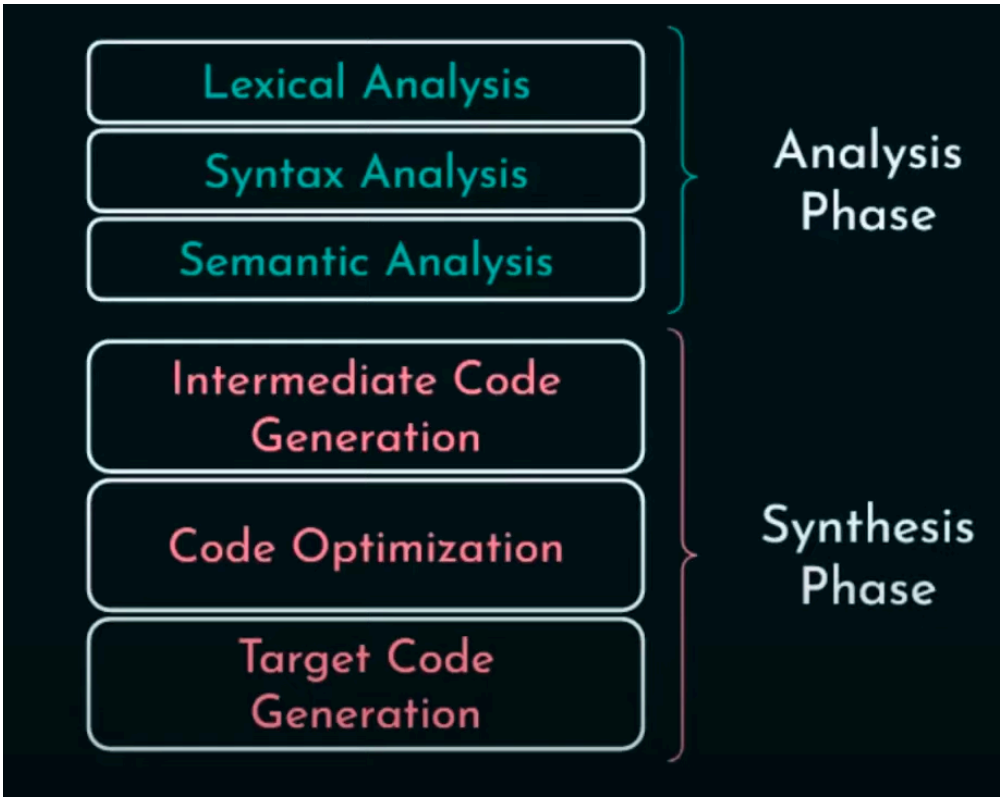
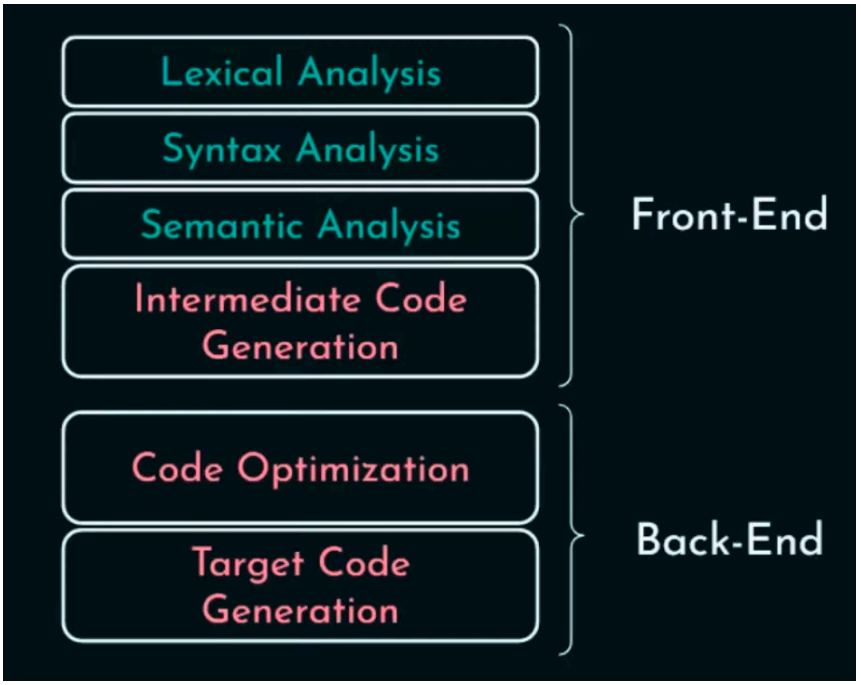
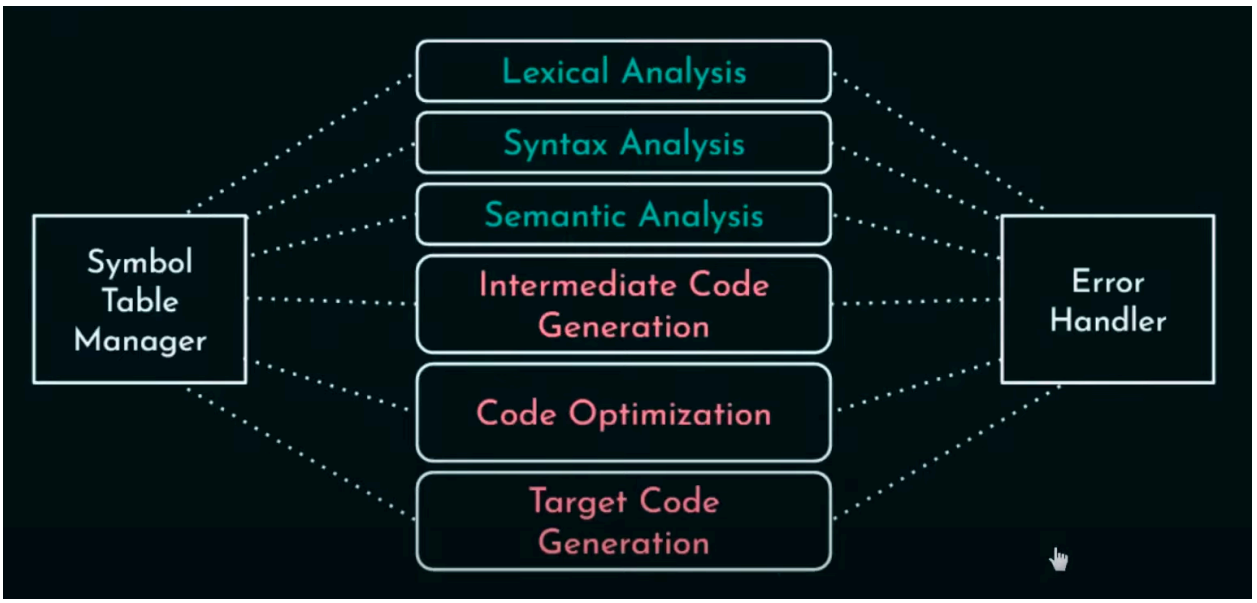
In His name

