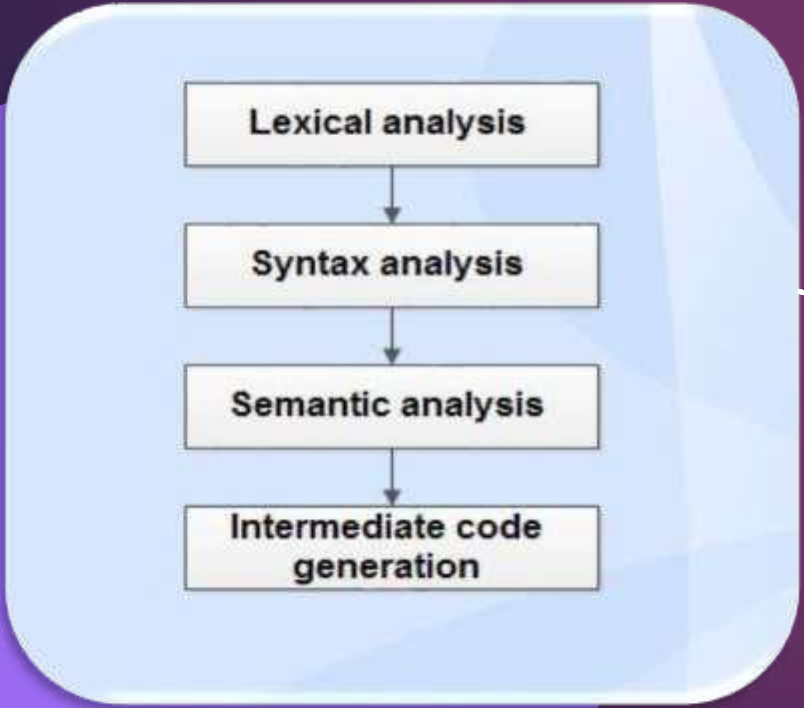
LL(1) PARSER

MODEL OF COMPILER FRONT END

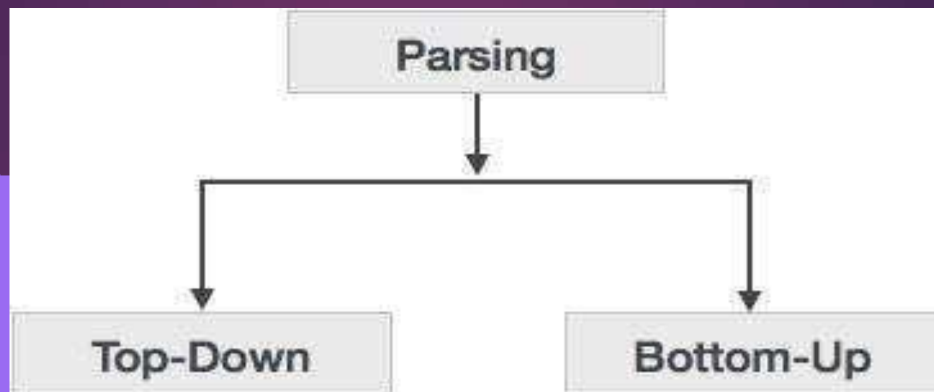
Front End



Syntax analysis

Also called parsing , where generates parse tree

