Introduction to Compiler Design | Neso Academy

nesoacademy.org/cs/12-compiler-design/ppts/01-introduction-to-compiler-design

CHAPTER - 1

Compiler Design

Neso Academy

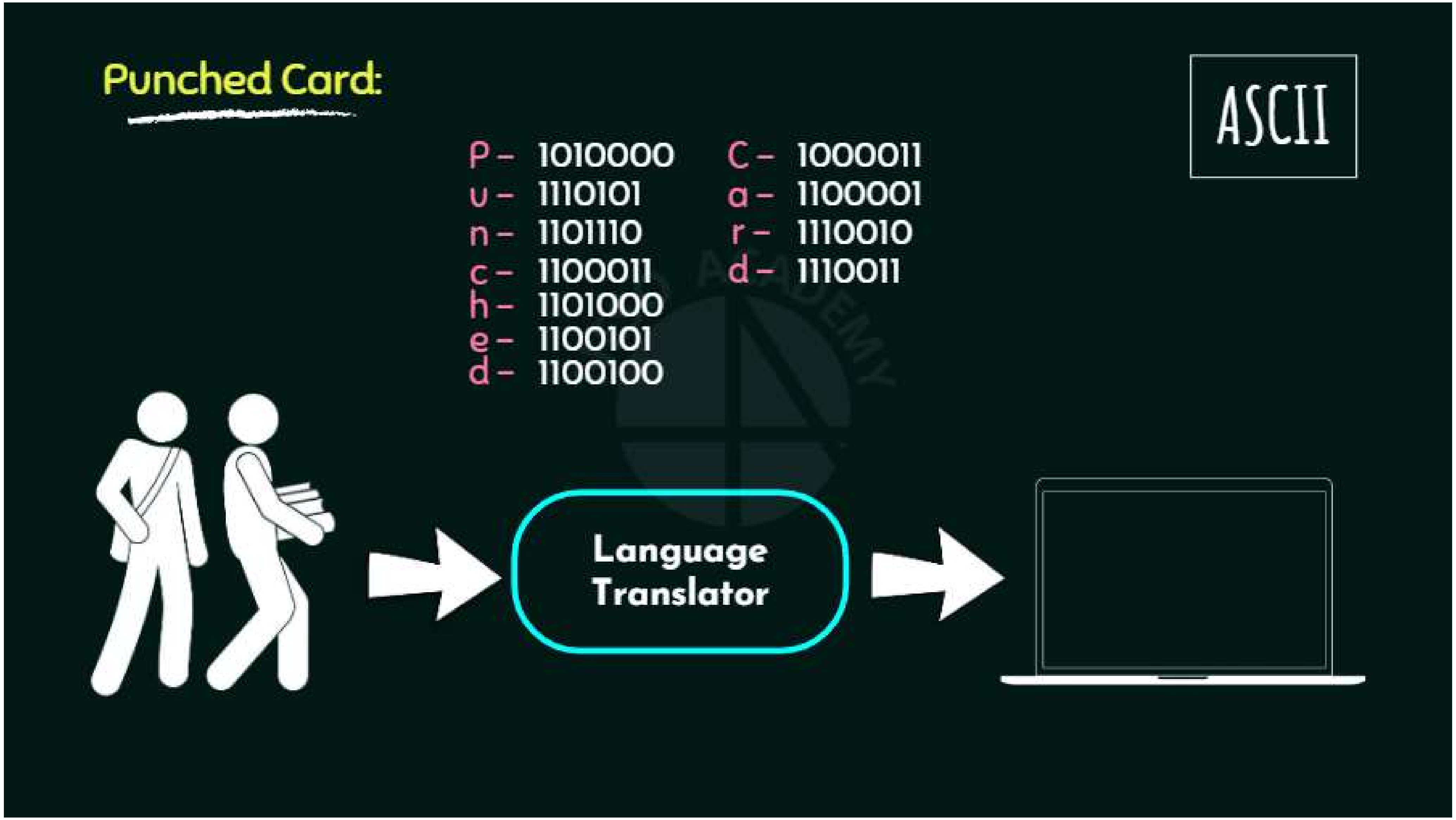

01
Paper Tape
Punched Card

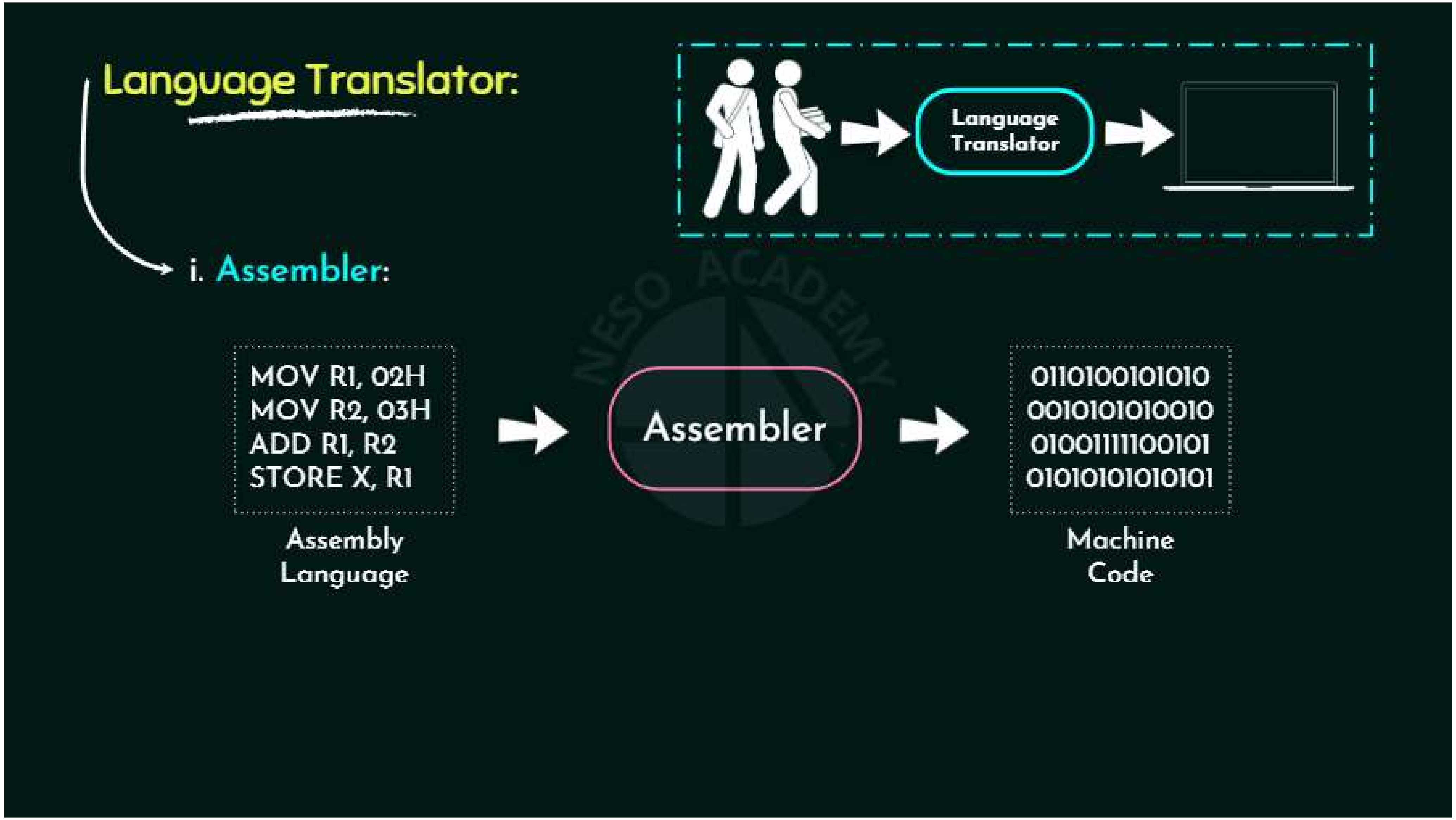
Punched Card:

ASCII

P – 1010000
u – 1110101
n – 1101110
c – 1100011
h – 1101000
e – 1100101
d – 1100100

C – 1000011
a – 1100001
r – 1110010
d – 1110011


Punched Card:
ASCII
P- 1010000
u- 1110101
n- 1101110
c- 1100011
h- 1101000
e- 1100101
d- 1100100
C- 1000011
a- 1100001
r- 1110010
d- 1110011
Language Translator


Language Translator:
Language Translator
i. Assembler:
MOV R1, 02H