Compiler phases







Lexical Analyzer:

`x = a+b*c;`

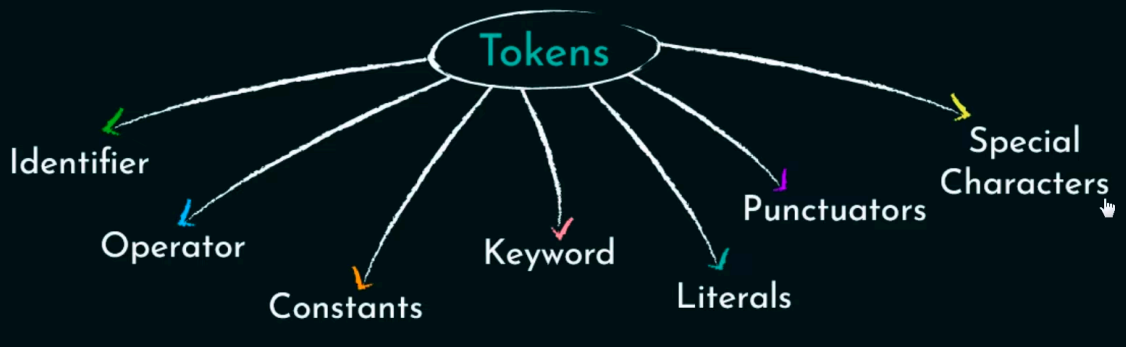
Lexical Analysis

Recognizes tokens using Regexs.
E.g. Regex for identifier:
l(l+d)*|_(l+d)*
l : letter
d : digit
_ : underscore

Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

Lexical Analyzer:

- Scans the Pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens.



Syntax Analyzer:

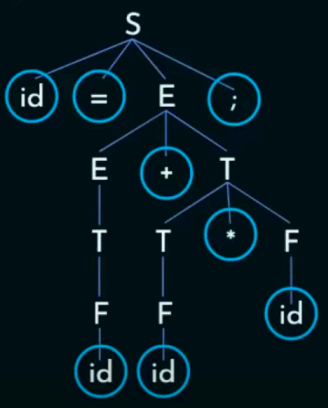
Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

x = a+b*c;

Syntax Analysis

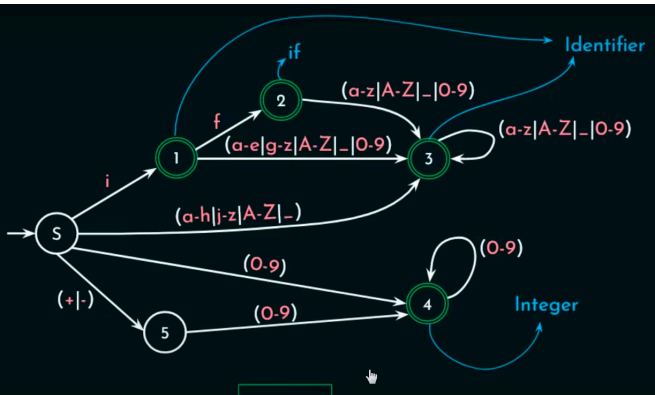
S → id = E ;
E → E + T|T
T → T * F|F
F → id

id = id + id * id ;

