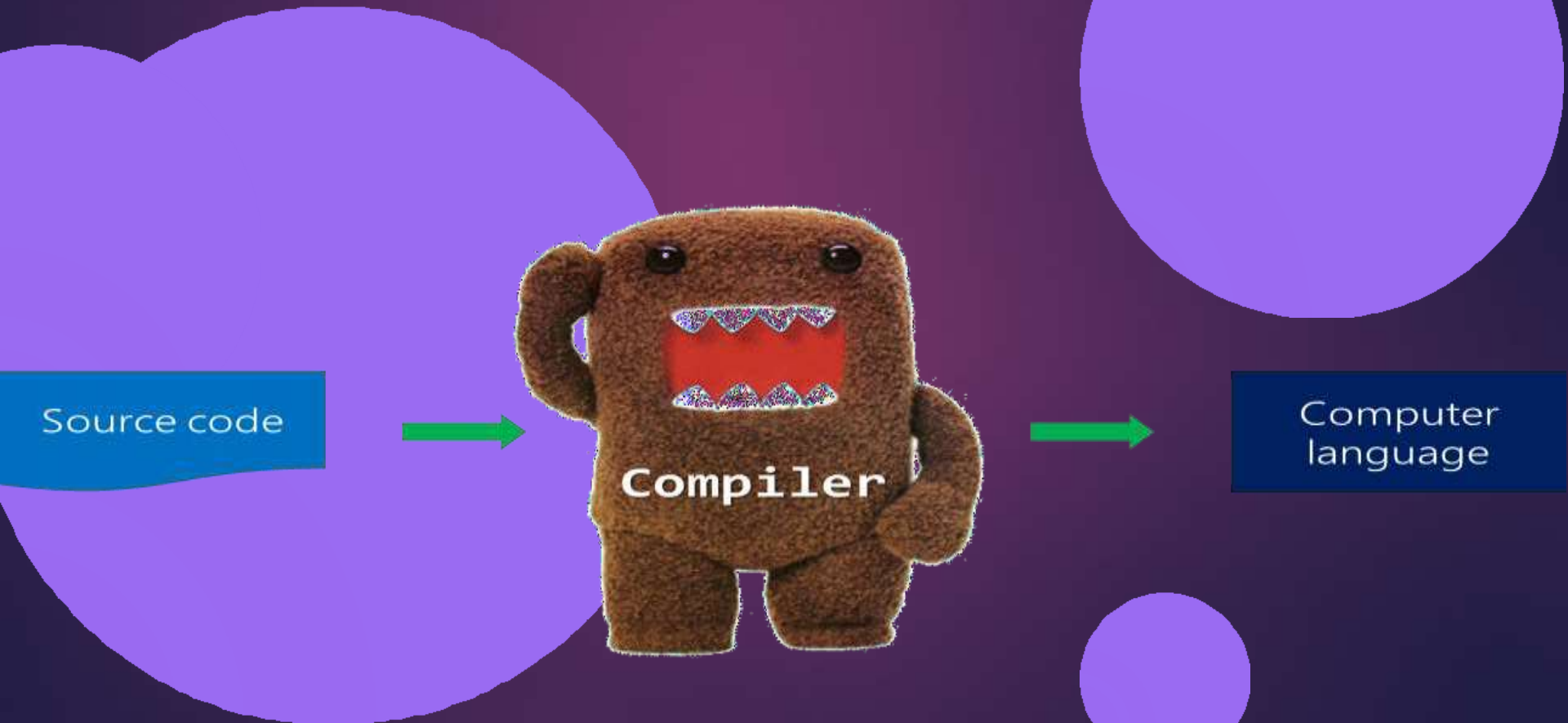
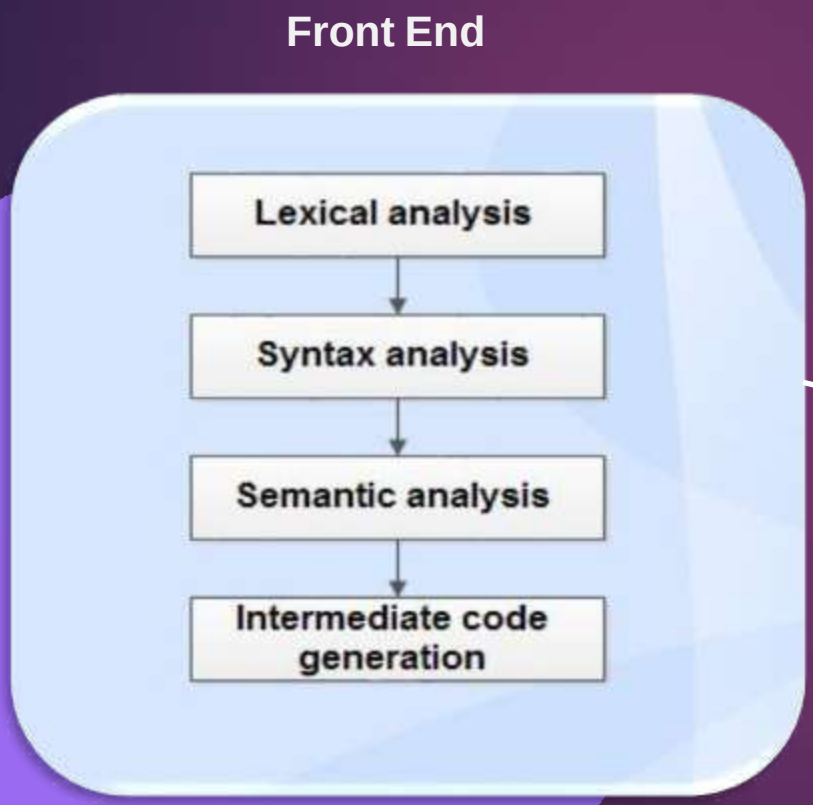
LL(1) PARSER



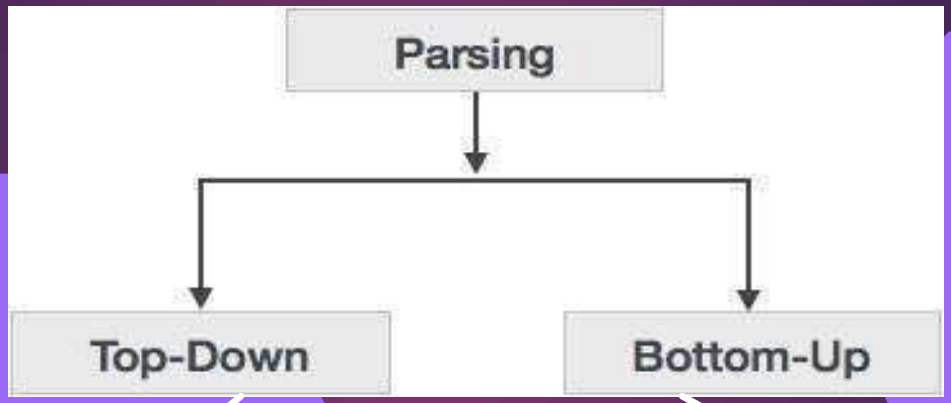
MODEL OF COMPILER FRONT END



Syntax analysis

Also called parsing , where generates parse tree

PARSING



When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

Where bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

1.The Role of the Parser • We categorize the parsers into two groups:

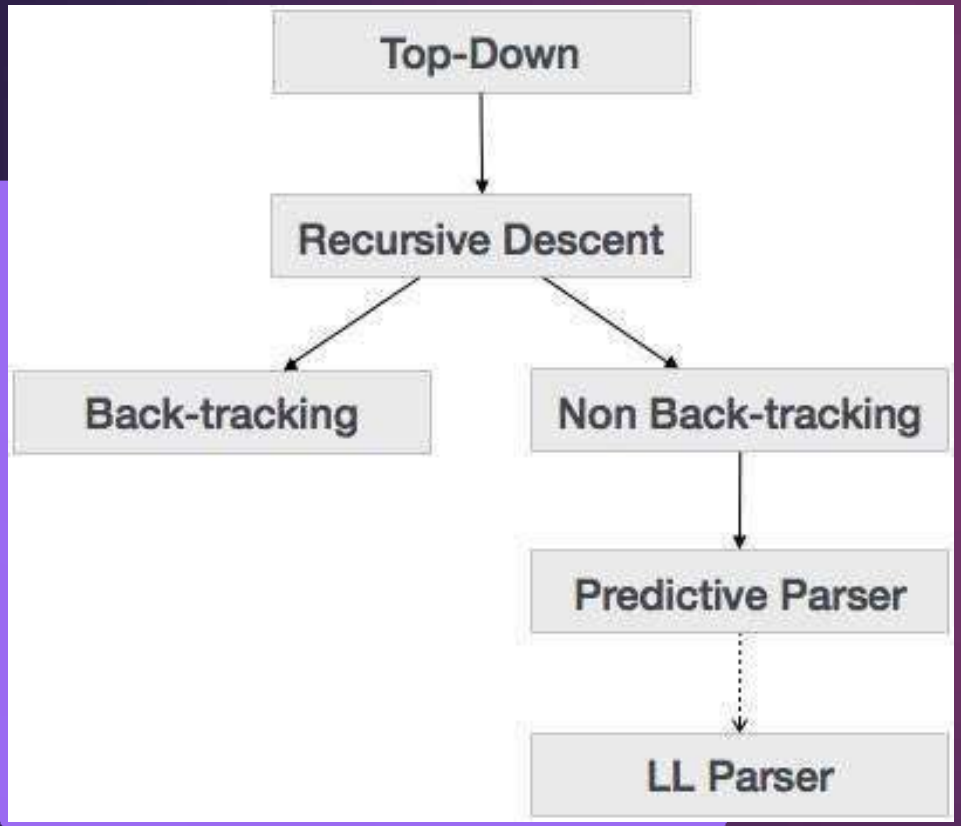
2. Top-Down Parser – the parse tree is created from top to bottom, starting from the root.

3. Bottom-Up Parser – the parse tree is created from bottom to top, starting from the leaves •

Both top-down and bottom-up parsers scan the input from left to right and one symbol at a time.

• Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars. –
LL for top-down parsing –
LR for bottom-up parsing

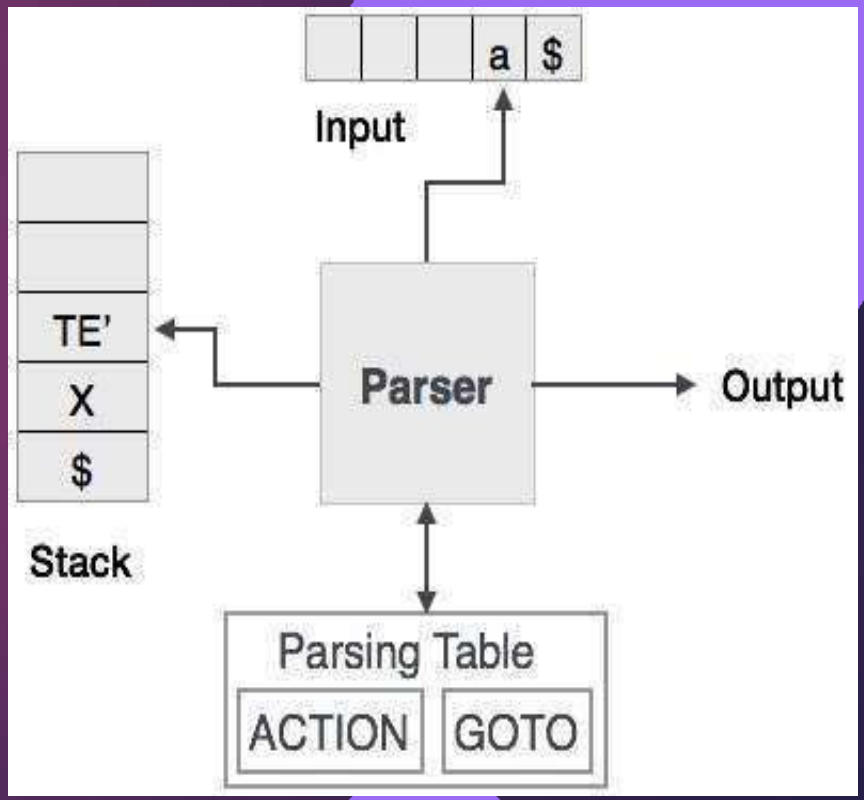
TOP DOWN PERSER



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.

PREDICTIVE PARSER

- Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree.
- Both the stack and the input contains an end symbol **$**to denote that the stack is empty and the input is consumed.
- The parser refers to the parsing table to take any decision on the input and stack element combination.