MOV R2, 03H
ADD R1, R2
STORE X, R1
Assembler
0110100101010
0010101010010
0100111100101
01010101010101
Assembly Language
Machine Code

Language Translator:

i. Assembler

ii. Interpreter:

PHP, Python, Ruby, JavaScript

iii. Compiler:

C, Erlang, C++

-- Middle level Language

- Direct Memory Access through Pointers
- Bit manipulation using Bitwise Operator
- Writing Assembly Code within C Code

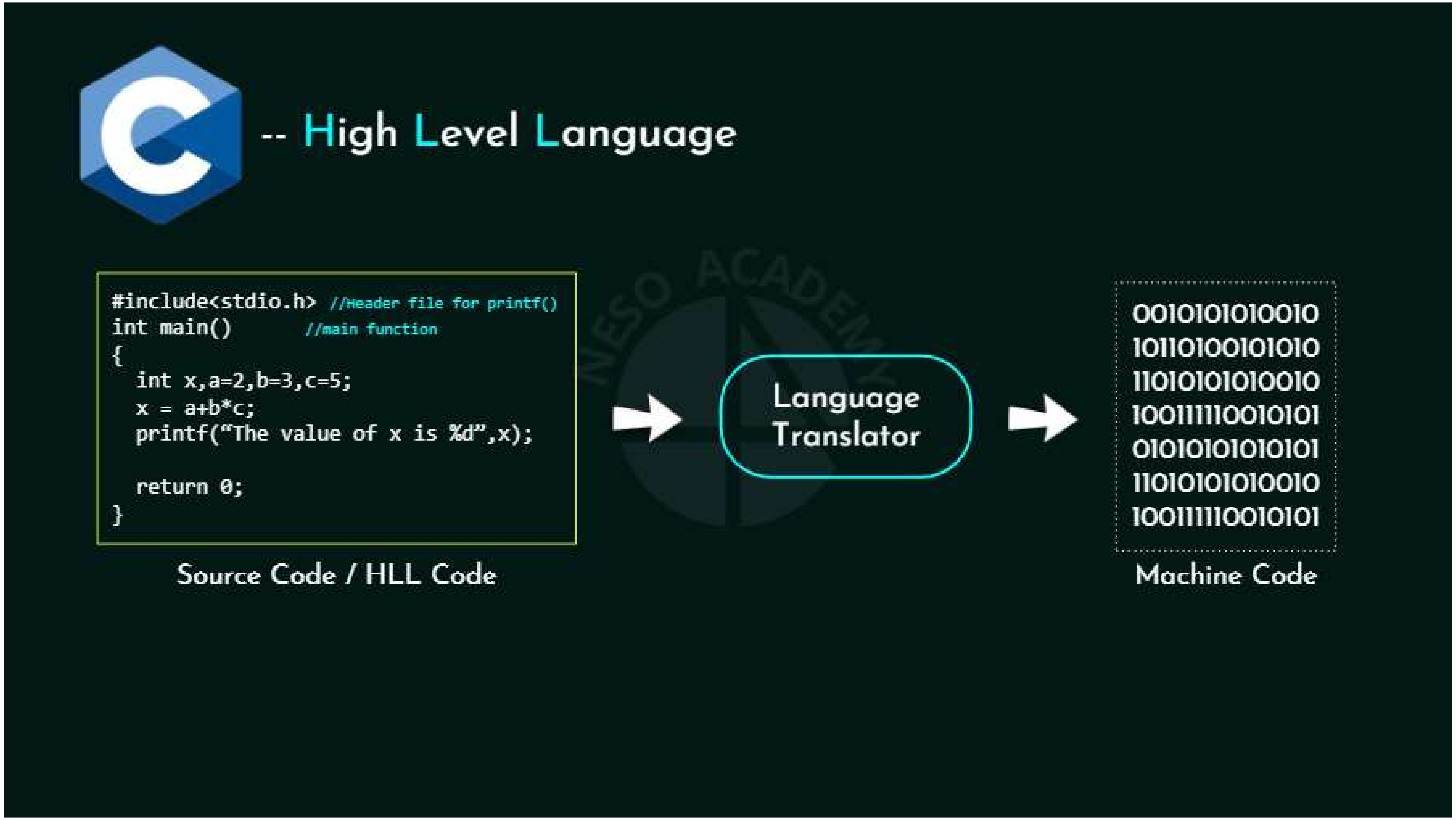
-- High Level Language

```c
#include<stdio.h> //Header file for printf()
int main()              //main function
{
     int x,a=2,b=3,c=5;
     x = a+b*c;
     printf(“The value of x is %d”,x);

     return 0;

}
```

