y2 Y3 ….. Yk is a production, then place *a* in FIRST (X) if for some i, *a* is in FIRST (Y_i) and ε is in all of FIRST (Y1), FIRST (Y2) ….. FIRST (Y_{i-1})
4. If ε is in FIRST (Yj) for all j=1,2,...k; then add ε to FIRST (X)

Construction of Parsing Table

Computing FOLLOW (A)

1. Place $ in FOLLOW (S), where S is the start symbol
2. If there is a production A-→ α B β, then everything in FIRST (β) except ε is placed in FOLLOW (B)
3. If there is a production A-→ α B β, and FIRST (β) contains ε, then everything in FOLLOW (A) is in FOLLOW (B)
4. If there is a production A → α B, then everything in FOLLOW (A) is in FOLLOW (B)

Construction of Predictive Parsing Table

1. For each production $A \rightarrow \alpha$ of the grammar, do the following steps -

A) For each terminal *a* in FIRST (α), add $A \rightarrow \alpha$ to M [A, *a*]

B) If ε is in FIRST (α), add $A \rightarrow \alpha$ to M [A, b] for each terminal *b* in FOLLOW (A)

C) If ε is in FIRST (α) and \$ is in FOLLOW (A), add $A \rightarrow \alpha$ to M [A, \$]

2. Make each undefined entry of M "error"

Example

E -> TE'
E' -> +TE' | Ɛ
T -> FT'
T' -> *FT' | Ɛ
F -> (E) | **id**

E $ => TE' $=> FT'E' $=>idT'E' $
=>idE' $
⇒id+TE' $
⇒Id+FT'E' $
⇒Id+id T' E' $
=>id+id E' $ (T'→   e)
=> id+id $ (E'→   e)

Id+id$

Example

E -> TE'
E' -> +TE' | Ɛ
T -> FT'
T' -> *FT' | Ɛ
F -> (E) | **id**

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

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

LL(1) PARSING TABLE

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

Error Recovery in LL(1) Parsers

- A parser should try to determine that an error has occurred as soon as possible.
- After an error has occurred, the parser should pick a likely place to resume the parse. A parser should try to parse as much code as possible.
- A parser should try to avoid the error cascade problem.
- A parser must avoid infinite loops as errors.
- *Panic-mode error recovery - skipping symbols on the input until a token in a selected set of synchronizing tokens.*
- A set of *synchronising tokens* are used for this purpose.

Error Recovery in LL(1) Parsers

- Sets of synchronising tokens are directly built into the parsing table.
- Given the non-terminal A on top of the stack and an input token that is not in FIRST(A) (or in FOLLOW(A) if ε is in FIRST(A)), there are the following three alternatives:
  - Pop A from the stack [if current input token is $ or is in FOLLOW(A)] – **pop**
  - Successively pop tokens from the input until a token is seen for which we restart the parse [if the current token is not $ and is not in FIRST(A) U FOLLOW(A)] – **scan**
  - Push a new non-terminal (usually start symbol) onto the stack and scan forward until a symbol in the FIRST set of start is found

Error Recovery in LL(1) Parsers

	Id	+	*	(	)	$
E	E→TE′	scan	scan	E→TE′	pop	pop
E′	scan	E′→+TE′	scan	scan	E′→ë	E′→ë
T	T→FT′	pop	scan	T→FT′	pop	pop
T′	scan	T′→ë	T′→*FT′	scan	T′→ë	T′→ë
F	F→id	pop	pop	F→(E)	pop	pop

Home Work (to be submitted by midnight of 17th March)

Construct the LL(1) parsing table for the given grammars.

Show the parsing of the string *aabbc* using the left-side grammar.

What abnormality do you find in the right-side grammar?

S --> ABC

A --> aA | ε

B --> bB | ε

C --> c

(Terminals = {a, b, c}, Non-terminals = {S, A, B}, Start Symbol = S)

$Z \rightarrow d$
$Z \rightarrow XYZ$
$Y \rightarrow \varepsilon$
$Y \rightarrow c$
$X \rightarrow Y$
$X \rightarrow a$

(Terminals = {a, c, d}, Non-terminals ={Z, X, Y}, Start Symbol = S