Predictive Parsing

Recursive descent parsing is a top-down parsing method

- Each nonterminal has one (recursive) procedure tha tis responsible for parsing the nonterminal's syntactic category of input tokens
- When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information

Predictive parsing is a special form of recursive descent parsing where we use one lookahead token to unambiguously determine the parse operations

Top-Down Parsing

- The parse tree is created from top to bottom.
- **Top-down parser**
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
    - Non-Recursive Predictive Parser is also known as LL(1) parser.

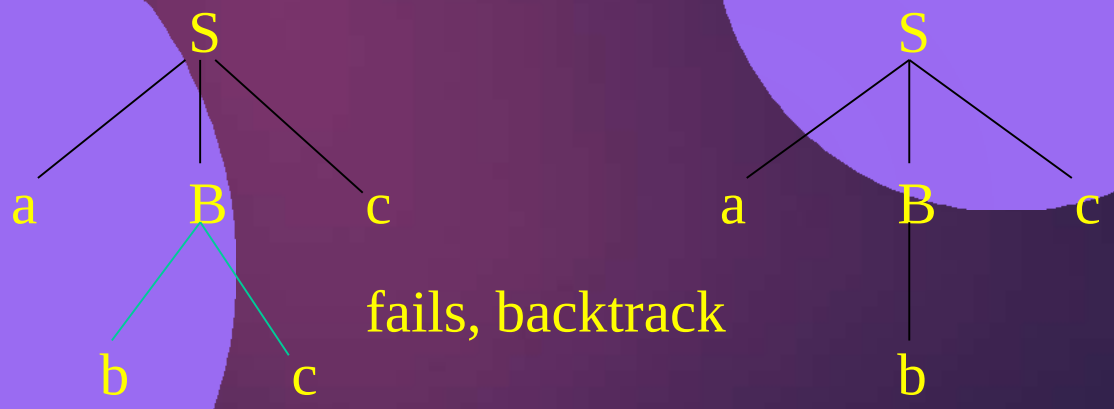
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

input: abc



Predictive Parser

a grammar → (eliminate left recursion) → (left factor) → a grammar suitable for predictive parsing (a LL(1) grammar)

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \rightarrow \alpha_1 \mid ... \mid \alpha_n$ input: ... a .....

current token

Predictive Parser (example)

stmt → `if` ..... |
`while` ..... |
`begin` ..... |
`for` .....

- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule.
- We eliminate the left recursion in the grammar, and left factor it.

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

```
proc A {
    - match the current token with a, and move to the next token;
    - call 'B';
    - match the current token with b, and move to the next token;
}
```

Recursive Predictive Parsing

$A \rightarrow aBb \mid bAB$

```
proc A {
    case of the current token {
        'a':  - match the current token with a, and move to the next token;
              - call 'B';
              - match the current token with b, and move to the next token;
        'b':  - match the current token with b, and move to the next token;
              - call 'A';
              - call 'B';
    }
}
```

Recursive Predictive Parsing

- When to apply ε-productions.

  $A \rightarrow aA \mid bB \mid \varepsilon$

- If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.
- Most correct choice: We should apply an ε-production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

```
proc A {
    case of the current token {
        a:  - match the current token with a,
              and move to the next token;
            - call B;
            - match the current token with e,
              and move to the next token;
        c:  - match the current token with c,
              and move to the next token;
            - call B;
            - match the current token with d,
              and move to the next token;
        f:  - call C
    }
}
```

first set of C

```
proc C {
        match the current token
        with f,  and move to the
        next token; }
```

```
proc B {
case of the current token {
        b: - match the current token
        with b,  and move to the next
                        token;
            - call B
    e,d: do nothing
  }
}
```

follow set of B

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.
- In LL(1) the first "L " → scanning the input from left to right and
  second "L" → producing a leftmost derivation and
  the "1" → one input symbol of lookahead at each step
- It uses stack explicity
- In non recursive predictive parser ,production is applied on the parsing table

Example Predictive Parser (Grammar)

$$
\begin{aligned}
\textit{type} &\rightarrow \textit{simple} \\
&\mid \textbf{\^{} id} \\
&\mid \textbf{array [} \textit{simple} \textbf{] of } \textit{type} \\
\textit{simple} &\rightarrow \textbf{integer} \\
&\mid \textbf{char} \\
&\mid \textbf{num dotdot num}
\end{aligned}
$$

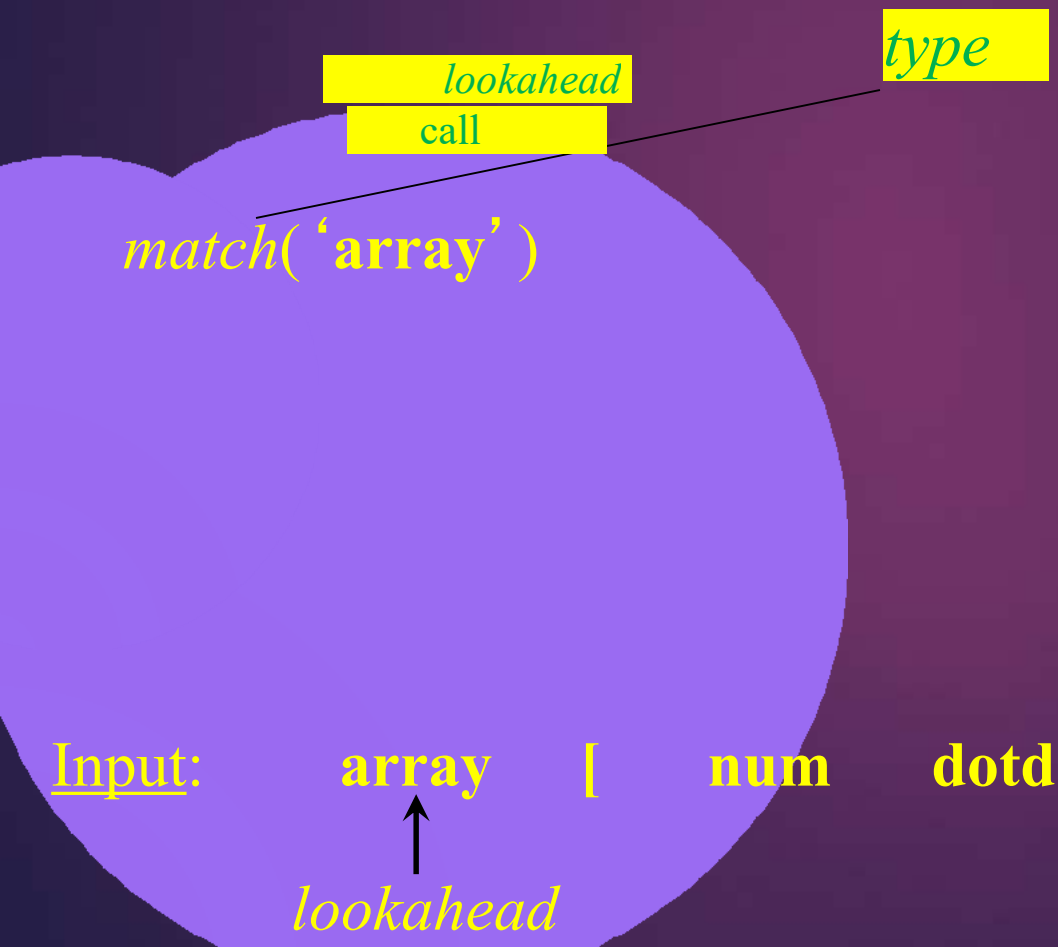
Example Predictive Parser (Program Code)

```
procedure match(t : token);
begin
   if lookahead = t then
      lookahead := nexttoken()
   else error()
end;
```

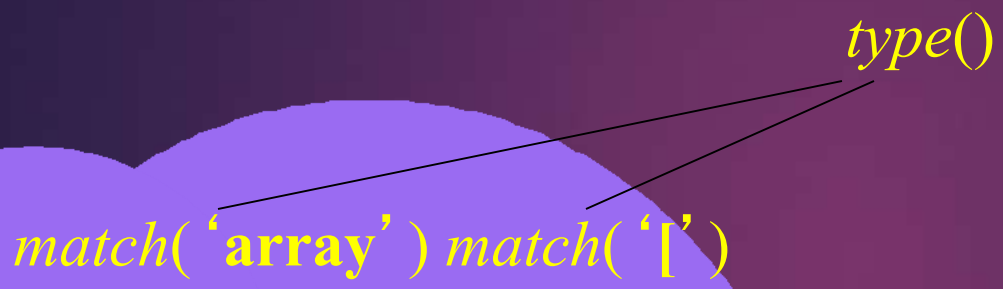
```
procedure type();
begin
   if lookahead in { 'integer', 'char', 'num' } then
      simple()
   else if lookahead = '^' then
      match( '^' ); match(id)
   else if lookahead = 'array' then
      match( 'array' ); match( '[' ); simple();
      match( ']' ); match( 'of' ); type()
   else error()
end;
```

```
procedure simple();
begin
   if lookahead = 'integer' then
      match( 'integer' )
   else if lookahead = 'char' then
      match( 'char' )
   else if lookahead = 'num' then
      match( 'num' );
      match( 'dotdot' );
      match( 'num' )
   else error()
end;
```

Example Predictive Parser (Execution Step 1)



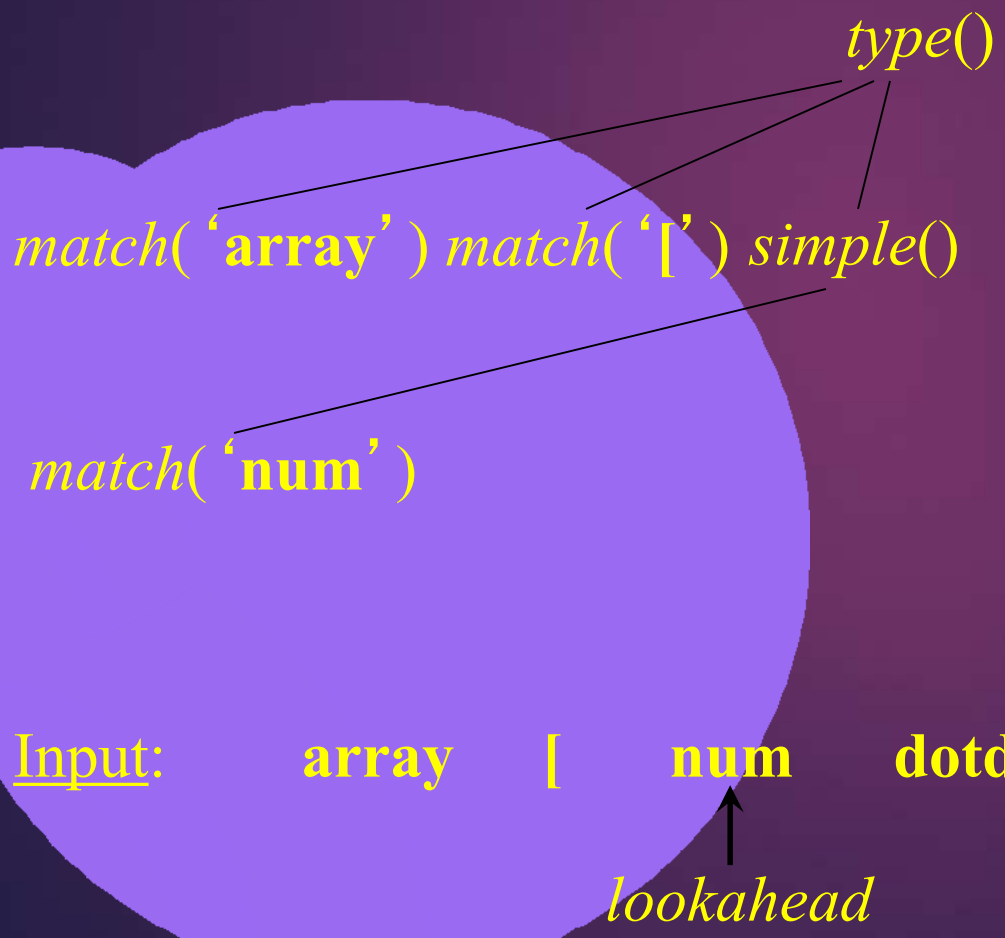
Example Predictive Parser (Execution Step 2)