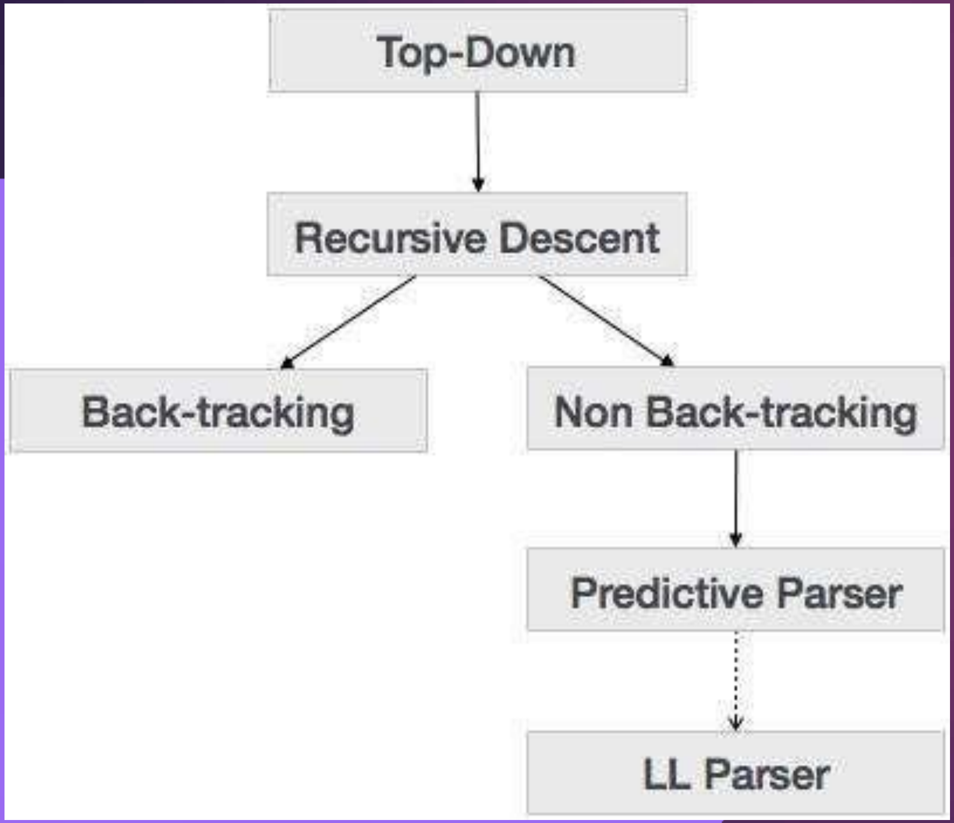
PARSING



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.

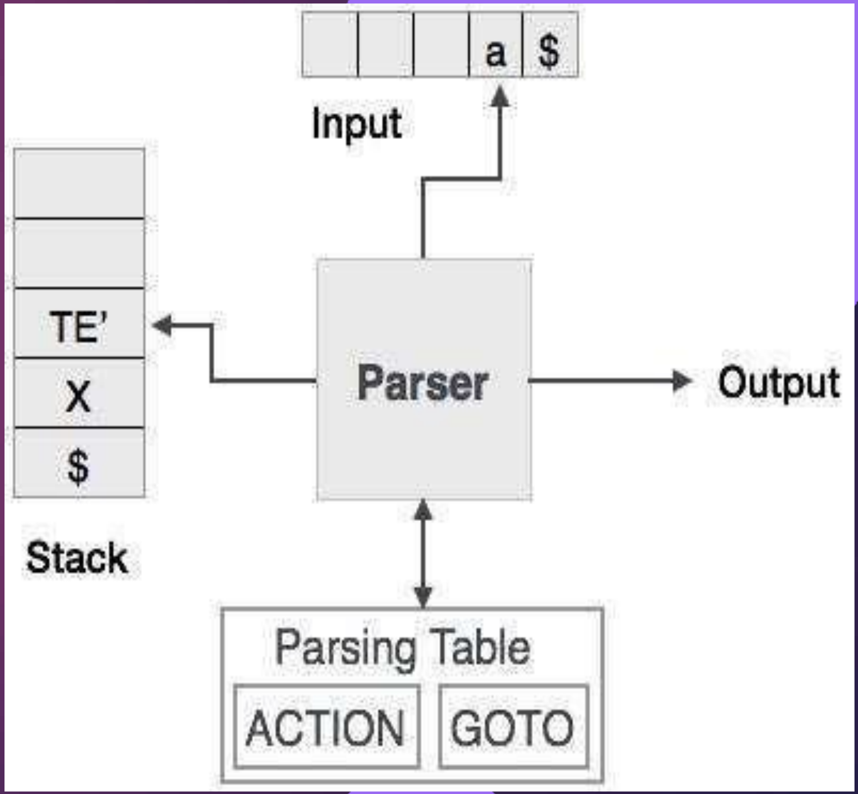
TOP DOWN PARSER



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.