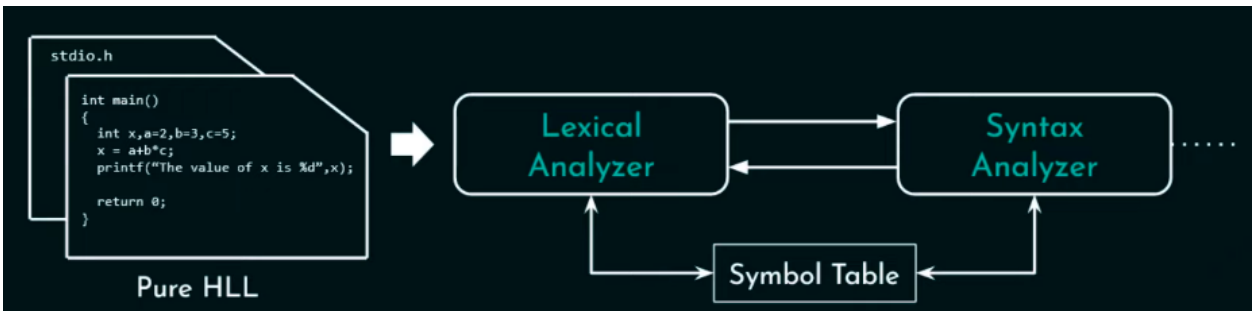


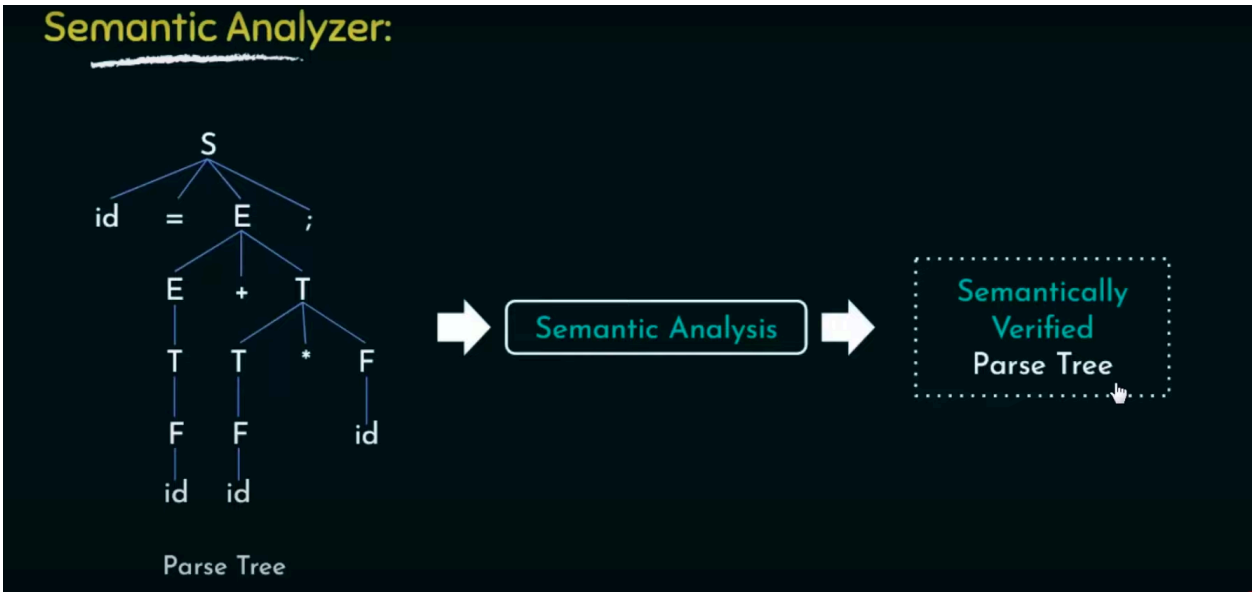
Parse Tree

C-Tokens:

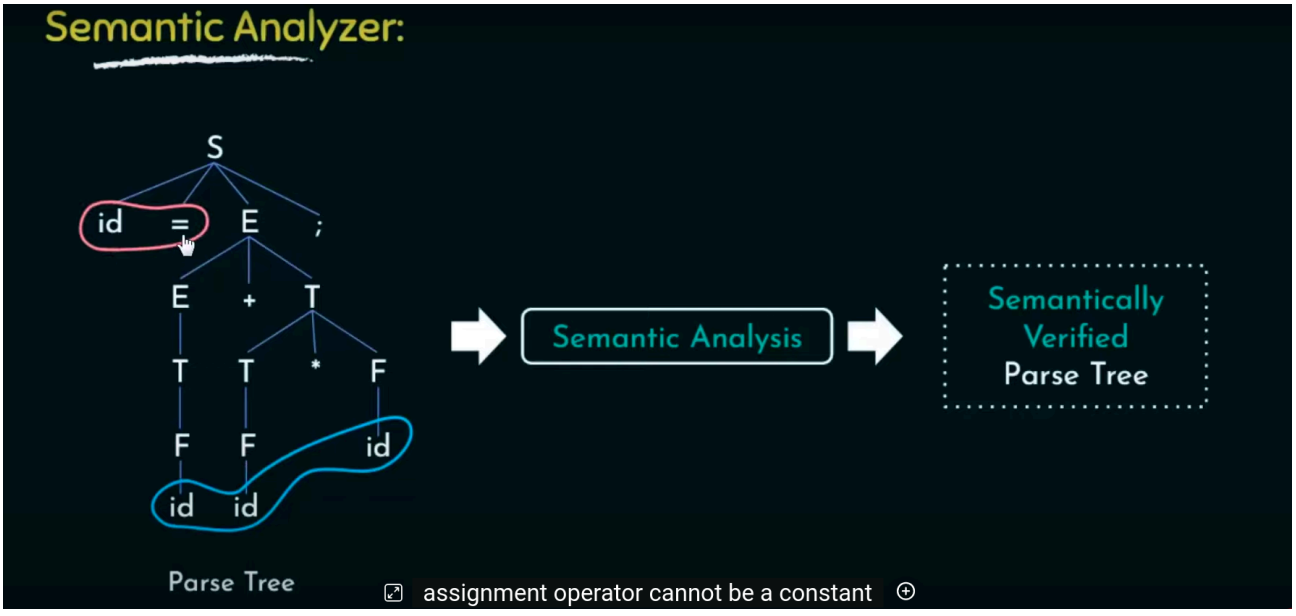


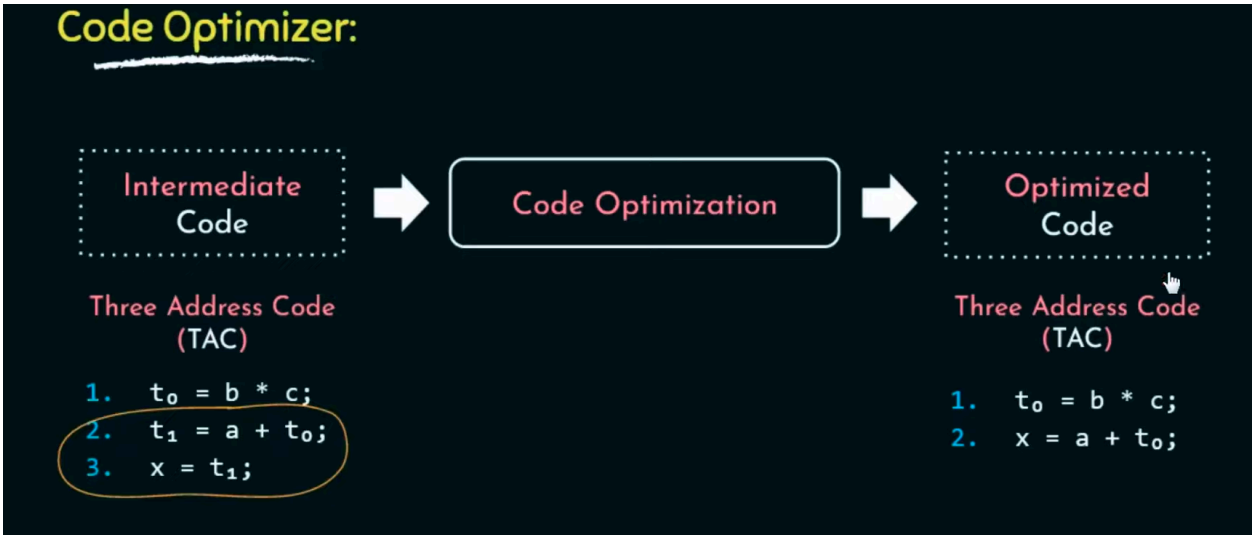
DFA

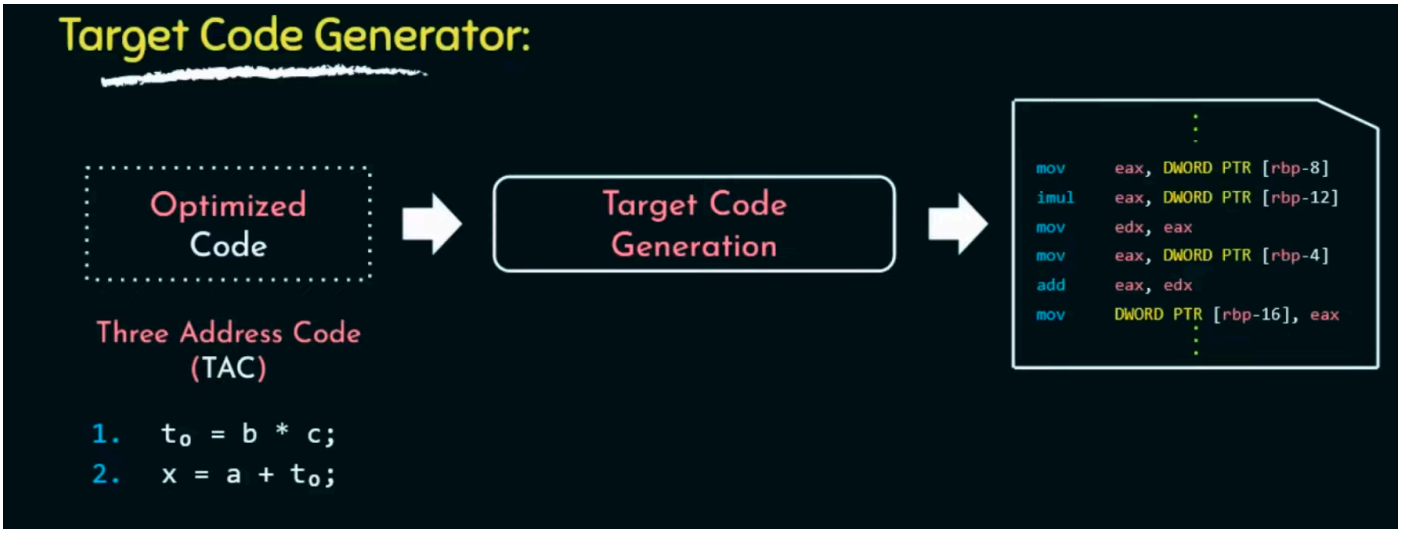




Responsible for : type checking, array bound checking , correctness of scope resolution Type mismatch errors, undeclared variables, misuse of reserved words, multiple declaration of variable in single scope, accessing out of scope variable , mismatch between actual and formal parameters & … => meaningfulness of parse tree and verifies that