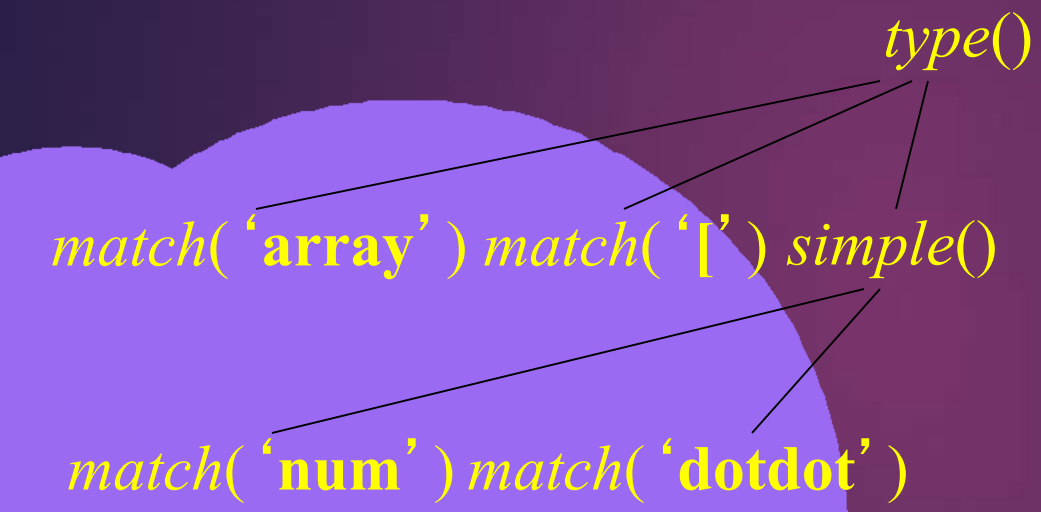
Input: **array [num dotdot num] of integer**

lookahead

Example Predictive Parser (Execution Step 3)



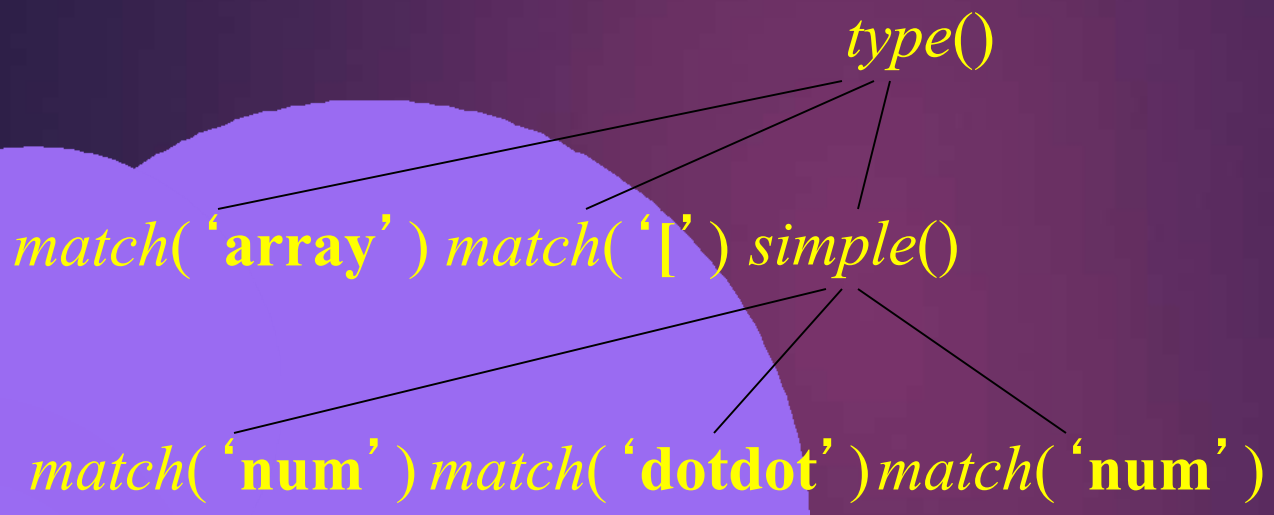
Example Predictive Parser (Execution Step 4)



Input: **array [num dotdot num] of integer**

lookahead

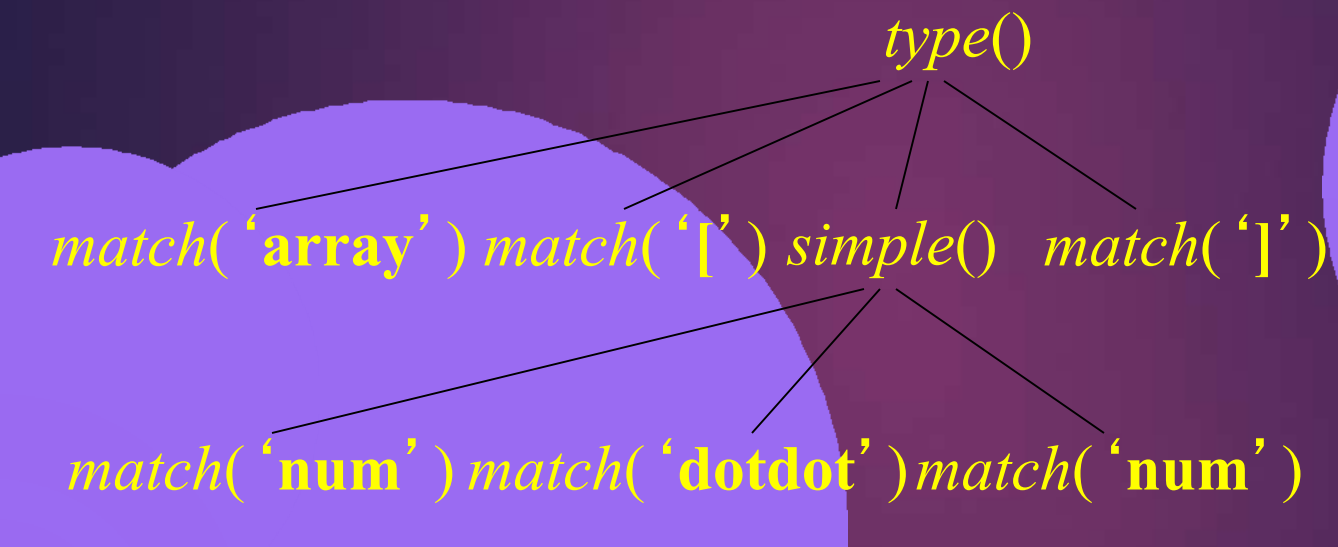
Example Predictive Parser (Execution Step 5)



Input: **array [num dotdot num] of integer**

lookahead

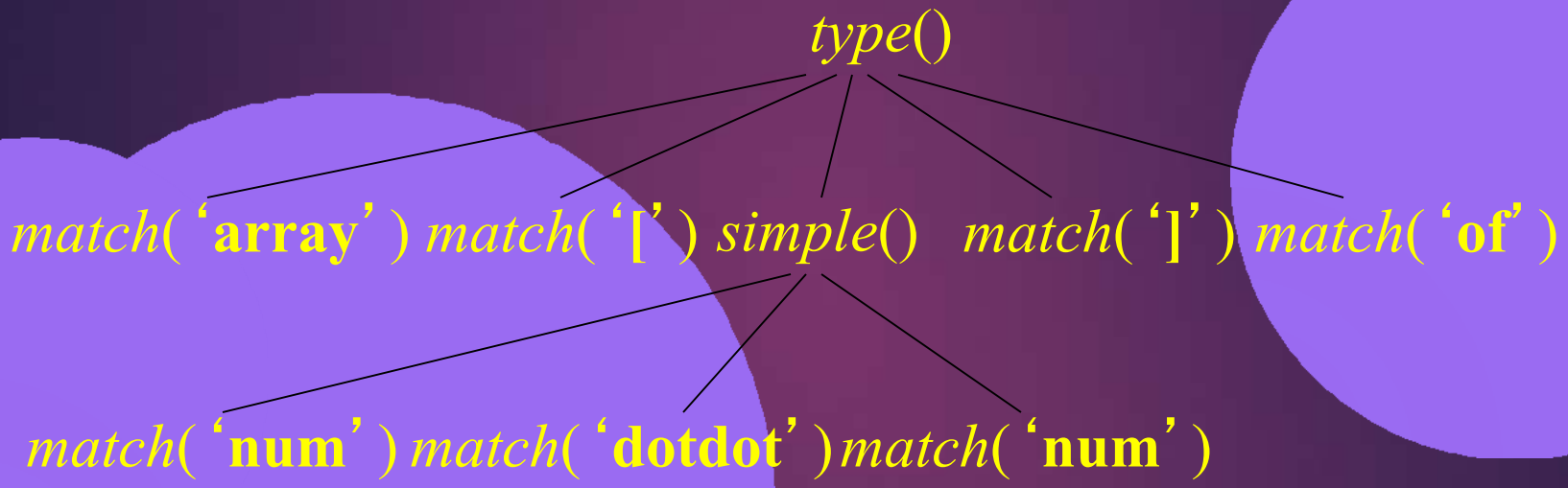
Example Predictive Parser (Execution Step 6)



Input: **array [num dotdot num] of integer**

lookahead

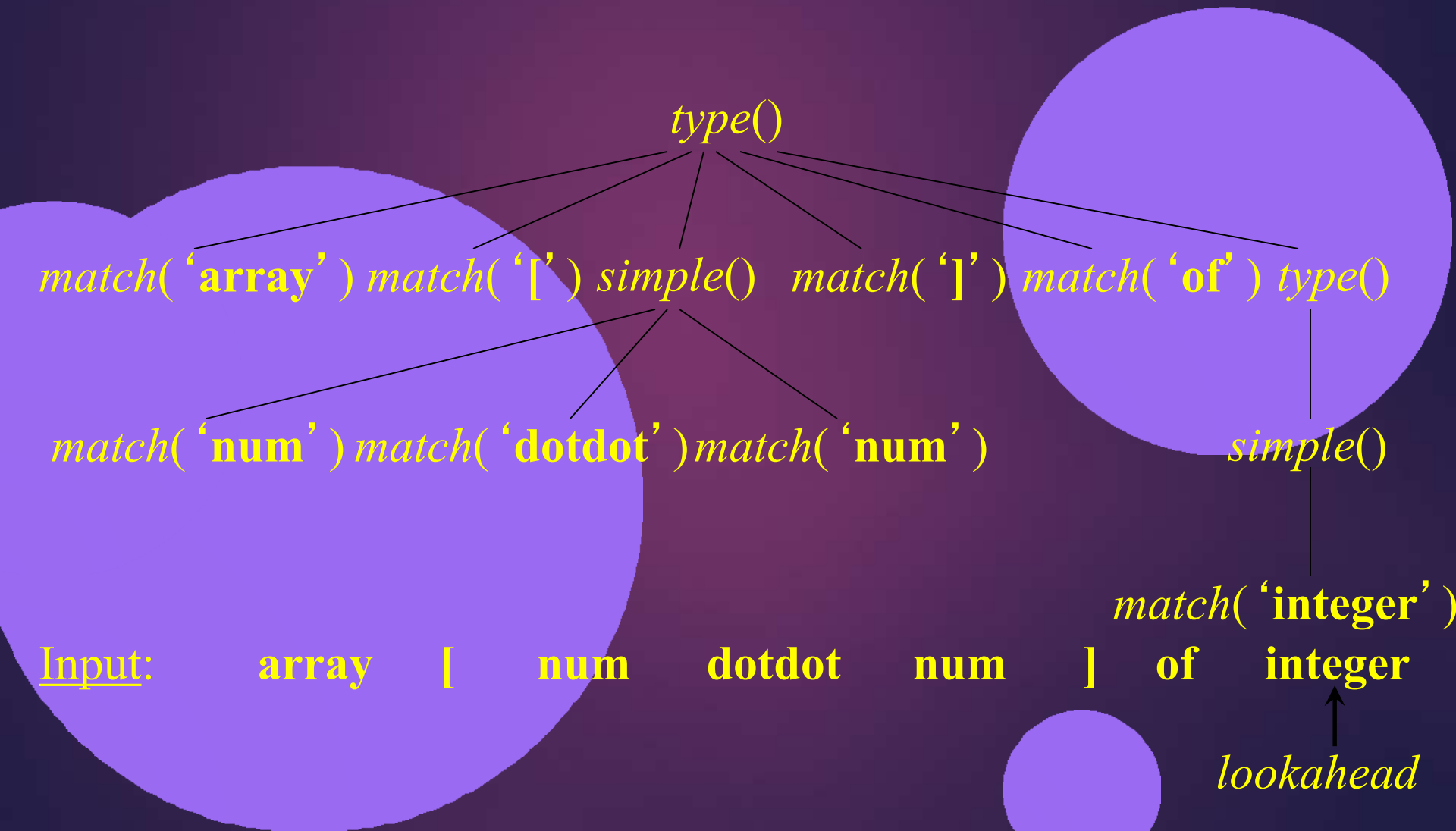
Example Predictive Parser (Execution Step 7)