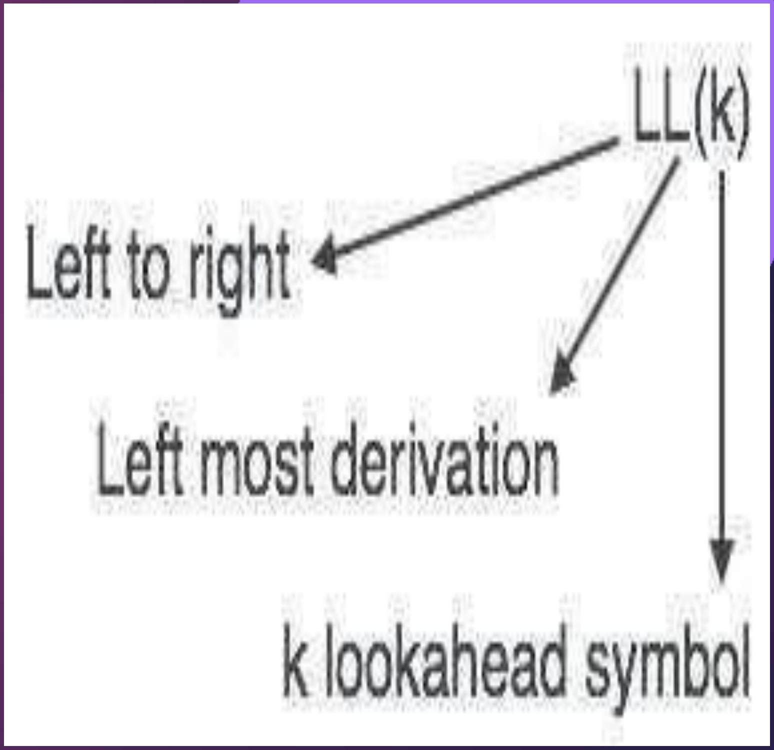


LL(1) PARSER

- 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**.

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.

TYPES OF RECURSION

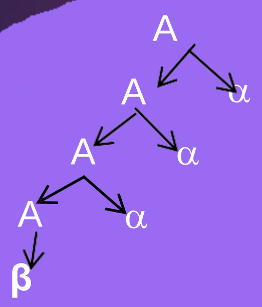
LEFT RECURSION

RIGHT RECURSION

Left Recursion

For grammar:

A -> A α**| β**

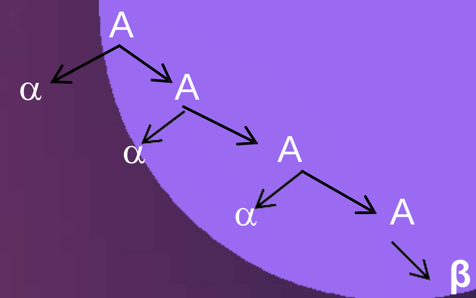


This parse tree generate **β** α *

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. **$A \rightarrow A\ \alpha | \beta$**
- 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

.

$A \longrightarrow Br$

$B \longrightarrow Cs$

$C \longrightarrow At$

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

Elimination of Left-Recursion

- Suppose the grammar were

$$A \to A\alpha \mid \beta$$

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

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

Replace

$$A \to A\alpha \mid \beta$$

With

$$A \to \beta A'$$

$$A' \to \alpha A' \mid \varepsilon$$

EXAMPLE OF IMMEDIATE LEFT RECURSION:

Consider the left recursive grammar

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

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

$$E \rightarrow T\,E'$$

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

$$T \rightarrow F\,T'$$

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

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

Continue…

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

$$A \rightarrow A\alpha 1 \mid A\alpha 2 \mid \cdot \cdot \cdot \mid A\alpha m \mid \beta 1 \mid \beta 2 \mid \cdot \cdot \cdot \mid \beta n$$

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

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

$$A' \rightarrow \alpha 1A' \mid \alpha 2A' \mid \cdot \cdot \cdot \mid \alpha mA' \mid \varepsilon$$

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:

$$S \rightarrow AA \mid 0$$
$$A \rightarrow SS \mid 1$$

Considering the ordering S, A, we get:

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

And removing immediate left recursion, we get

$$S \rightarrow AA \mid 0$$
$$A \rightarrow 0SA' \mid 1A'$$
$$A' \rightarrow \varepsilon \mid ASA'$$

STEP:FIRST & FOLLOW

Why using FIRST and FOLLOW

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

  $A \rightarrow BAc$; 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 | 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

<u>GRAMMAR:</u>

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

<u>GRAMMAR:</u>
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						

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’				
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						

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’						
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						

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

TABLE: FIRST & FOLLOW

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: PARSING TABLE

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						

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	F -> id					

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	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