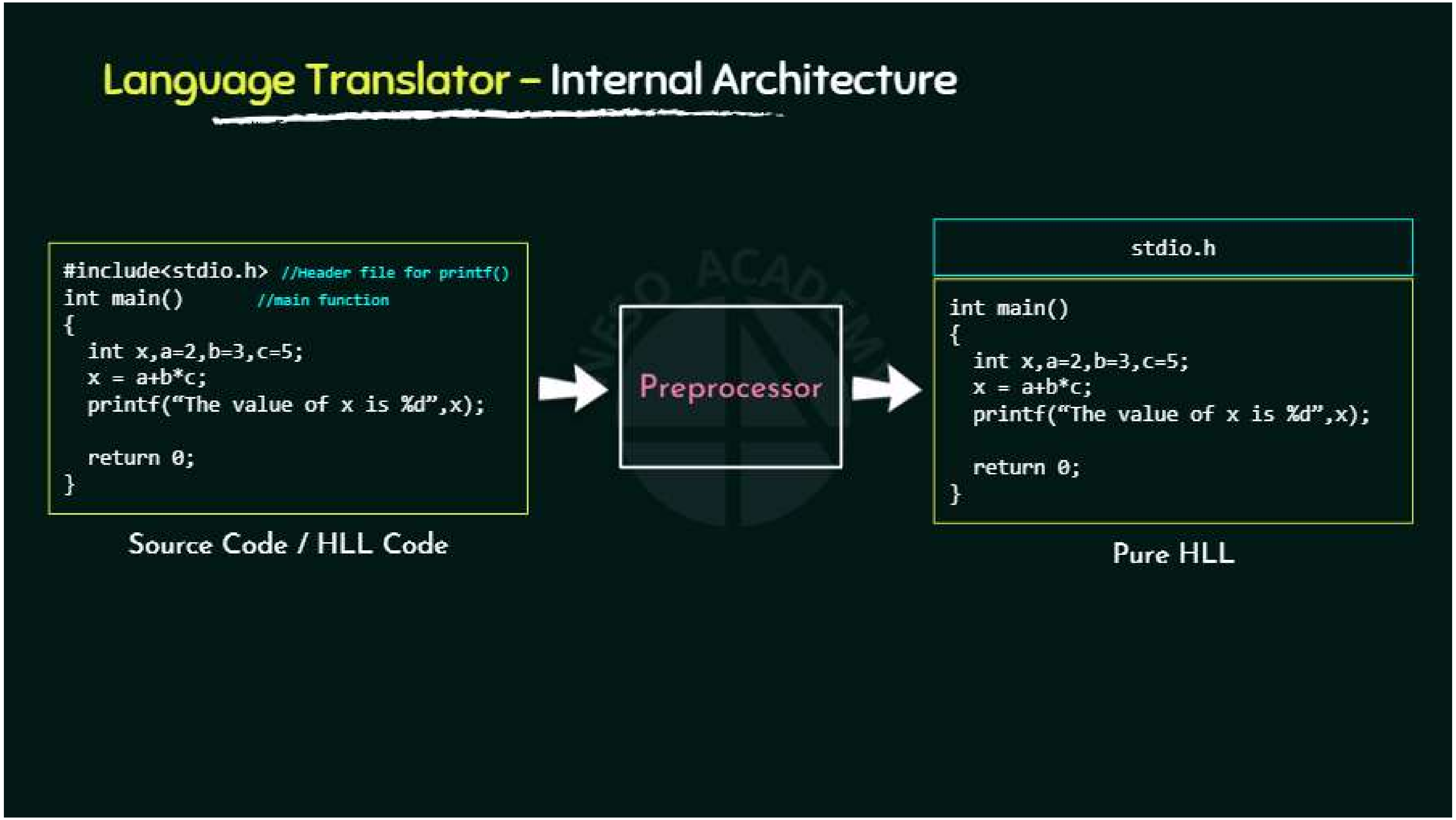
Source Code / HLL Code


-- High Level Language
#include<stdio.h> //Header file for printf()
int main() //main function
{
int x,a=2,b=3,c=5;
x = a+b*c;
printf("The value of x is %d",x);
return 0;
}
Source Code / HLL Code
Language Translator
0010101010010
1011010010101010
1101010101010010
1001111110010101
0101010101010101
1101010101010010
1001111110010101
Machine Code