Input: **array [num dotdot num] of integer**

lookahead

Example Predictive Parser (Execution Step 8)



$expr \rightarrow term\ rest$

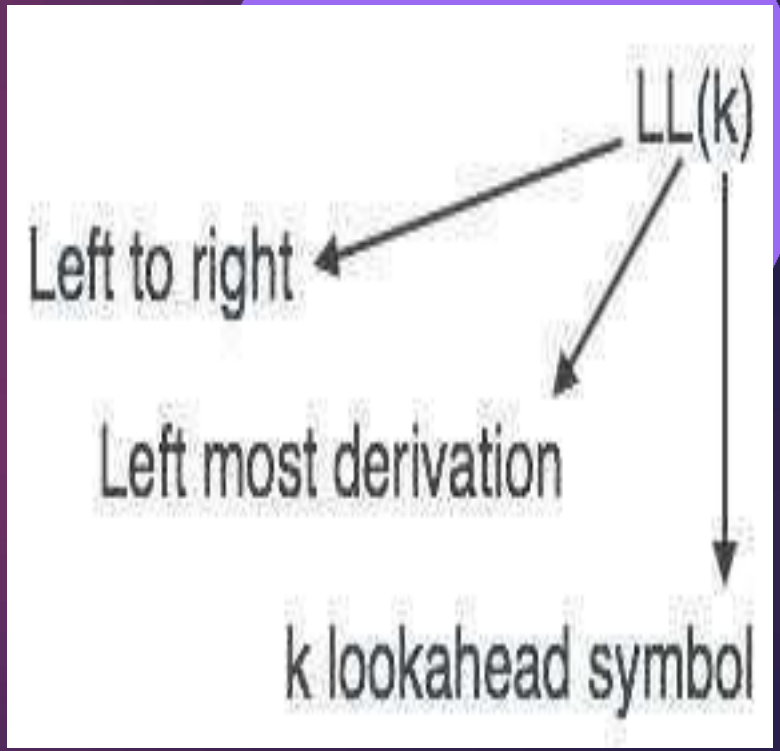
$rest \rightarrow + term\ \{ \text{print}(\text{“+”}) \}\ rest$
$\quad | - term\ \{ \text{print}(\text{“-”}) \}\ rest$
$\quad | \varepsilon$

$term \rightarrow \mathbf{0}\ \{ \text{print}(\text{“0”}) \}$
$term \rightarrow \mathbf{1}\ \{ \text{print}(\text{“1”}) \}$
$\ldots$
$term \rightarrow \mathbf{9}\ \{ \text{print}(\text{“9”}) \}$

```
main()
{   lookahead = getchar();
    expr();
}
expr()
{   term();
    while (1) /* optimized by inlining rest()
                 and removing recursive calls */
    {   if (lookahead == '+')
        {   match('+'); term(); putchar('+');
        }
        else if (lookahead == '-')
        {   match('-'); term(); putchar('-');
        }
        else break;
    }
}
term()
{   if (isdigit(lookahead))
    {   putchar(lookahead); match(lookahead);
    }
    else error();
}
match(int t)
{   if (lookahead == t)
        lookahead = getchar();
    else error();
}
error()
{   printf("Syntax error\n");
    exit(1);
}
```

LL(1) PARSER

- An LL parser is called an LL(*k*) parser if it uses *k* tokens of look ahead when parsing a sentence.
- LL grammars, particularly LL(1) grammars, as parsers are easy to construct, and many computer languages are designed to be LL(1) for this reason.
- The 1 stands for using **one** input symbol of look ahead at each step to make parsing action decision.



CONTINUE…

LL(*k*) parsers must predict which production replace a non-terminal with as soon as they see the non-terminal. The basic LL algorithm starts with a stack containing [*S*, $] (top to bottom) and does whichever of the following is applicable until done:

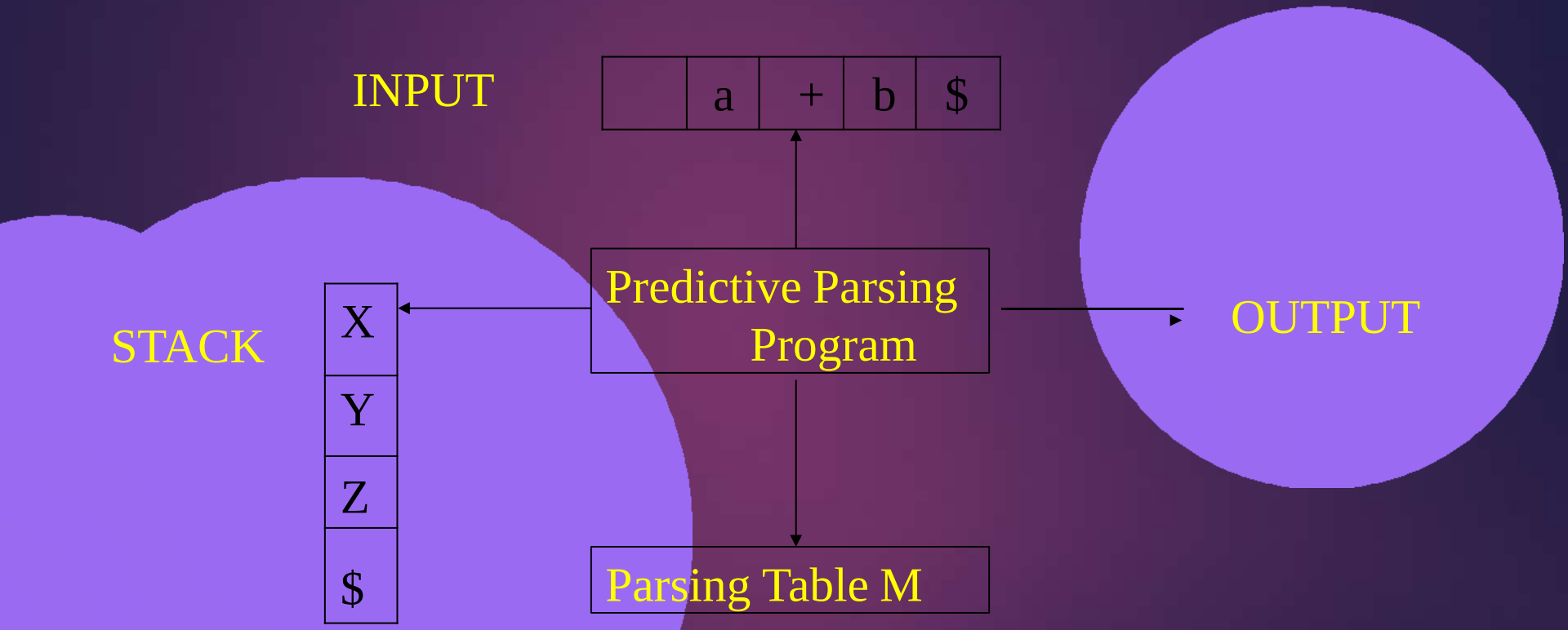
- If the top of the stack is a non-terminal, replace the top of the stack with one of the productions for that non-terminal, using the next *k* input symbols to decide which one (without moving the input cursor), and continue.
- If the top of the stack is a terminal, read the next input token. If it is the same terminal, pop the stack and continue. Otherwise, the parse has failed and the algorithm finishes.
- If the stack is empty, the parse has succeeded and the algorithm finishes. (We assume that there is a unique EOF-marker $ at the end of the input.)

So look ahead meaning is - **looking at input tokens without moving the input cursor**.

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.
- In LL(1) the first "L" → scanning the input from left to right and
  second "L" → producing a leftmost derivation and
  the "1" → one input symbol of lookahead at each step
- It uses stack explicity
- In non recursive predictive parser ,production is applied on the parsing table

Non-Recursive Predictive Parsing



LL(1) Parser

Input buffer

- Input string to be parsed .The end of the string is marked with a special symbol $.

Output

- A production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

Stack

- Contains the grammar symbols
- At the bottom of the stack, there is a special end marker symbol $.
- Initially the stack contains only the symbol $ and the starting symbol S. ie, $S ← initial stack
- When the stack is emptied (ie. only $ left in the stack), the parsing is completed.

Parsing table

- A two-dimensional array M[A,a]
- Each row (A) ,is a non-terminal symbol
- Each column (a), is a terminal symbol or the special symbol $
- Each entry holds a production rule.

LL(1) Parser – 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**:-
- If X = a = $ ➔ parser halts and announces successful completion of the parsing
- If X = a # $ ➔ parser pops X from the stack, and advances the input pointer to the next input symbol

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

11. 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.

Non Recursive Predictive Parsing program

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 parser is in configuration ,it has $S on the stack with S , the start symbol of G on top ,and w$ in the input buffer. The program that utilizes the parsing table M to produce a parse for the input

Algorithm:

Algorithm:

set ip to point to the first symbol of w$;

repeat

let X be the top of the stack and a the symbol pointed by ip;

if X is a terminal or $ then

if X=a then

pop X from the stack and advance ip

else error( )

else

if M [X ,a] = $X \longrightarrow Y_1 Y_2 \ldots Y_K$ then begin

pop X from the stack;

push $Y_K \ldots \ldots Y_2 Y_1$ on to the stack ,with Y_1 on top;

output the production $X \longrightarrow Y_1 Y_2 \ldots Y_K$

end

else error( )

until X= $

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.
- In LL(1) the first "L " → scanning the input from left to right and
  second "L" → producing a leftmost derivation and
  the "1" → one input symbol of lookahead at each step
- It uses stack explicity
- In non recursive predictive parser ,production is applied on the parsing table

PRIME REQUIREMENT OF LL(1)

- The grammar must be -
  - ✓ no left factoring
  - ✓ no left recursion

- FIRST() & FOLLOW()
- Parsing Table
- Stack Implementation
- Parse Tree

STEP: LEFT FACTORING

LEFT FACTORING

- A grammar is said to be left factored when it is of the form –

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots\ldots \mid \alpha\beta_n \mid \gamma$

- The productions start with the same terminal (or set of terminals).
- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen to make the right choice.

For the grammar

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots\ldots \mid \alpha\beta_n \mid \gamma$

The equivalent left factored grammar will be –

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

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots \mid \beta_n$

CONTINUE…

- For example :

the input string is - aab & grammar is

S ->aAb|aA|ab

A ->bAc|ab

After removing left factoring -

S ->aA’

A’ ->Ab|A|b

A ->ab|bAc

STEP: LEFT RECURSION

RECURSION

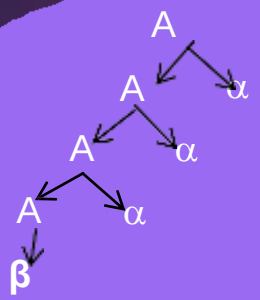
RECURSION:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

Left Recursion

For grammar:

A -> A α| β

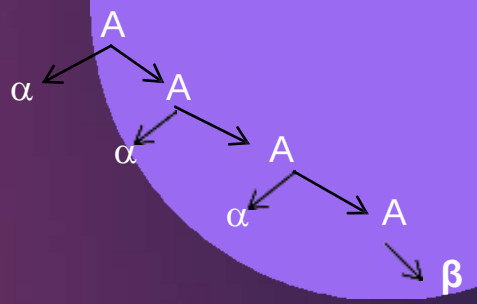


This parse tree generate **β** α *

Right Recursion

For grammar:

A -> α A| β



This parse tree generate α * **β**

Right recursion-

- A production of grammar is said to have right recursion if the right most variable RHS is same as variable of its LHS. e.g. **A -> α A| β**
- A grammar containing a production having right recursion is called as a right recursive grammar.
- Right recursion does not create any problem for the top down parsers.
- Therefore, there is no need of eliminating right recursion from the grammar.

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. e.g. **A -> A α|β**
- A grammar containing a production having left recursion is called as a left recursive grammar.
- Left recursion is eliminated because top down parsing method can not handle left recursive grammar.

Left Recursion

A grammar is left recursive if it has a nonterminal A such that there is a derivation

A -> A α**| β** for some string α**.**

Immediate/direct left recursion:

A production is immediately left recursive if its left hand side and the head of its right hand side are the same symbol, e.g. **A ->A** α **,** where **α** is α sequence of non terminals and terminals.