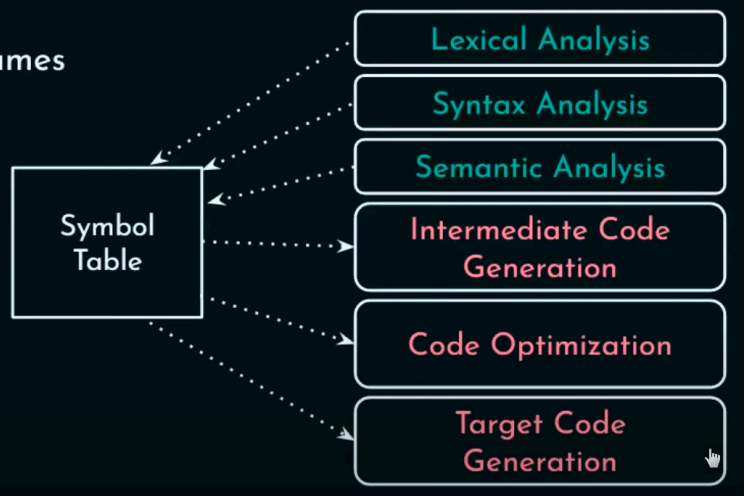



Code Optimizer:
Intermediate Code
Code Optimization
Optimized Code
Three Address Code (TAC)
1. t0 = b * c;
2. t1 = a + t0;
3. x = t1;
Three Address Code (TAC)
1. t0 = b * c;
2. x = a + t0;


Target Code Generator:
Optimized Code
Target Code Generation
Three Address Code (TAC)
1. t0 = b * c;
2. x = a + t0;
mov eax, DWORD PTR [rbp-8]
imul eax, DWORD PTR [rbp-12]
mov edx, eax
mov eax, DWORD PTR [rbp-4]
add eax, edx
mov DWORD PTR [rbp-16], eax

Symbol Table:

-- Data Structure
- Variable & Function names
- Objects
- Classes
- Interfaces



Symbol Table – Entries

```
int count;
char x[] = "NESO ACADEMY";
```

Name	Type	Size	Dimension	Line of Declaration	Line of Usage	Address
count	int	2	0	--	--	--
x	char	12	1	--	--	--

Generation — address info of identifiers.

Symbol Table – Operations

- Non-Block Structured Language:
  -- Contains single instance of the variable declaration.
  -- Operations:
  i. Insert() ii. Lookup()
- Block Structured Language:
  -- Variable declaration may happen multiple times.
  -- Operations:
  i. Insert() ii. Lookup() iii. Set() iv. Reset()

Eliminate Left Recursion

Ex1: $P \rightarrow P + Q \mid Q$

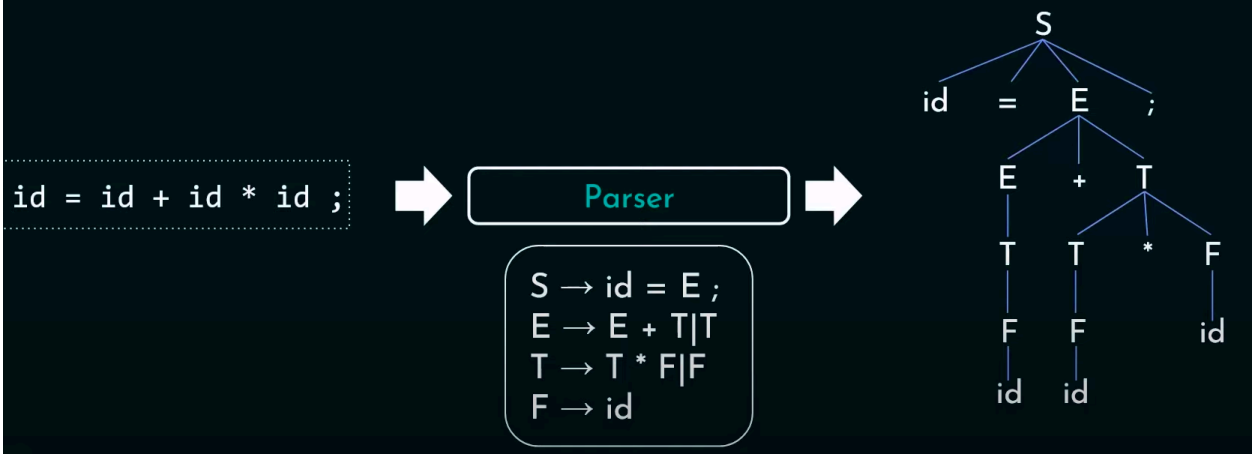
$A \rightarrow A\ \alpha \mid \beta$

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

$P \rightarrow QP'$
$P' \rightarrow +QP' \mid \varepsilon$

Parser:

A parser is a program that generates a parse tree for the given string, if the string is generated from the underlying grammar.

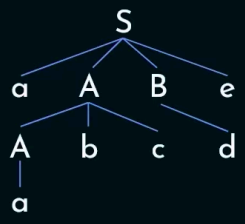


Generation of Parse Tree:

S → aABe, A → Abc | a, B → d

aabcde

Top down approach:

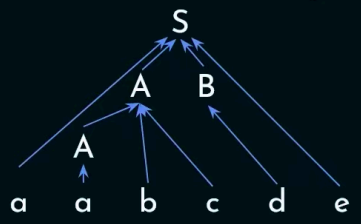


S ⇒ aABe
⇒ aAbcBe
⇒ aabcBe
⇒ aabcde

(Left most Derivation)

Decision:
Which production to use.