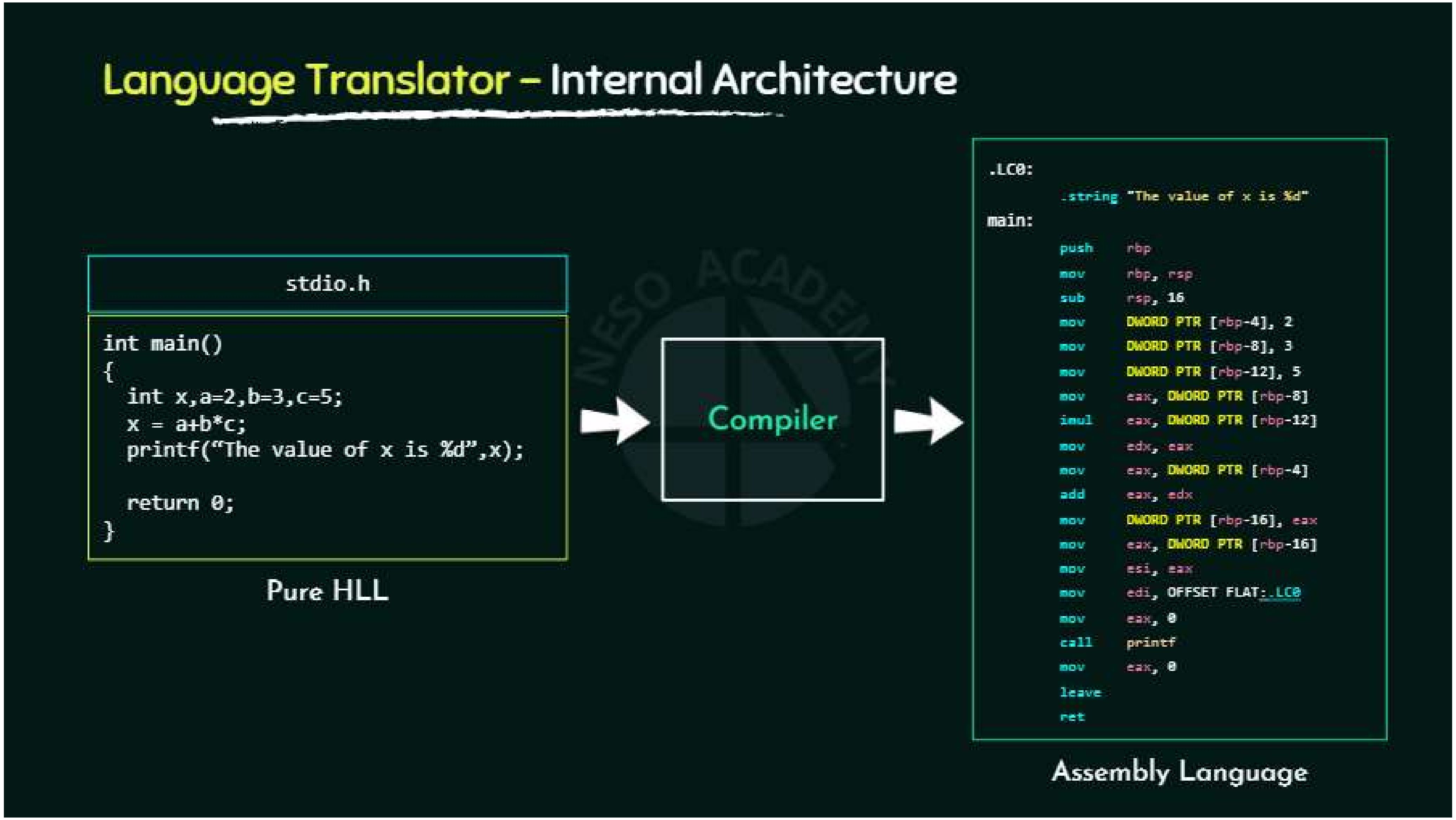
Language Translator – Internal Architecture