Indirect left recursion:

Indirect left recursion occurs when the definition of left recursion is satisfied via several substitutions. It entails a set of rules following the pattern

.

$$A \rightarrow Br$$

$$B \rightarrow Cs$$

$$C \rightarrow At$$

Here, starting with a, we can derive **A ->** Atsr

Elimination of Left-Recursion

- Suppose the grammar were

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

How could the parser decide how many times to use the production $A \rightarrow A\alpha$ before using the production A --> β ?

- Left recursion in a production may be removed by transforming the grammar in the following way.

Replace

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

With

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

EXAMPLE OF IMMEDIATE LEFT RECURSION

Consider the left recursive grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

Apply the transformation to E:

$E \rightarrow T\ E'$

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

Then apply the transformation to T:

$T \rightarrow F\ T'$

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

Now the grammar is

$E \rightarrow T\ E'$

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

$T \rightarrow F\ T'$

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

$F \rightarrow (E) \mid \mathbf{id}$

Continue…

The case of several immediate left recursive α -productions. Assume that the set of all α -productions has the form

$A \rightarrow A\,\alpha 1 \mid A\,\alpha 2 \mid \cdots \mid A\,\alpha m \mid \beta 1 \mid \beta 2 \mid \cdots \mid \beta n$

Represents all the α -productions of the grammar, and no βi begins with A, then we can replace these α -productions by

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

$A' \rightarrow \alpha 1 A' \mid \alpha 2 A' \mid \cdots \mid \alpha m A' \mid \varepsilon$

Example:

Consider the left recursive grammar

$$S \rightarrow SX \mid SSb \mid XS \mid a$$

$$X \rightarrow Xb \mid Sa$$

Apply the transformation to *S*:

$$S \rightarrow XSS' \mid aS'$$

$$S' \rightarrow XS' \mid SbS' \mid \varepsilon$$

Apply the transformation to *X*:

$$X \rightarrow SaX'$$

$$X' \rightarrow bX' \mid \varepsilon$$

Now the grammar is

$$S \rightarrow XSS' \mid aS'$$
$$S' \rightarrow XS' \mid SbS' \mid \varepsilon$$
$$X \rightarrow SaX'$$
$$X' \rightarrow bX' \mid \varepsilon$$

Example of elimination indirect left recursion:

$S \rightarrow A\,A \mid 0$

$A \rightarrow S\,S \mid 1$

Considering the ordering S, A, we get:

$S \rightarrow A\,A \mid 0$

$A \rightarrow AAS \mid 0S \mid 1$

And removing immediate left recursion, we get

$S \rightarrow A\,A \mid 0$

$A \rightarrow 0S\,A' \mid 1A'$

$A' \rightarrow \varepsilon \mid ASA'$

STEP:FIRST & FOLLOW

Why using FIRST and FOLLO

During parsing FIRST and FOLLOW help us to choose which production to apply , based on the next input signal.

We know that we need of backtracking in syntax analysis, which is really a complex process to implement. There can be easier way to sort out this problem by using FIRST AND FOLLOW.

If the compiler would have come to know in advance, that what is the “first character of the string produced when a production rule is applied”, and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply .

FOLLOW is used only if the current non terminal can derive ε .

Rules of FIRST

FIRST always find out the terminal symbol from the grammar. When we check out FIRST for any symbol then if we find any terminal symbol in first place then we take it. And not to see the next symbol.

- If a grammar is

  $A \rightarrow a$ then FIRST (A)={ a }

- If a grammar is

  $A \rightarrow a\ B$ then FIRST (A)={ a }

Rules of FIRST

- If a grammar is

  A→ aB lε then FIRST (A)={ a, ε }

- If a grammar is

  A → BcD lε

  B → eD l(A)

  Here B is non terminal. So, we check the transition of B and find the FIRST of A.

  then FIRST (A)={ e,(, ε}

Rules of FOLLOW

For doing FOLLOW operation we need FIRST operation mostly. In FOLLOW we use a $ sign for the start symbol. FOLLOW always check the right portion of the symbol.

- If a grammar is

  A → BAc ; A is start symbol.

Here firstly check if the selected symbol stays in right side of the grammar. We see that c is right in A.

then FOLLOW (A) = {c , $ }

Rules of FOLLOW

- If a grammar is

$A \rightarrow BA'$

$A' \rightarrow *Bc$

Here we see that there is nothing at the right side of A' . So

FOLLOW (A')= FOLLOW (A)= { $ }

Because A' follows the start symbol.

Rules of FOLLOW

- If a grammar is

  A → BC

  B → Td

  C →*D l ε

When we want to find FOLLOW (B),we see that B follows by C . Now put the FIRST(C) in the there.

FIRST(C)={*, ε }.

But when the value is € it follows the parents symbol. So FOLLOW(B)={*,$ }

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E		
E’		
T		
T’		
F		

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	
E'		
T		
T'		
F		

Example of FIRST and FOLLOW

GRAMMAR:
E -> TE'
E'-> +TE'|ε
T -> FT'
T' -> *FT'|ε
F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	
E'	{ + , ε }	
T		
T'		
F		

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	
E'	{ + , ε }	
T	{ id , (}	
T'		
F		

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	
E'	{ + , ε }	
T	{ id , (}	
T'	{ * , ε }	
F		

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	
E'	{ + , ε }	
T	{ id , (}	
T'	{ * , ε }	
F	{ id , (}	

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E'	{ + , ε }	
T	{ id , (}	
T'	{ * , ε }	
F	{ id , (}	

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E'	{ + , ε }	{ $,) }
T	{ id , (}	
T'	{ * , ε }	
F	{ id , (}	

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E'	{ + , ε }	{ $,) }
T	{ id , (}	{ $,) ,+ }
T'	{ * , ε }	
F	{ id , (}	

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E'	{ + , ε }	{ $,) }
T	{ id , (}	{ $,) ,+ }
T'	{ * , ε }	{ $,) , + }
F	{ id , (}	

Example of FIRST and FOLLOW

GRAMMAR:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> *FT'|ε

F -> (E)|id

Symbol	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E'	{ + , ε }	{ $,) }
T	{ id , (}	{ $,) ,+ }
T'	{ * , ε }	{ $,) , + }
F	{ id , (}	{ $,) , + , * }

STEP: PARSING TABLE

Example of LL(1) grammar

E -> TE’

E’ -> +TE’|ε

T -> FT’

T’ -> *FT’|ε

F -> (E)|id

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
FIRST & FOLLOW

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E						
E’						
T						
T’						
F						

TABLE:
FIRST & FOLLOW

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’					
E’						
T						
T’						
F						

TABLE:
FIRST & FOLLOW

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’						
T						
T’						
F						

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
FIRST & FOLLOW

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’-> +TE’				
T						
T’						
F						

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
FIRST & FOLLOW

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’-> +TE’			E’ -> ε	E’ -> ε
T						
T’						
F						

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
FIRST & FOLLOW

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’-> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’					
T’						
F						

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
FIRST & FOLLOW

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’-> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’						
F						

TABLE:
FIRST & FOLLOW

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’-> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’			T’ -> *FT’			
F						

TABLE:
FIRST & FOLLOW

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	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						

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
FIRST & FOLLOW

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	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					

Production	Symbol	FOLLOW
E -> TE’	{ (, id }	{ $,) }
E’ -> +TE’|ε	{ + , ε }	{ $,) }
T -> FT’	{ (, id }	{ + , $,) }
T’ -> *FT’|ε	{ * , ε }	{ + , $,) }
F -> (E)|id	{ (, id }	{ *, + , $,) }

TABLE:
FIRST & FOLLOW

TABLE:
PARSING
TABLE

Non Terminal	INPUT SYMBOLS					
	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)		

Continue…

Non Terminal	INPUT SYMBOLS					
	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)		

TABLE: PARSING TABLE

- This grammar is LL(1).
- So, the parse tree can be derived from the stack implementation of the given parsing table.

Continue…

- There are grammars which may requite LL(1) parsing.
- For e.g. Look at next grammar…..

Continue…

GRAMMAR:

S → iEtSS' | a

S' → eS |ε

E → b

TABLE: FIRST & FOLLOW

SYMBOL	FIRST	FOLLOW
S	a , i	$, e
S'	e , ε	$, e
E	b	t

TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S						
S'						
E						

TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a					
S'						
E						

TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'						
E						

TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S' → eS			
E						

TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S' → eS S'→ε			S'→ε
E						

TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S' → eS S'→ε			S'→ε
E		E → b				

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
FAST & FOLLOW

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S' → eS S'→ε			S'→ε
E		E → b				

AMBIGUITY

```
S --> A | a
 A --> a
```

	First	follow
S –> A/a	{ a }	{ $ }
A –>a	{ a }	{ $ }

Not LL1Grammar

	a	$
S	S –> A, S –> a	
A	A –> a	

	First()	Follow()
$S \rightarrow ABCD$	$\{a, b\}$	$\{\$\}$
$A \rightarrow a/\varepsilon$	$\{a, \varepsilon\}$	$\{b\}$
$B \rightarrow b$	$\{b\}$	$\{c, d, e\}$
$C \rightarrow c/\varepsilon$	$\{c, \varepsilon\}$	$\{d, e\}$
$D \rightarrow d/\varepsilon$	$\{d, \varepsilon\}$	$\$$

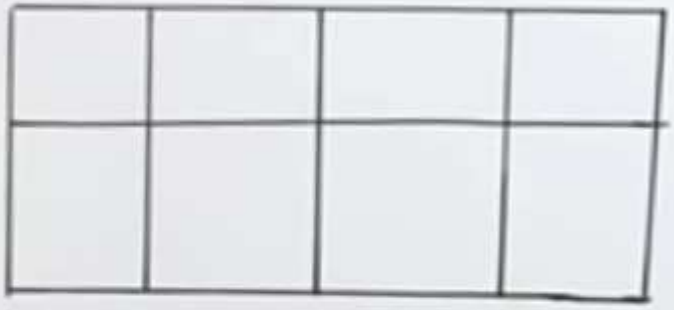
	a	b	c	d	e	$
S	$S \rightarrow ABCD$	$S \rightarrow ABCD$				
A	$A \rightarrow a$	$A \rightarrow \varepsilon$				
B		$B \rightarrow b$				
C			$C \rightarrow c$	$C \rightarrow \varepsilon$	$C \rightarrow \varepsilon$	
D				$D \rightarrow d$		$D \rightarrow \varepsilon$

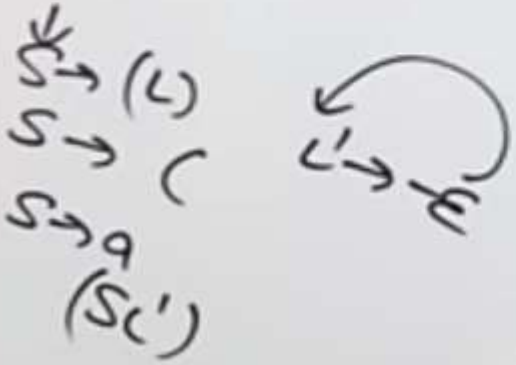
"LL(1) Grammar Parsing Table"

	First	Follow
$S \to^{1} (L) \mid^{2} a$	{(, a}	{$, ,,)}
$L \to SL'^{3}$	{(, a}	{)}
$L' \to \underset{4}{\varepsilon} \mid , S\underset{5}{L'}$	{ε, ,}	{)}

	(	)	a	,	$
S	S	1		2		
L	L	3		3		
L'	L'		4		5	

$S \to aSbS \mid bSaS \mid \varepsilon$





$S \to \underline{a^{1}SbS} \mid bSaS \mid \varepsilon^{3}$ (first / follow)

(a, b, ε) / b, a, $

	a	b	$
S	1/3	2/3	3

Checking of LL(1) grammar

$S \to (aSa) \mid (bS) \mid (c)$

$\alpha_1 \quad \alpha_2 \quad \alpha_3$

a b c

$S \to iCtSS_1 \mid a$

$S_1 \to eS \mid \epsilon$

$C \to b$

$S \to \alpha_1 \mid \alpha_2 \mid \alpha_3$ No Null. ϵ

$F(\alpha_1) \cap F(\alpha_2) \cap F(\alpha_3) = \phi$ LL(1)

$\neq \phi$ Not LL(1)

$S \to \alpha_1 \mid \alpha_2 \mid \epsilon$

LL(1) $F(\alpha_1) \cap F(\alpha_2) \cap Follow(S)$

Not LL(1) $\neq \phi$

$= \phi$

$S \to aSbS$
$/bSaS$
$/\epsilon$

$S \to aABb$
$A \to c/\epsilon$
$B \to d/\epsilon$

$S \to A/a$
$A \to a$

$S \to aB/\epsilon$
$B \to bC/\epsilon$
$C \to cS/\epsilon$

$S \to AB$
$A \to a/\epsilon$
$B \to b/\epsilon$

$S \to aSA/\epsilon$
$A \to c/\epsilon$

$S \to A$
$A \to Bb/Cd$
$B \to aB/\epsilon$
$C \to cC/\epsilon$

$S \to aAa/\epsilon$
$A \to abS/\epsilon$

$S \to iEtSS'/a$
$S' \to eS/\epsilon$
$E \to b$

Continue…

- The grammar is ambiguous and it is evident by the fact that we have two entries corresponding to M[S',e] containing S' → ε and S' → eS.
- Note that the ambiguity will be solved if we use LL(2) parser, i.e. Always see for the two input symbols.
- LL(1) grammars have distinct properties.
  - No ambiguous grammar or left recursive grammar can be LL(1).
- Thus , the given grammar is not LL(1).