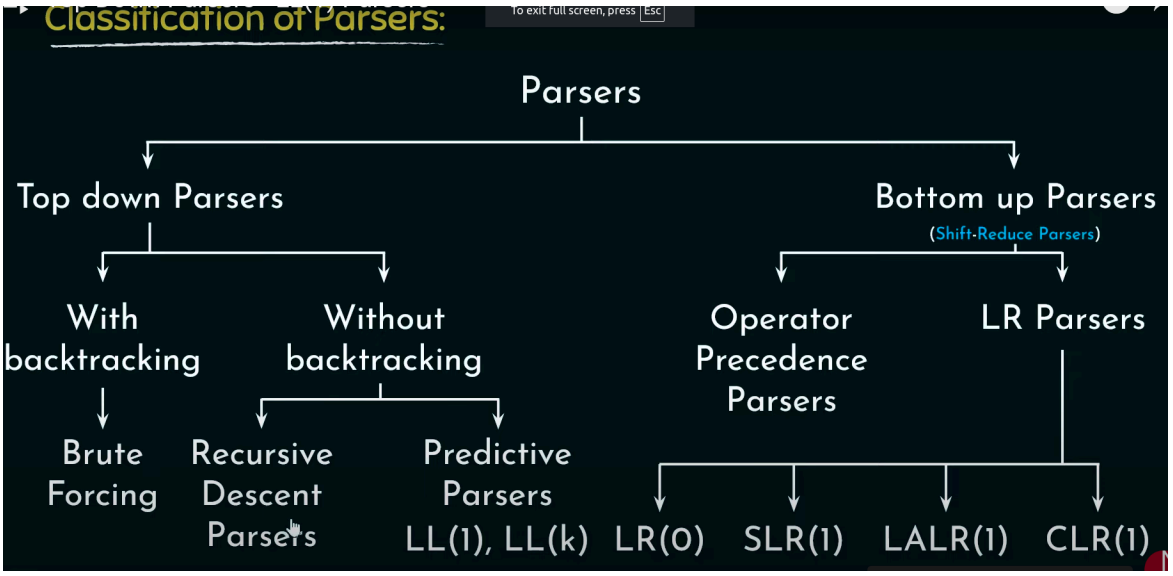
Bottom up approach:

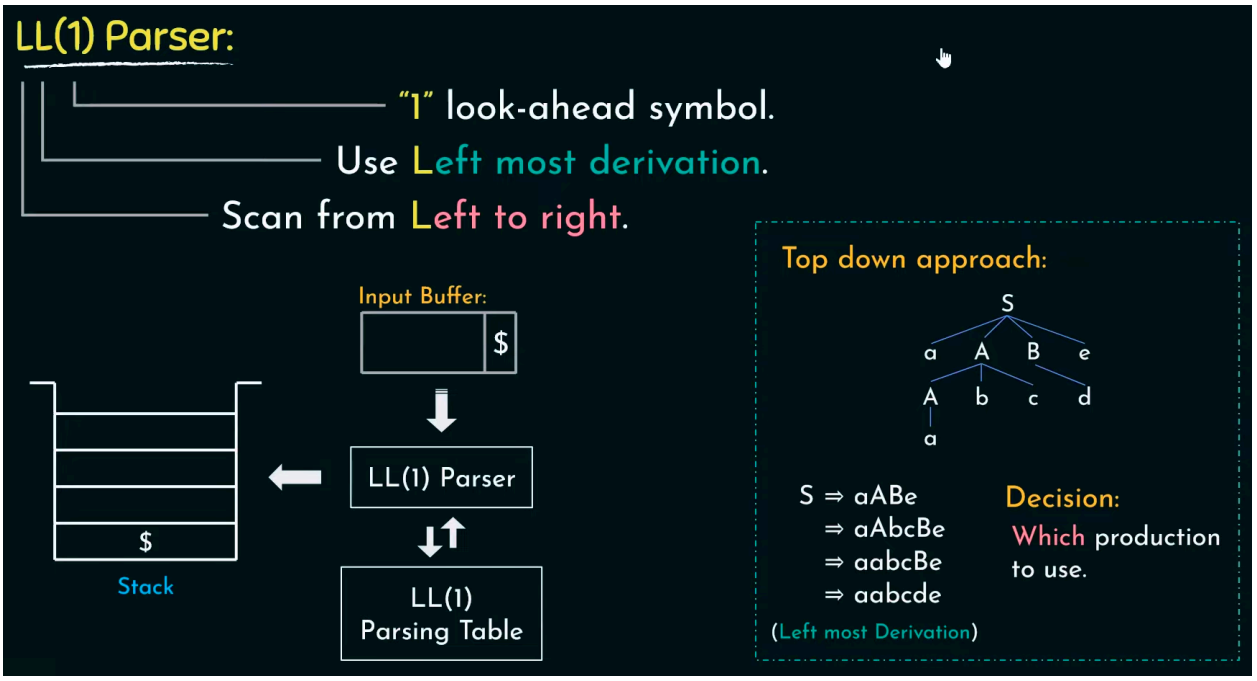


S ⇒ aABe
⇒ aAde
⇒ aAbcde
⇒ aabcde

(Right most Derivation) - In reverse.

Decision:
When to reduce.


Classification of Parsers:
To exit full screen, press Esc
Parsers
Top down Parsers
Bottom up Parsers
(Shift-Reduce Parsers)
With backtracking
Without backtracking
Operator Precedence Parsers
LR Parsers
Brute Forcing
Recursive Descent Parsers
Predictive Parsers
LL(1), LL(k)
LR(0)
SLR(1)
LALR(1)
CLR(1)


LL(1) Parser:
"1" look-ahead symbol.
Use Left most derivation.
Scan from Left to right.
Input Buffer:
$
LL(1) Parser
$
Stack
LL(1) Parsing Table
Top down approach:
S
a A B e
A b c d
a
S ⇒ aABe
⇒ aAbcBe
⇒ aabcBe
⇒ aabcde
(Left most Derivation)
Decision:
Which production to use.