Language Translator

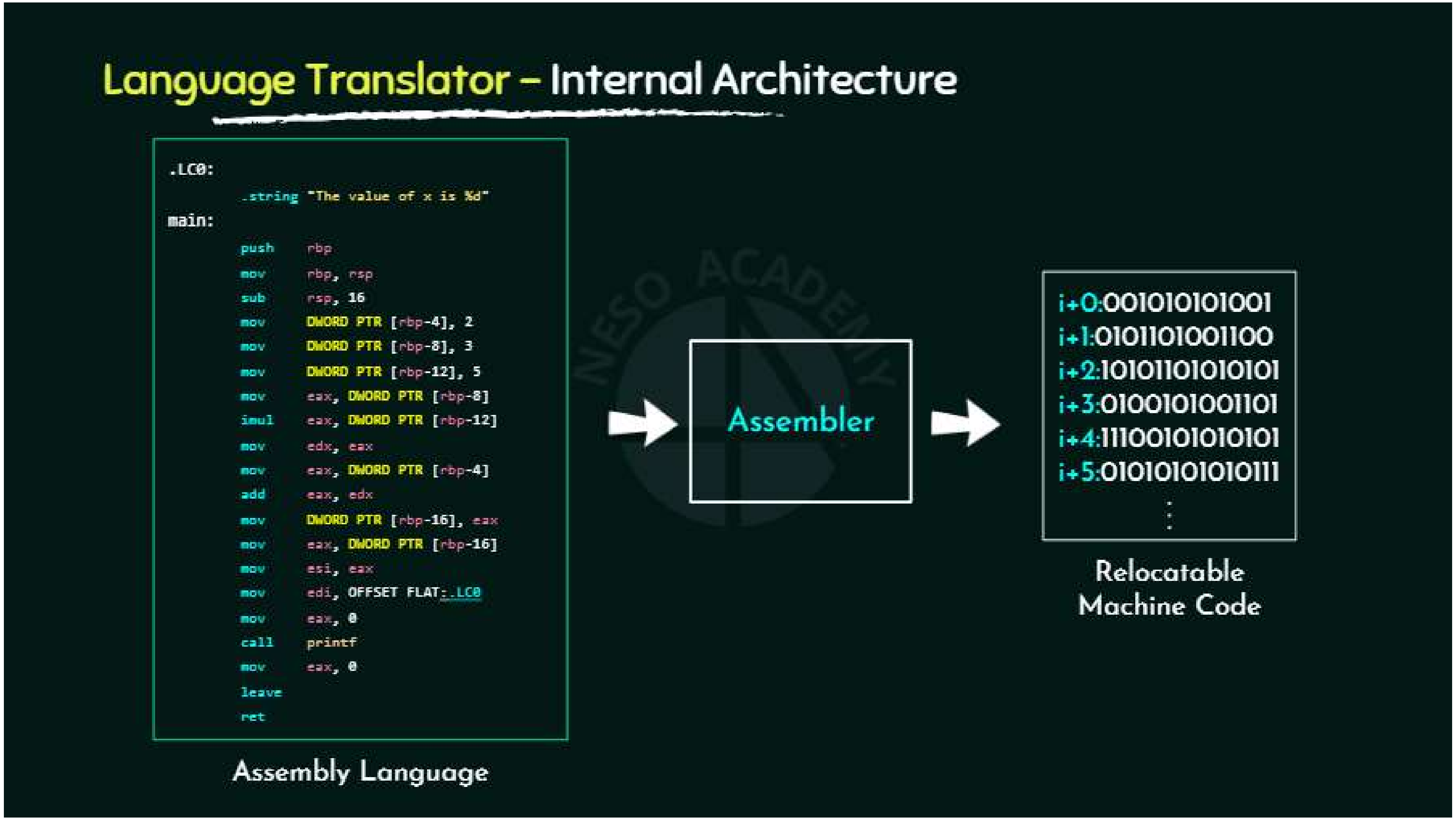
Language Translator – Internal Architecture


Language Translator – Internal Architecture
#include<stdio.h> //Header file for printf()
int main() //main function
{
int x,a=2,b=3,c=5;
x = a+b*c;
printf("The value of x is %d",x);
return 0;
}
Source Code / HLL Code
Preprocessor
stdio.h
int main()
{
int x,a=2,b=3,c=5;
x = a+b*c;
printf("The value of x is %d",x);
return 0;
}
Pure HLL

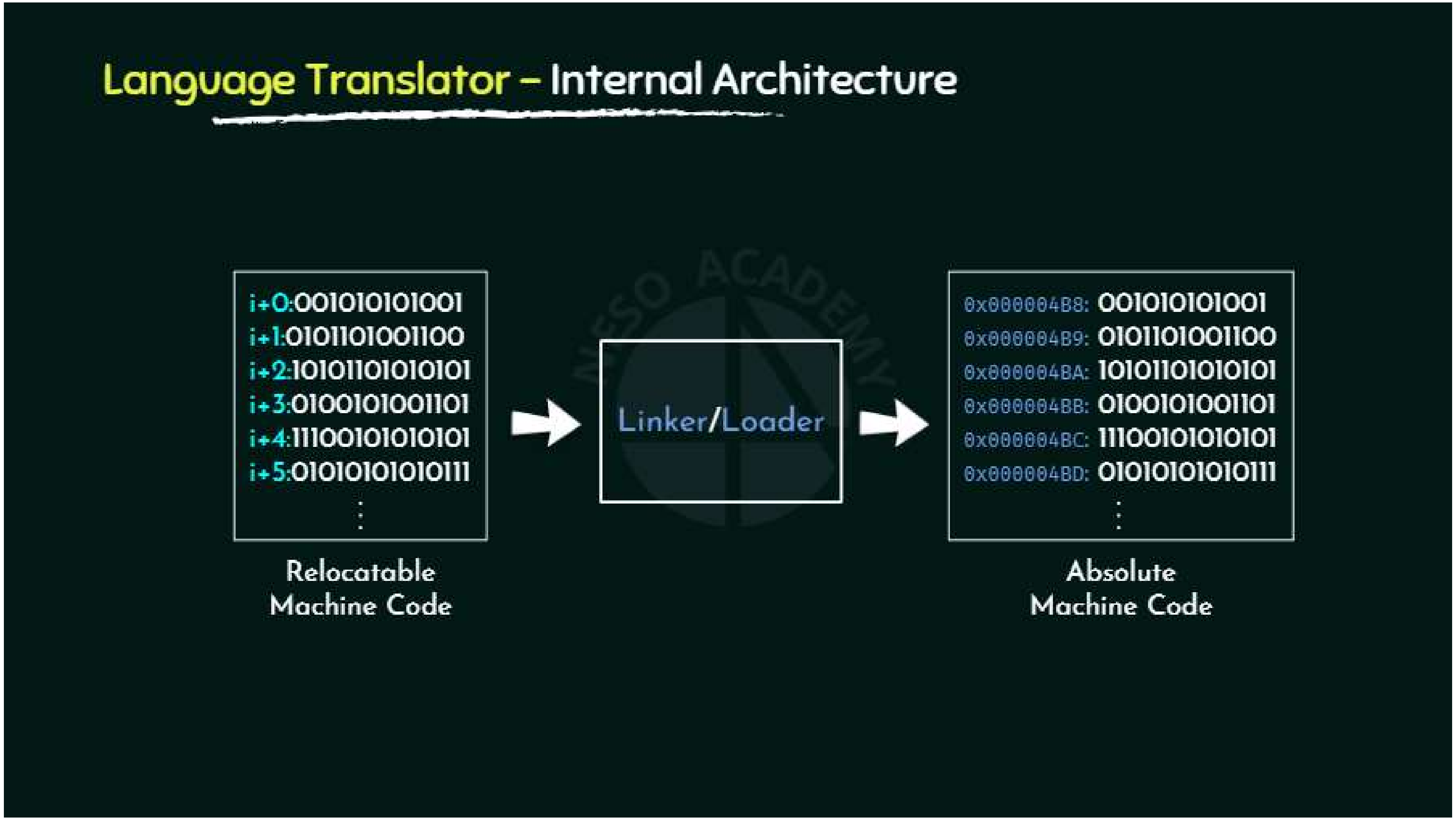
Language Translator – Internal Architecture