STEP: STACK IMPLEMENTATION

STACK Implementation

- The predictive parser uses an explicit stack to keep track of pending non-terminals. It can thus be implemented without recursion.
- Note that productions output are tracing out a *lefmost derivation*
- The grammar symbols on the stack make up *left-sentential forms*

LL(1) Stack

- The *input buffer* contains the string to be parsed; $ is the end-of-input marker
- The *stack* contains a sequence of grammar symbols
- Initially, the stack contains the start symbol of the grammar on the top of $.

LL(1) Stack

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, *X* and a determine the action of the parser.
  - There are *three* possibilities.

LL(1) Stack

1. $X = a = \$$,
   the parser halts and annouces successful completion.
2. $X = a \neq \$$
   the parser pops x off the stack and advances input pointer to next input symbol
3. If X is a nonterminal, the program consults entry $M[x,a]$ of parsing table M.

   If the entry is a production $M[x,a] = \{x \rightarrow uvw\}$ then the parser replaces x on top of the stack by wvu (with u on top).

   As output, the parser just prints the production used:
   $x \rightarrow uvw$.

LL(1) Stack

Grammar:

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

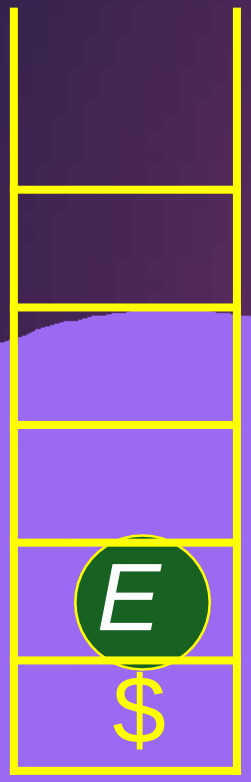
Example:

Let's parse the input string

id+id∗id

Using the nonrecursive LL(1) parser

id + id * id $



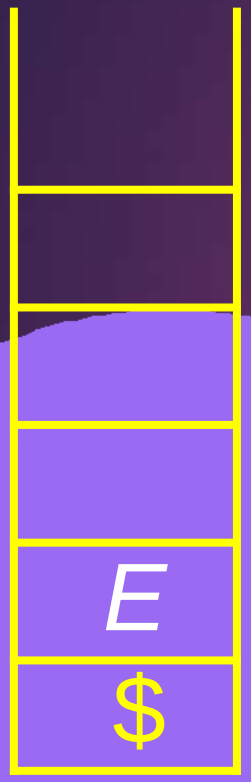
stack

Parsing Table

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			$F \rightarrow (E)$		

id + id * id $

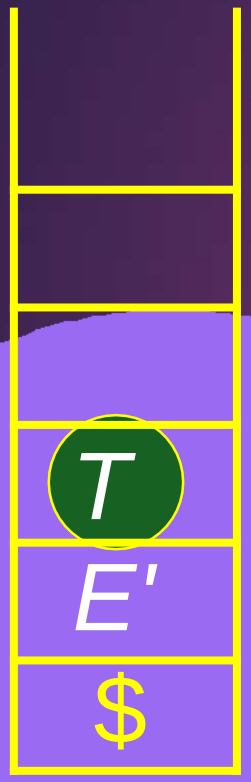


stack

$E \rightarrow TE'$

Parsing Table

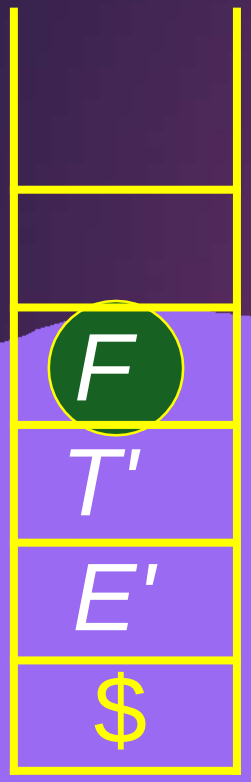
id + id * id $



stack

Non Terminal	INPUT SYMBOLS					
	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)		

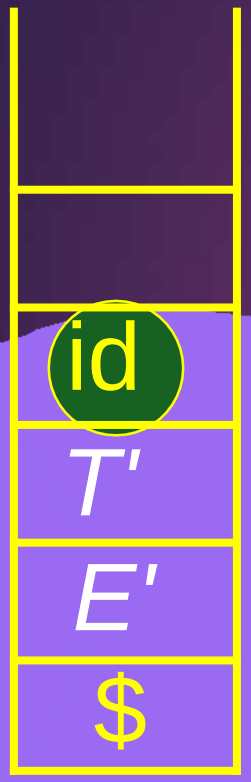
id + id * id $



stack

Non Terminal	INPUT SYMBOLS					
	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)		

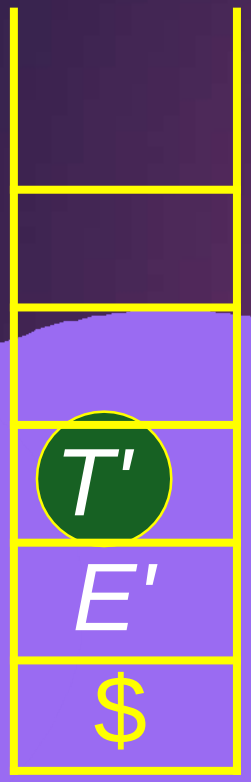
id + id * id $



stack

Non Terminal	INPUT SYMBOLS					
	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)		

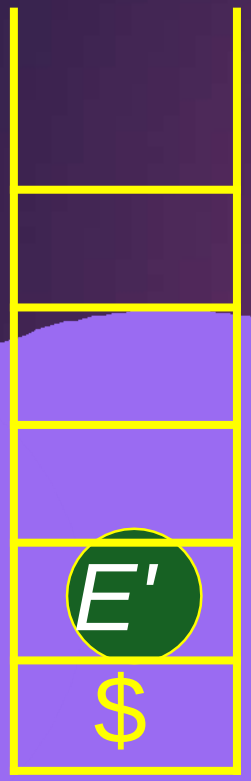
+ id * id $



stack

Non Terminal	INPUT SYMBOLS					
	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)		

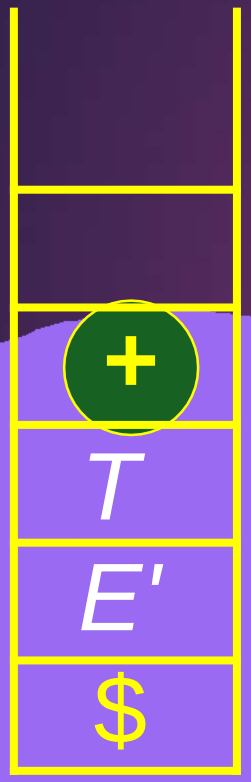
+ id * id $



stack

Non Terminal	INPUT SYMBOLS					
	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)		

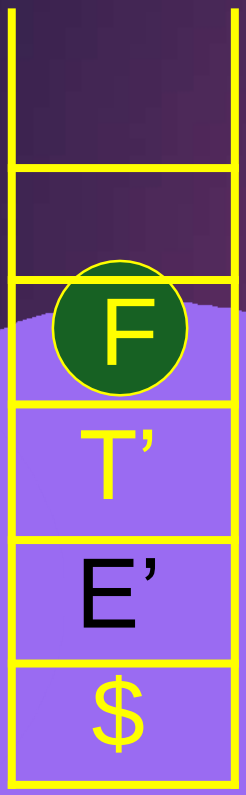
+ id * id $



stack

Non Terminal	INPUT SYMBOLS					
	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)		

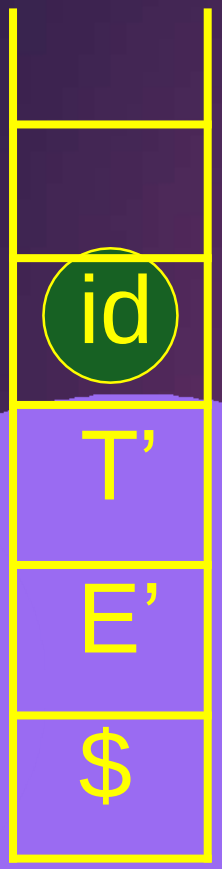
id * id $



stack

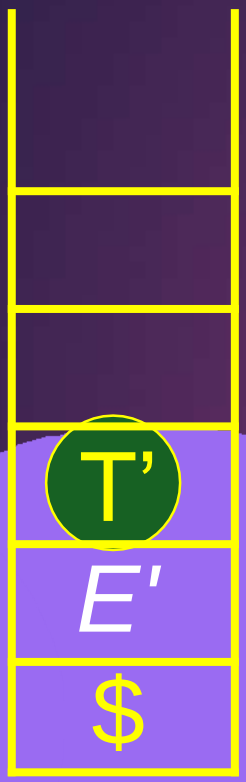
Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E' -> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> id			F -> (E)		

id * id $



stack

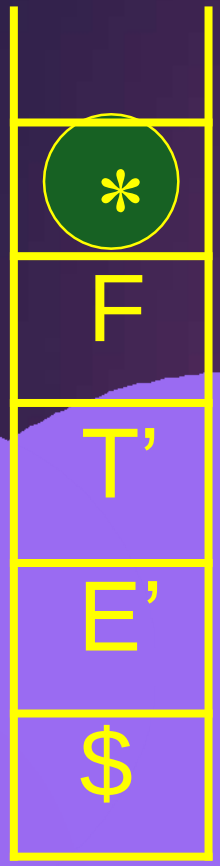
Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E' -> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> id			F -> (E)		



stack

* id $

Non Terminal	INPUT SYMBOLS					
	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)		

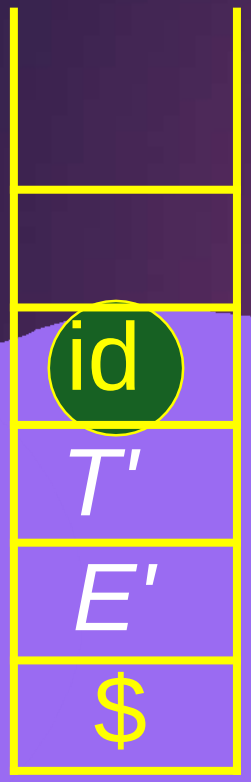


stack

* id $

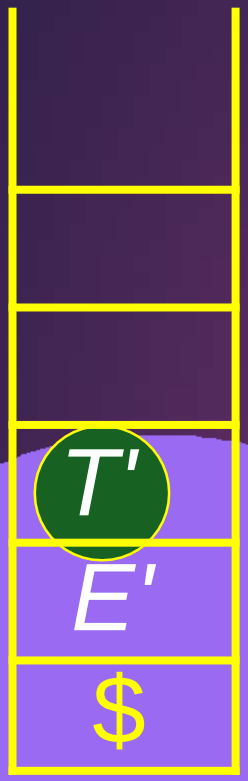
Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E' -> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> id			F -> (E)		

id $



stack

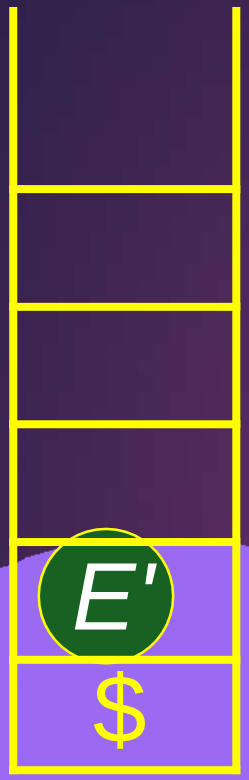
Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> FT'			T -> FT'		
T'		T' -> ε	T' -> *FT'		T' -> ε	T' -> ε
F	F -> id			F -> (E)		



stack

$

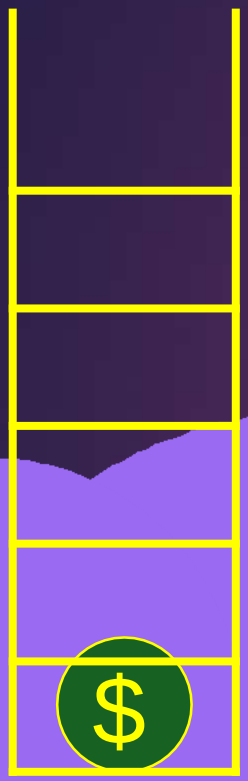
Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> FT'			T -> FT'		
T'		T' -> ε	T' -> *FT'		T' -> ε	T' -> ε
F	F -> id			F -> (E)		



stack

$

Non Terminal	INPUT SYMBOLS					
	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)		



stack

Non Terminal	INPUT SYMBOLS					
	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)		