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

TABLE: FIRST & FOLLOW

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

TABLE: PARSING TABLE

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

TABLE: FIRST & FOLLOW

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

TABLE: PARSING TABLE

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

TABLE: FIRST & FOLLOW

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

TABLE: PARSING TABLE

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

TABLE:
FIRST & FOLLOW

TABLE:
PARSING TABLE

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

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

TABLE:
FIRST & FOLLOW

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

TABLE:
PARSING TABLE

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

TABLE:
FIRST & FOLLOW

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

TABLE:
PARSING TABLE

TABLE: FIRST & 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				

AMBIGUITY

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

<u>Grammar:</u>

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

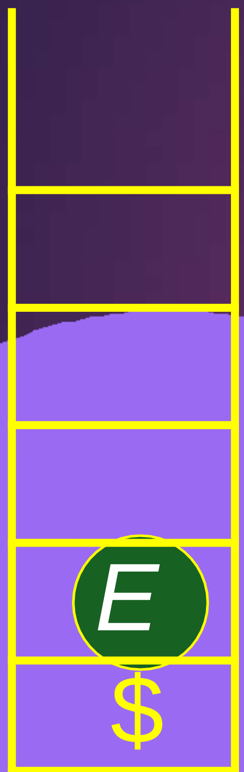
Example:

Let’s parse the input string

<u>id</u>+<u>id</u>∗<u>id</u>

Using the nonrecursive LL(1) parser

id + id * id $



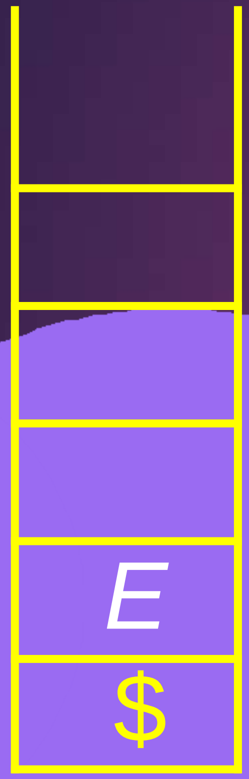
stack

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

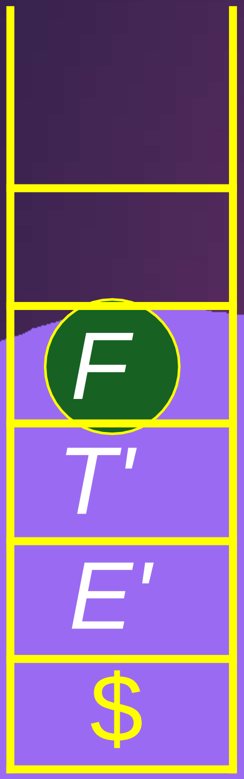
id + id * id $

T
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)		

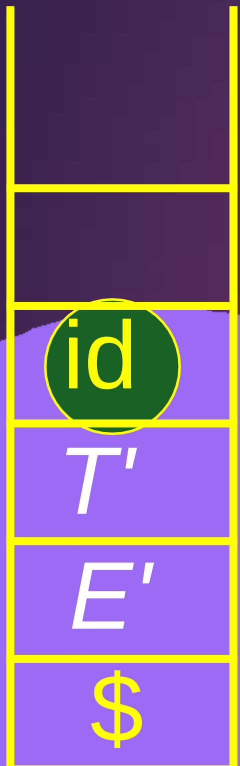
id + id * id $



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)		

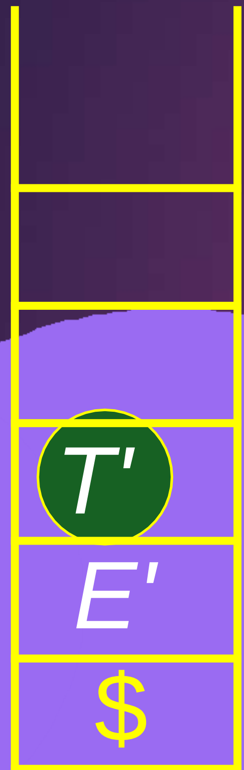
id + id * id $



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)		

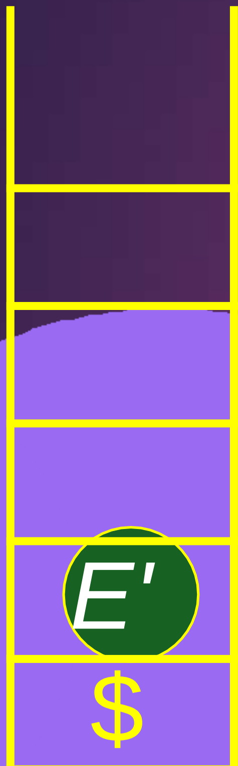
\+ id * id $



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 $

+
T
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)		