Language Translator – Internal Architecture
stdio.h
int main()
{
int x,a=2,b=3,c=5;
x = a+b*c;
printf("The value of x is %d",x);
return 0;
}
Pure HLL
Compiler
.LC0:
.string "The value of x is %d"
main:
push rbp
mov rbp, rsp
sub rsp, 16
mov DWORD PTR [rbp-4], 2
mov DWORD PTR [rbp-8], 3
mov DWORD PTR [rbp-12], 5
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
mov eax, DWORD PTR [rbp-16]
mov esi, eax
mov edi, OFFSET FLAT:.LC0
mov eax, 0
call printf
mov eax, 0
leave
ret
Assembly Language

Language Translator – Internal Architecture

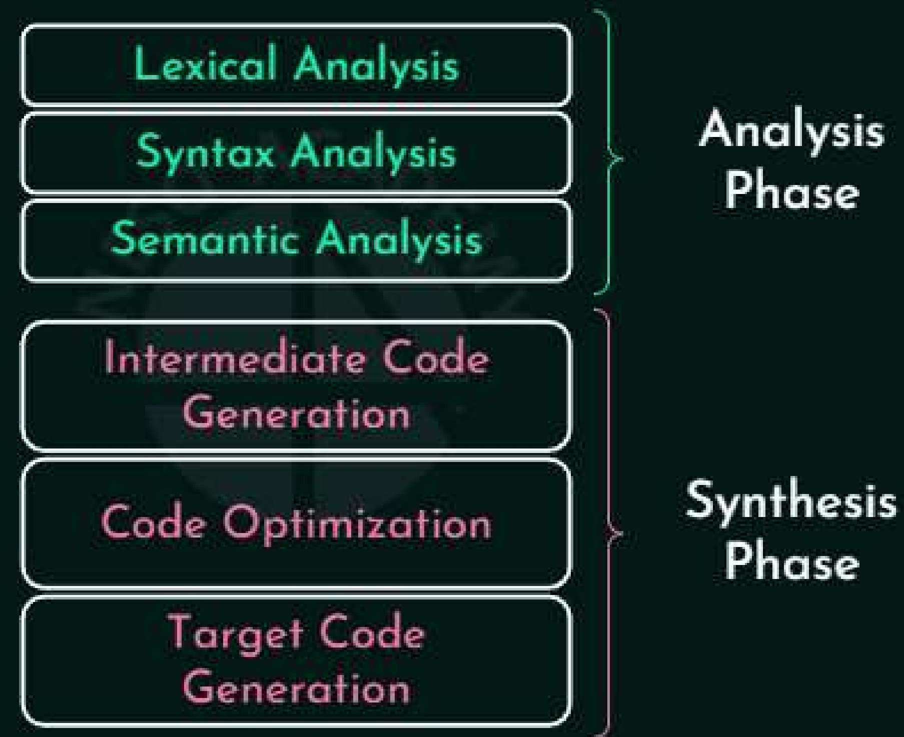

Language Translator – Internal Architecture
.LC0:
.string "The value of x is %d"
main:
push rbp
mov rbp, rsp
sub rsp, 16
mov DWORD PTR [rbp-4], 2
mov DWORD PTR [rbp-8], 3
mov DWORD PTR [rbp-12], 5
mov eax, DWORD PTR [rbp-8]
imul eax, DWORD PTR [rbp-12]
mov edx, eax
mov eax, DWORD PTR [rbp-4]
add eax, edx
mov DWORD PTR [rbp-16], eax
mov eax, DWORD PTR [rbp-16]
mov esi, eax
mov edi, OFFSET FLAT:.LC0
mov eax, 0
call printf
mov eax, 0
leave
ret
Assembly Language
Assembler
i+0:001010101001
i+1:0101101001100
i+2:10101101010101
i+3:0100101001101
i+4:11100101010101
i+5:01010101010111
Relocatable
Machine Code

Language Translator – Internal Architecture


Language Translator – Internal Architecture
i+0:001010101001
i+1:0101101001100
i+2:10101101010101
i+3:0100101001101
i+4:11100101010101
i+5:01010101010111
Relocatable Machine Code
Linker/Loader
0x000004B8: 001010101001
0x000004B9: 0101101001100
0x000004BA: 10101101010101
0x000004BB: 0100101001101
0x000004BC: 11100101010101
0x000004BD: 01010101010111
Absolute Machine Code