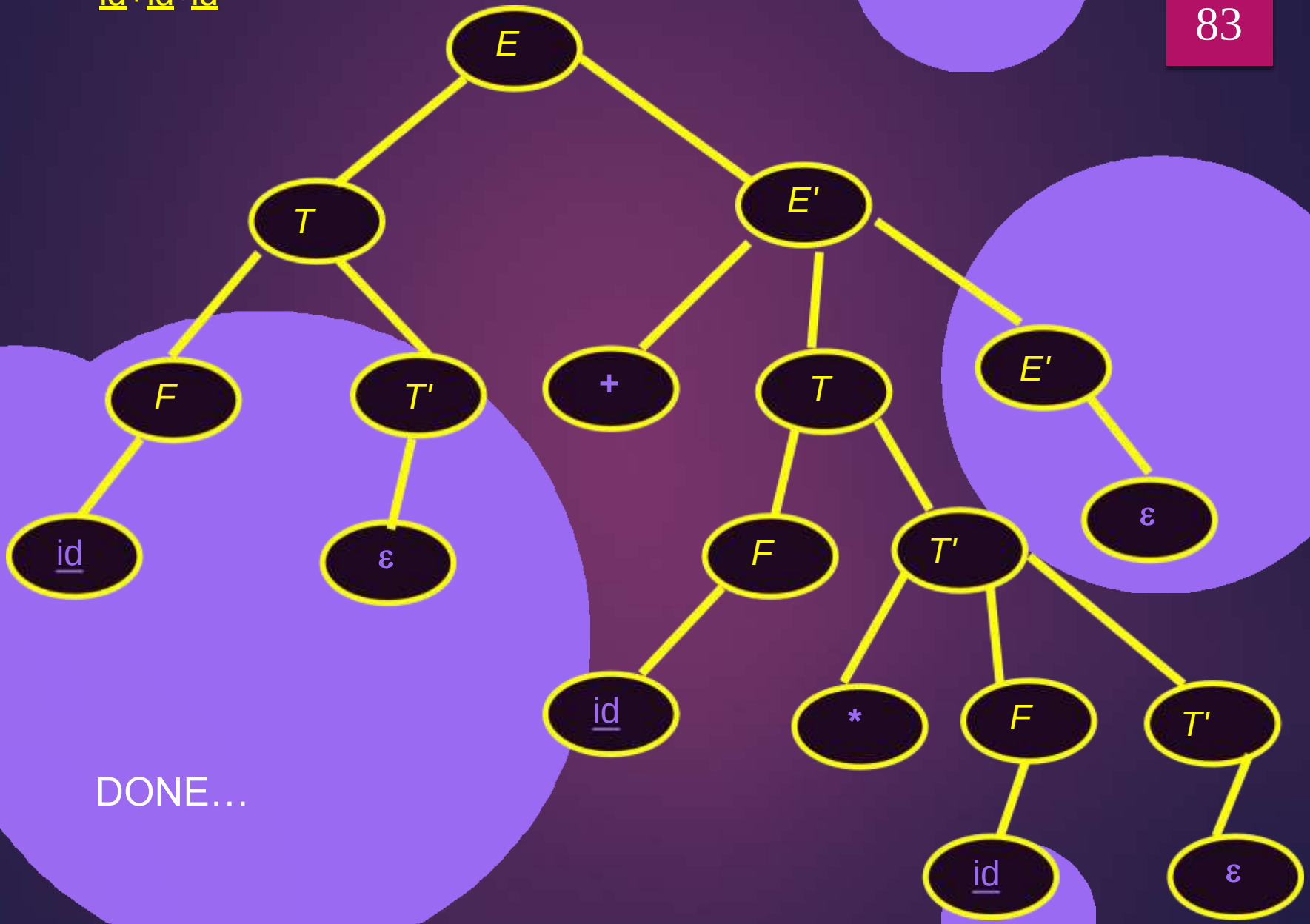
MATCHED	STACK	INPUT	ACTION
	E$	id+id * id$	
	TE'$	id+id * id$	E->TE'
	FT'E'$	id+id * id$	T->FT'
	id T'E'$	id+id * id$	F->id
id	T'E'$	+id * id$	Match id
id	E'$	+id * id$	T'->Є
id	+TE'$	+id * id$	E'-> +TE'
id+	TE'$	id * id$	Match +
id+	FT'E'$	id * id$	T-> FT'
id+	idT'E'$	id * id$	F-> id
id+id	T'E'$	* id$	Match id
id+id	* FT'E'$	* id$	T'-> *FT'
id+id *	FT'E'$	id$	Match *
id+id *	idT'E'$	id$	F-> id
id+id * id	T'E'$	$	Match id
id+id * id	E'$	$	T'-> Є
id+id * id	$	$	E'-> Є

STEP: LL(1) PARSE TREE

LL(1) Parse Tree

- Top-down parsing expands a parse tree from the start symbol to the leaves
- Always expand the leftmost non-terminal

id+id∗id



DONE…

$S \rightarrow AA$ 1

$A \rightarrow aA$ 2

$A \rightarrow b$ 3

			$

I/P Buffer

→ dP

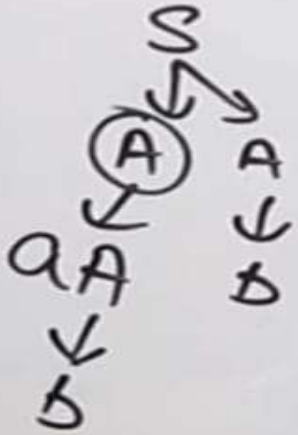
a	b	a	b	$

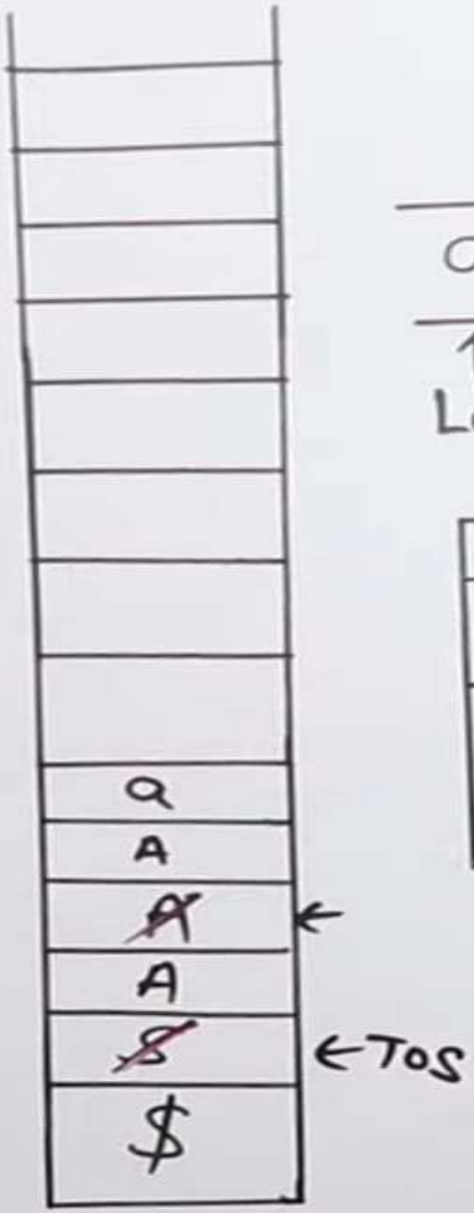
I/P Buffer

↑ Look ahead

	a	b	$
S	S→AA (1)	S→AA (1)	
A	A→aA (2)	A→b (3)	

PT

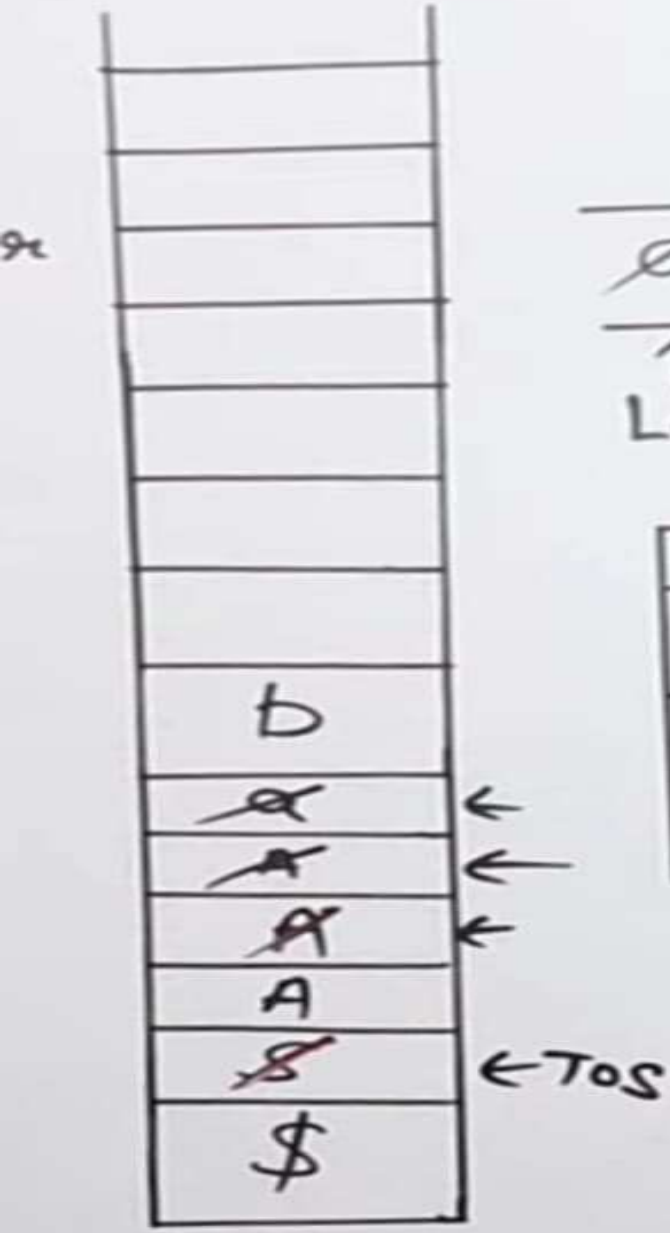




$S \rightarrow AA$ 1

$A \rightarrow aA$ 2

$A \rightarrow b$ 3



a	b	a	b	$

I/P Buffer

Look ahead

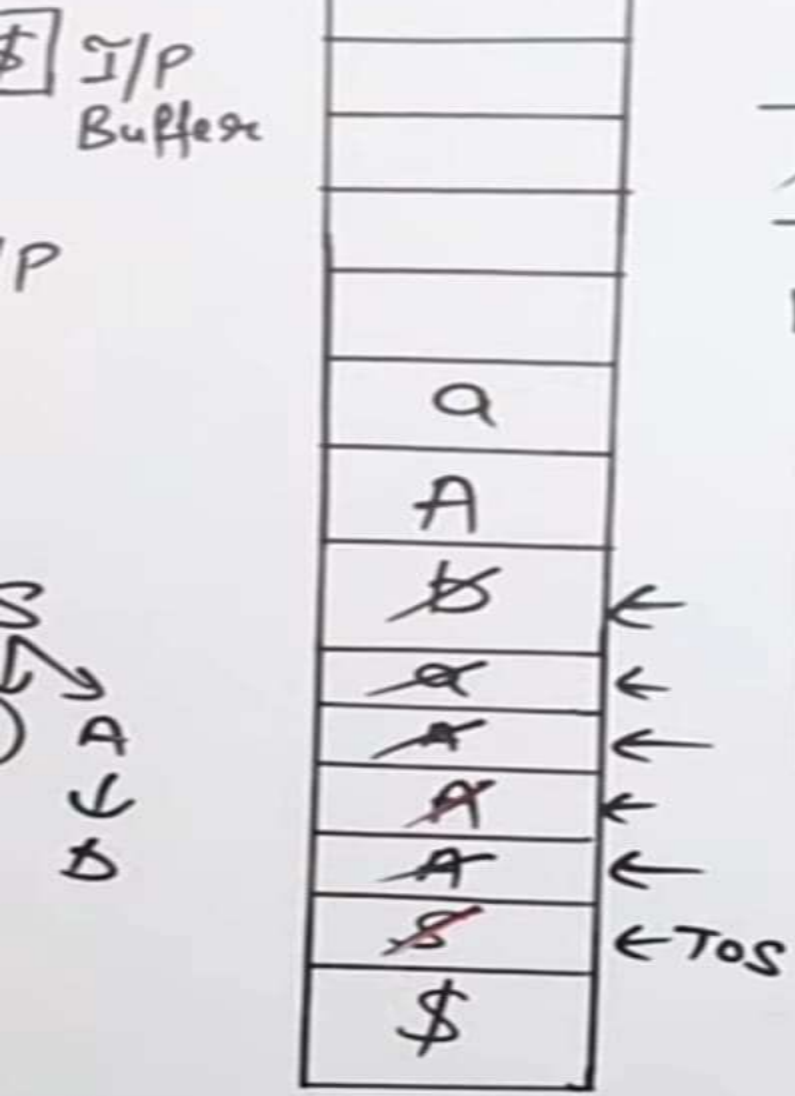
	a	b	$
S	S→AA (1)	S→AA (1)	
A	A→aA (2)	A→b (3)	