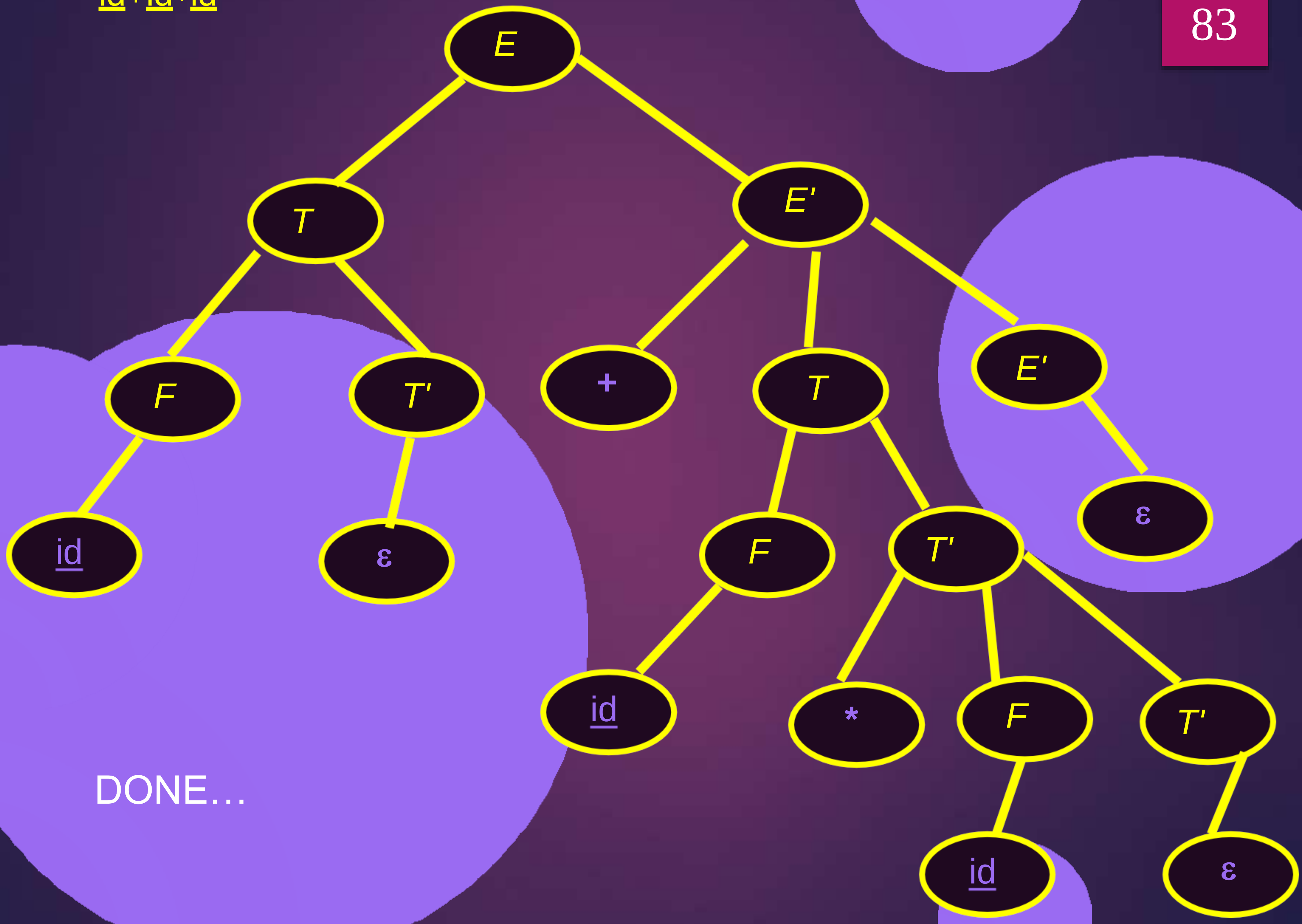
MATCHED	STACK	INPUT	ACTION
	E$	id+id * id$	
	TE’$	id+id * id$	E->TE’
	FT’E’$	id+id * id$	T->FT’
	id T’E’$	id+id * id$	F->id
id	T’E’$	+id * id$	Match id
id	E’$	+id * id$	T’->Є
id	+TE’$	+id * id$	E’-> +TE’
id+	TE’$	id * id$	Match +
id+	FT’E’$	id * id$	T-> FT’
id+	idT’E’$	id * id$	F-> id
id+id	T’E’$	* id$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