Language Translator – Internal Architecture

Compiler – Internal Architecture

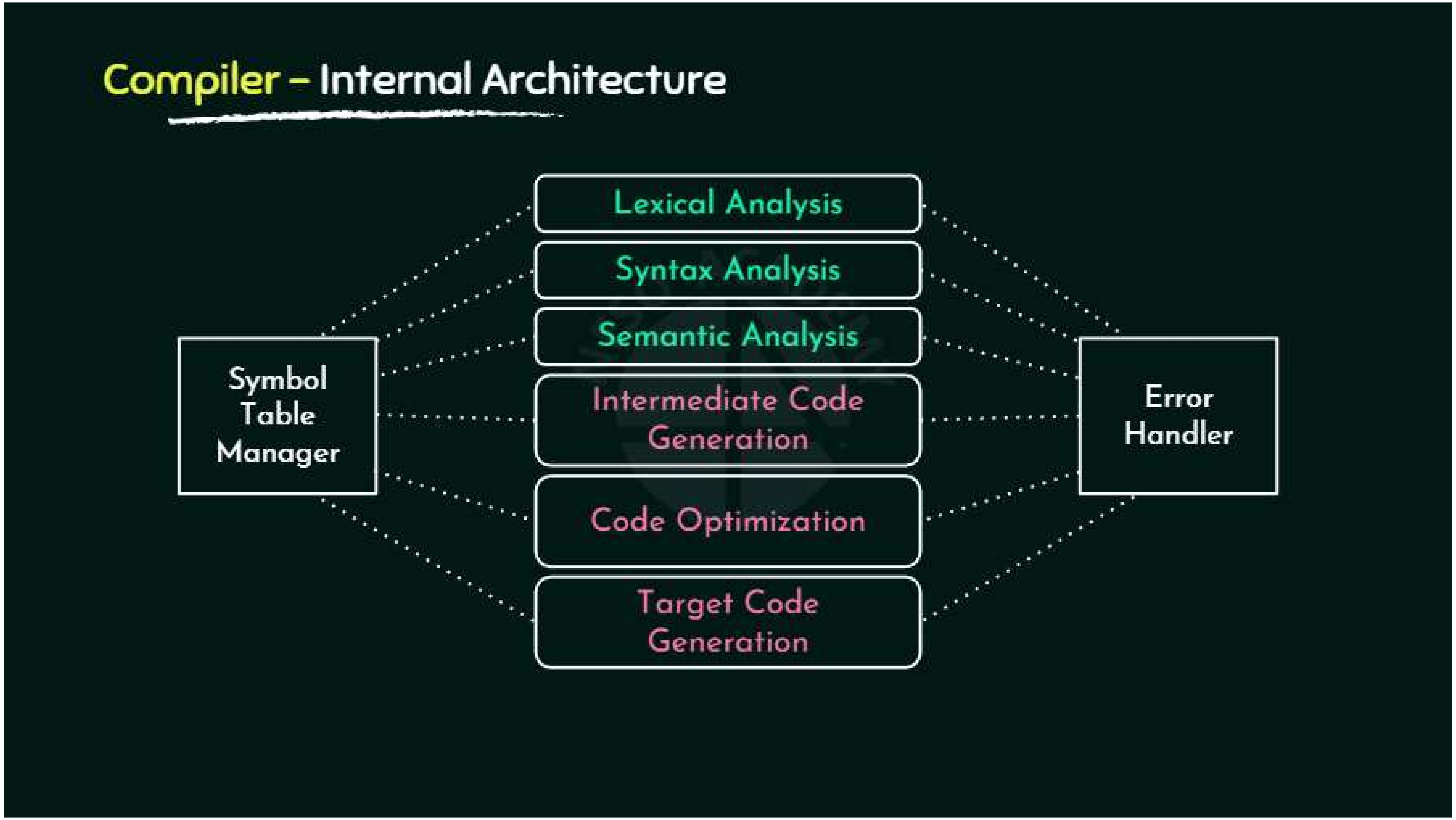


Compiler – Internal Architecture

ET VOILÀ !

Mac OS

Lexical Analysis	Front-End
Syntax Analysis	Front-End
Semantic Analysis	Front-End
Intermediate Code Generation	Front-End
Code Optimization	Back-End
Target Code Generation	Back-End


Compiler – Internal Architecture
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Target Code Generation
Symbol Table Manager
Error Handler

Syllabus:

- Introduction
- Syntax Analyzer
- Parsers - Top down
- Parsers - Bottom up
- Syntax Directed Translation Schemes
- Intermediate Code Generation
- Runtime Environment & Code Optimization

Prerequisite:

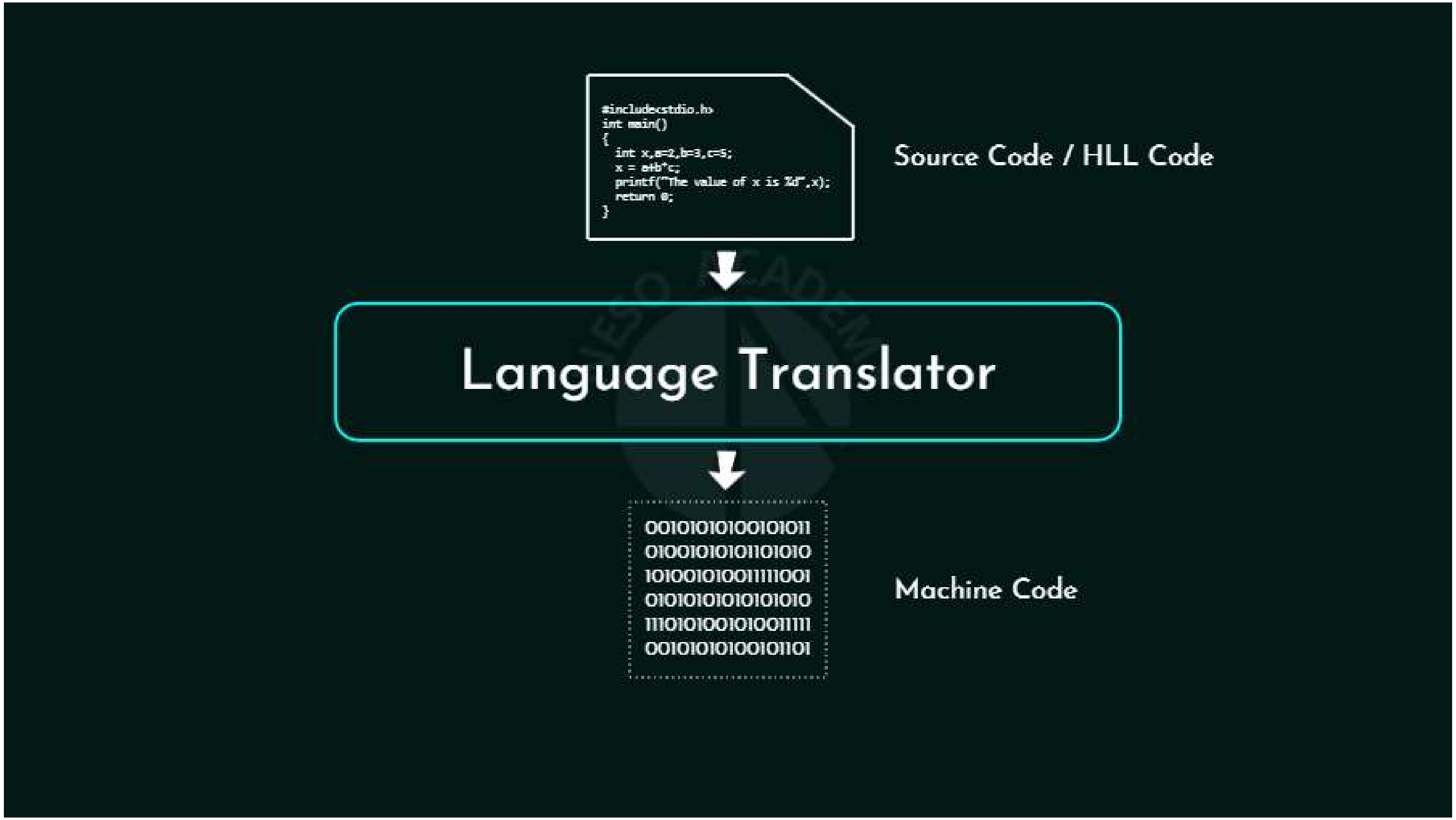
- Knowledge of Theory Of Computation.