LL(1) Parser:

1. FIRST():

   Given any non-terminal of a CFG, if we derive all the possible string from it, the first terminal(s) is the FIRST() of the non-terminal.

e.g.(2): S → ABC
A → a | ε
B → b
C → c

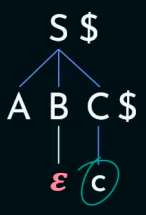
FIRST(S): { a, b }

LL(1) Parser:

2. FOLLOW():

   During the process of derivation, the terminal(s) that could follow the non-terminal are to be considered as FOLLOW() of the non-terminal.

e.g.: S → ABC
A → a
B → b | ε
C → c

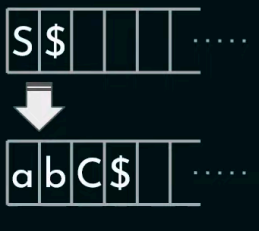


FOLLOW(S): { $ }
FOLLOW(A): { b, c }
FOLLOW(B): { c }
FOLLOW(C): { $ }

Derivation of FOLLOW:

	FIRST	FOLLOW
E → TE′	{id,(}	{$
E′ → +TE′ | ε	{+,ε}	
T → FT′	{id,(}	
T′ → *FT′ | ε	{*,ε}	
F → id | (E)	{id,(}	

S → abC



Derivation of FOLLOW:

	FIRST	FOLLOW
E → TE′	{id,(}	{$,)}
E′ → +TE′ | ε	{+,ε}	{$,)
T → FT′	{id,(}	
T′ → *FT′ | ε	{*,ε}	
F → id | (E)	{id,(}	

FOLLOW(E′) = FOLLOW(E) = { $,) }

Derivation of FOLLOW:

	FIRST	FOLLOW
E → TE′	{id,(}	{$,)}
E′ → +TE′ | ε	{+,ε}	{$,)}
T → FT′	{id,(}	
T′ → *FT′ | ε	{*,ε}	
F → id | (E)	{id,(}	

FOLLOW(E′) = FOLLOW(E′)

R.H.S. L.H.S.

Derivation of FOLLOW:

	FIRST	FOLLOW
E → TE'	{id,(}	{$,)}
E' → +TE' | ε	{+,ε}	{$,)}
T → FT'	{id,(}	{+,$,)}
T' → *FT' | ε	{*,ε}	
F → id | (E)	{id,(}	

FOLLOW(T) = FIRST(E') = { +, ε }

1. The following terminal symbol will be selected as FOLLOW.
2. The FIRST of the following non-terminal will be selected as FOLLOW.
3. If it is the right most in the RHS, the FOLLOW of the LHS will be selected.

Q1: Consider the following grammar:

P → xQRS
Q → yz | z
R → w | ε
S → y

What is FOLLOW(Q)?

(A) {R}

(B) {w}

(C) {w, y}

(D) {w, ε}

FOLLOW(Q) = {w, y}

FIRST(R) = {w, ε}

FIRST(S) = {y}

Q2: Find the FIRST and FOLLOW of all the non-terminals:

	FIRST	FOLLOW
S → ABCDE	{a, b, c}	{$}
A → a | ε	{a, ε}	{b, c}
B → b | ε	{b, ε}	{c}
C → c	{c}	{d, e, $}
D → d | ε	{d, ε}	{e, $}
E → e | ε	{e, ε}	{$}

Q3: Find the FIRST and FOLLOW of all the non-terminals:

	FIRST	FOLLOW
S → Bb | Cd	{a, b, c, d}	{$}
B → aB | ε	{a, ε}	{b}
C → cC | ε	{c, ε}	{d}

Q1: Find the FIRST and FOLLOW of all the non-terminals:

	FIRST	FOLLOW
S → aBDh	{a}	{$}
B → cC	{c}	{g, f, h}
C → bC | ε	{b, ε}	{g, f, h}
D → EF	{g, f, ε}	{h}
E → g | ε	{g, ε}	{f, h}
F → f | ε	{f, ε}	{h}

Q2: Find the FIRST and FOLLOW of all the non-terminals:

	FIRST	FOLLOW
S → ACB | CbB | Ba	{d, g, h, b, a, ε}	{$}
A → da | BC	{d, g, h, ε}	{h, g, $}
B → g | ε	{g, ε}	{$, a, h, g}
C → h | ε	{h, ε}	{g, $, b, h}

Construction of LL(1) Parsing table:

	id	(	)	*	+	$
E	E → TE′	E → TE′				
E′			E′ → ε		E′ → +TE′	E′ → ε
T	T → FT′	T → FT′				
T′			T′ → ε	T′ → *FT′	T′ → ε	T′ → ε
F	F → id	F → (E)				

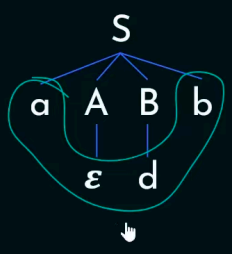
Rules:

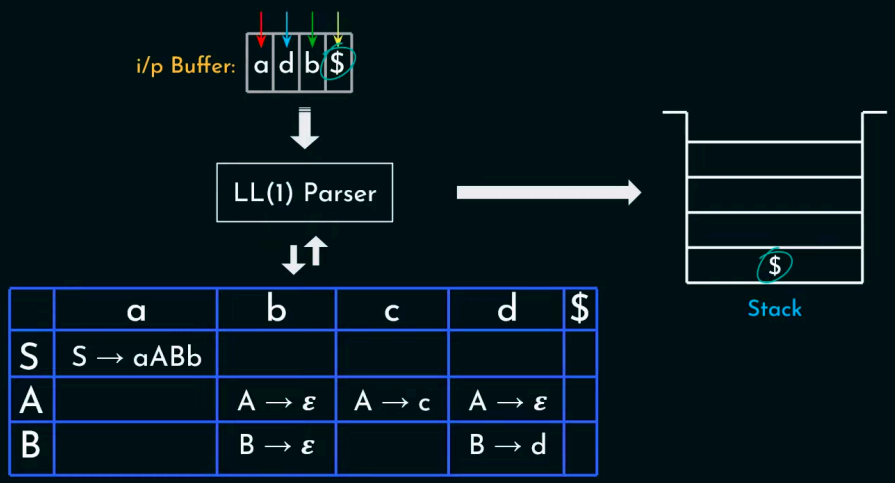
1. All the ε-productions are placed under FOLLOW sets.
2. Remaining productions are placed under the FIRSTs.

	FIRST	FOLLOW
E → TE′	{id,(}	{$,)}
E′ → +TE′ | ε	{+,ε}	{$,)}
T → FT′	{id,(}	{+,$,)}
T′ → *FT′ | ε	{*,ε}	{+,$,)}
F → id | (E)	{id,(}	{*,+,$,)}

LL(1) Parsing:

S → aABb
A → c | ε
B → d | ε





	a	b	c	d	$
S	S → aABb				
A		A → ε	A → c	A → ε	
B		B → ε		B → d	

LL(1) Parsing Table

a grammar is ll1 grammar if is parsing table for any non terminal has only one production for any terminal
If stack match buffer input -> pop stack and increase the pointer to buffer
If stack is epsilon -> pop
Else -> pop and insert in reverse mode with parsing_table[stack_top][buffer_pointer

Q1: Find out whether the following grammar is LL(1):

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$

Sol. FIRST(S): {a, b, ε}

FOLLOW(S): {$, b, a}

Not LL(1) Grammar

	a	b	$
S	S → aSbS S → ε	S → bSaS S → ε	S → ε

Q4: Find out whether the following grammar is LL(1):

$S \rightarrow A \mid a$

$A \rightarrow a$

Not LL(1) Grammar

Sol.

	FIRST	FOLLOW
S → A | a	{a}	{$}
A → a	{a}	{$}

Q5: Find out whether the following grammar is LL(1):

$S \rightarrow aB \mid \varepsilon$

$B \rightarrow bC \mid \varepsilon$

$C \rightarrow cS \mid \varepsilon$

LL(1) Grammar

Sol.

	FIRST	FOLLOW
S → aB | ε	{a, ε}	{$}
B → bC | ε	{b, ε}	{$}
C → cS | ε	{c, ε}	{$}

	a	b	c	$
S	S → aB			S → ε
B		B → bC		B → ε
C			C → cS	C → ε

Q1: Find out whether the following grammar is LL(1):

$S \rightarrow AB$
$A \rightarrow a \mid \varepsilon$
$B \rightarrow b \mid \varepsilon$

LL(1) Grammar

Sol.

	FIRST	FOLLOW
$S \rightarrow AB$	{a, b, ε}	{$}
$A \rightarrow a \mid \varepsilon$	{a, ε}	{b, $}
$B \rightarrow b \mid \varepsilon$	{b, ε}	{$}

	a	b	$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow a$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$
B		$B \rightarrow b$	$B \rightarrow \varepsilon$

Q2: Find out whether the following grammar is LL(1):

$S \rightarrow aSA \mid \varepsilon$
$A \rightarrow c \mid \varepsilon$

Not LL(1) Grammar

Sol.

	FIRST	FOLLOW
$S \rightarrow aSA \mid \varepsilon$	{a, ε}	{$, c}
$A \rightarrow c \mid \varepsilon$	{c, ε}	{$, c}

	a	c	$
S	$S \rightarrow aSA$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$
A		$A \rightarrow c$ $A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$

Q4: Find out whether the following grammar is LL(1):

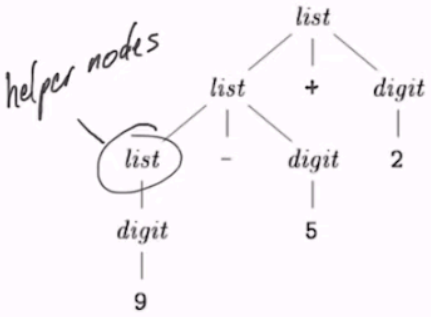
$S \rightarrow aAa \mid \varepsilon$
$A \rightarrow abS \mid \varepsilon$

Not LL(1) Grammar

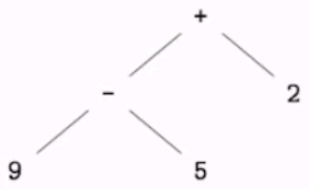
Sol.

	FIRST	FOLLOW
$S \rightarrow aAa \mid \varepsilon$	{a, ε}	{$, a}
$A \rightarrow abS \mid \varepsilon$	{a, ε}	{a}

parse tree



abstract syntax tree



9-5+2