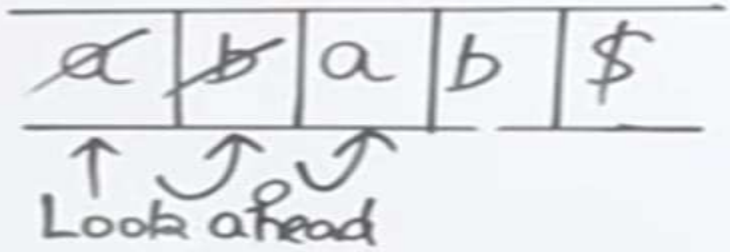
PT

$S \rightarrow AA$ 1

$A \rightarrow aA$ 2

$A \rightarrow b$ 3





	a	b	$
S	S→AA (1)	S→AA (1)	
A	A→aA (2)	A→b (3)	

PT

$S \rightarrow AA$ 1

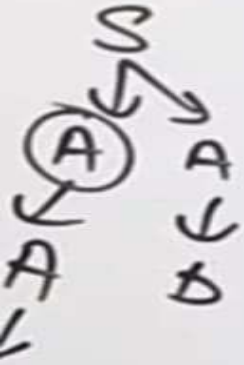
$A \rightarrow aA$ 2

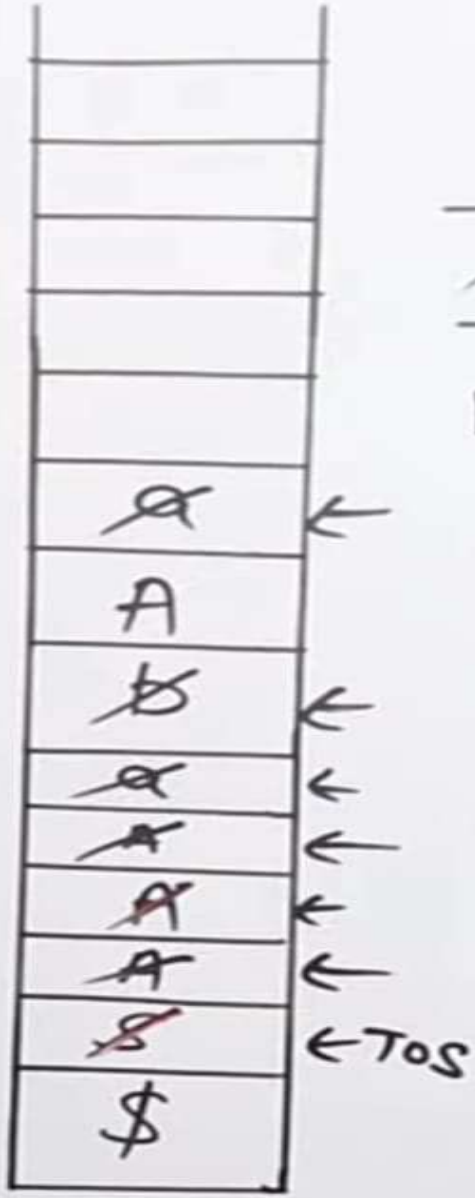
$A \rightarrow b$ 3

	$

I/P Buffer

> o/P





a	b	a	b	$

I/P Buffer

Look ahead

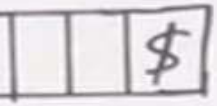
	a	b	$
S	S→AA 1	S→AA 1	
A	A→aA 2	A→b 3	

PT

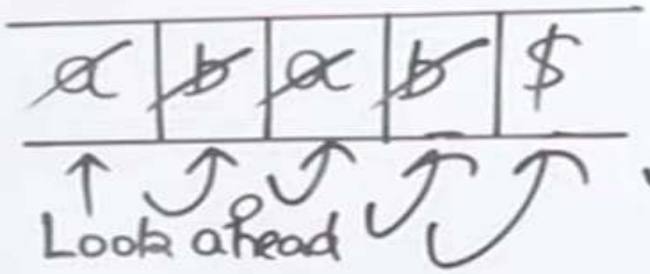
$S \rightarrow AA$ 1

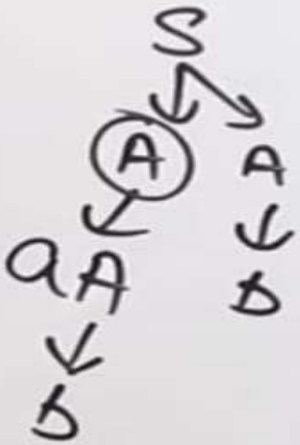
$A \rightarrow aA$ 2

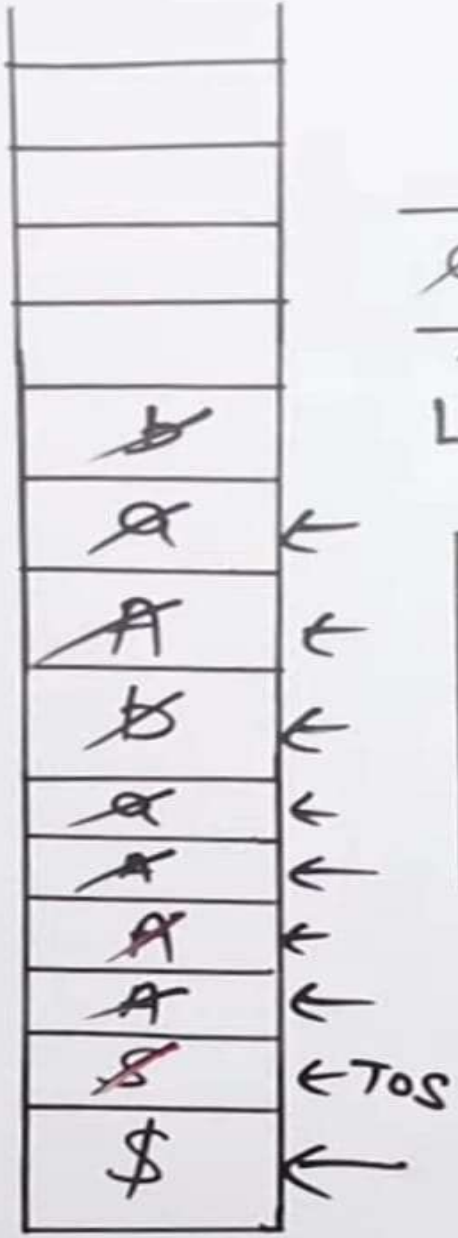
$A \rightarrow b$ 3



→ o/p







	a	b	$
S	S→AA (1)	S→AA (1)	
A	A→aA (2)	A→b (3)	

PT

exp --> term exp'

exp’ --> addop term exp' | empty

addop --> + | -

term --> factor term'

term’ --> mulop factor term' | empty

mulop --> *

factor --> (exp) | num

exp --> term exp'
exp’ --> addop term exp' | empty
addop --> + | -
term --> factor term'
term’ --> mulop factor term' |
empty
mulop --> *
factor --> (exp) | num

Example: Constructing LL(1) Parsing Table

	First	Follow
exp	{(, num}	{$,)}
exp’	{+,-, λ}	{$,)}
addop	{+,-}	{(,num}
term	{(,num}	{+,-,),$}
term’	{*, λ}	{+,-,),$}
mulop	{*}	{(,num}
factor	{(, num}	{*,+,-,),$}

1 exp → term exp’
2 exp’ → addop term exp’
3 exp’ → λ
4 addop → +
5 addop → -
6 term → factor term’
7 term’ → mulop factor term’
8 term’ → λ
9 mulop → *
10 factor → (exp)
11 factor → num

	(	)	+	-	*	n	$
exp	1					1	
exp’		3	2	2			3
addop			4	5			
term	6					6	
term’		8	8	8	7		8
mulop					9		
factor	10					11	

Stack implementation for given input string

(n+n)

n + n * n

n*/n

LL(1) Parser – Example1

$S \rightarrow aBa$
$B \rightarrow bB \mid \varepsilon$

Input : abba

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

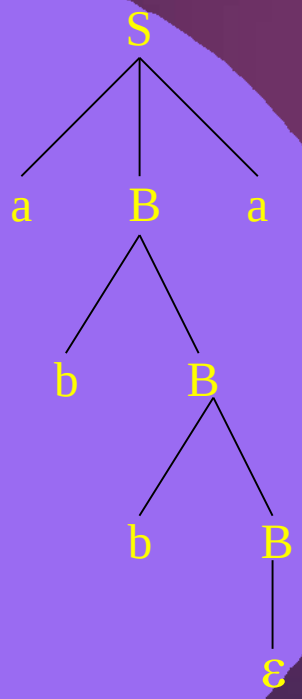
LL(1) Parsing Table

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

LL(1) Parser – Example1

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

LL(1) Parser – Example2

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id \mid (E)$

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

Input : id +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

stack	input	output
$E	id+id$	$E \rightarrow TE'$
$E'T	id+id$	$T \rightarrow FT'$
$E' T'F	id+id$	$F \rightarrow id$
$ E' T'id	id+id$	
$ E' T'	+id$	$T' \rightarrow \varepsilon$
$ E'	+id$	$E' \rightarrow +TE'$
$ E' T+	+id$	
$ E' T	id$	$T \rightarrow FT'$
$ E' T' F	id$	$F \rightarrow id$
$ E' T'id	id$	
$ E' T'	$	$T' \rightarrow \varepsilon$
$ E'	$	$E' \rightarrow \varepsilon$
$	$	accept

A Grammar which is not LL(1)

$S \rightarrow i\ C\ t\ S\ E \mid a$

$E \rightarrow e\ S \mid \varepsilon$

$C \rightarrow b$

FIRST(iCtSE) = {i}

FIRST(a) = {a}

FIRST(eS) = {e}

FIRST(ε) = {ε}

FIRST(b) = {b}

FIRST(S) = {i , a}

FIRST(E) = {e , ε}

FIRST(C) = {b}

FOLLOW(S) = { $,e }

FOLLOW(E) = { $,e }

FOLLOW(C) = { t }

	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow eS$ $E \rightarrow \varepsilon$			$E \rightarrow \varepsilon$
C		$C \rightarrow b$				

two production rules for M[E,e]

Problem ➔ ambiguity

A Grammar which is not LL(1)

- What we have to do , if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears FIRST(Aα) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST(Aα) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

  1. Both α and β cannot derive strings starting with same terminals.

  3. At most one of α and β can derive to ε.

  5. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

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 recover from that error case, and it should be able to continue the parsing with the rest of the input.

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token ("synch ") for that non-terminal.
- " synch " is placed in the parsing table for the positions of follow set of that non terminal.
- 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 **"synch "** then the non terminal on top of the stack is popped in an attempt to resume parsing
- If the token on top of the stack does not match the input symbol ,then we pop the token from the stack.

Panic-Mode Error Recovery in LL(1) Parsing example

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

Panic-Mode Error Recovery in LL(1) Parsing example

stack	input	remarks
$E	id * +id$	
$E’T	id * +id$	
$E’ T’F	id * +id$	
$ 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’	$	
$	$	

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.