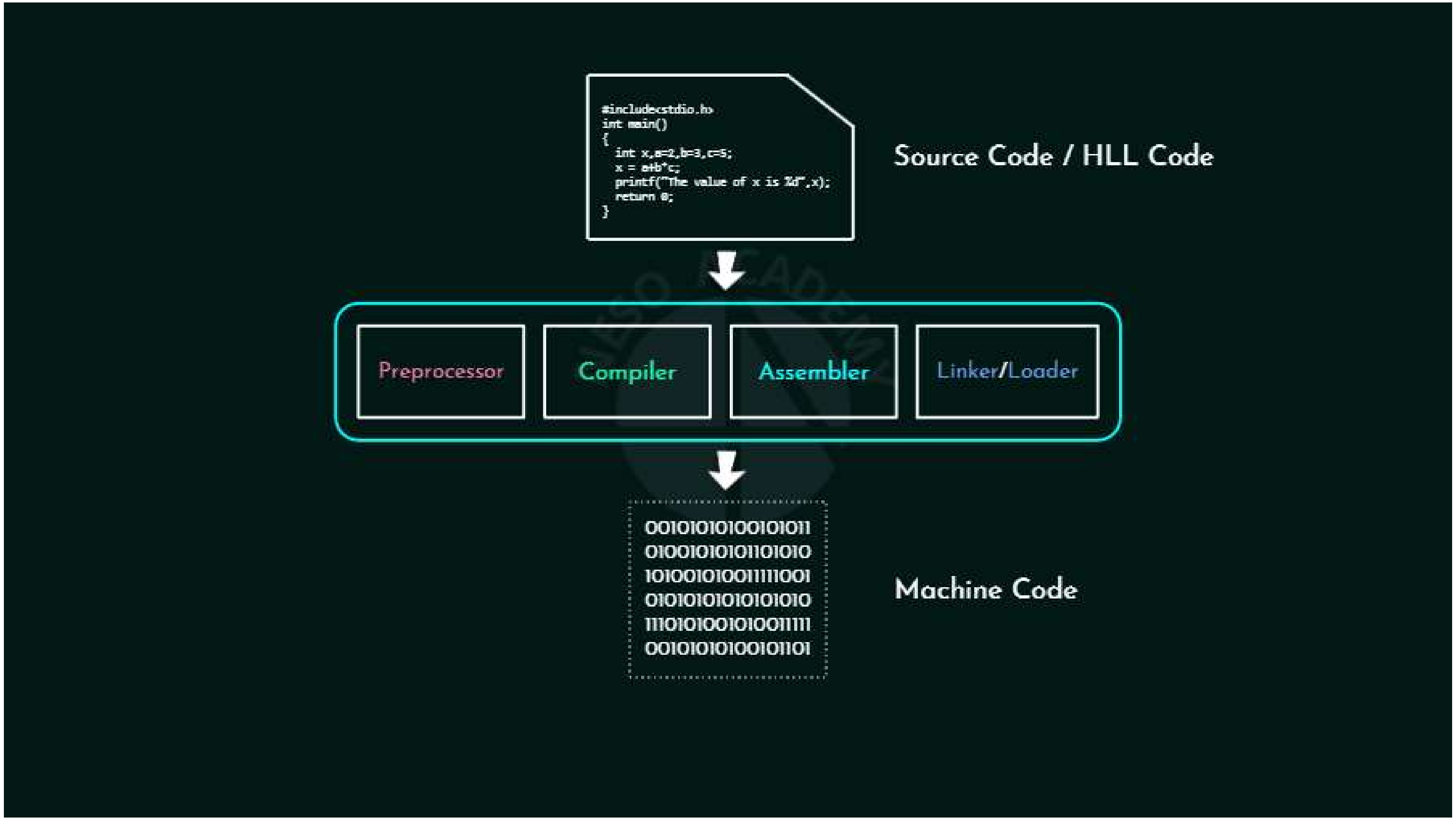
Target Audience:

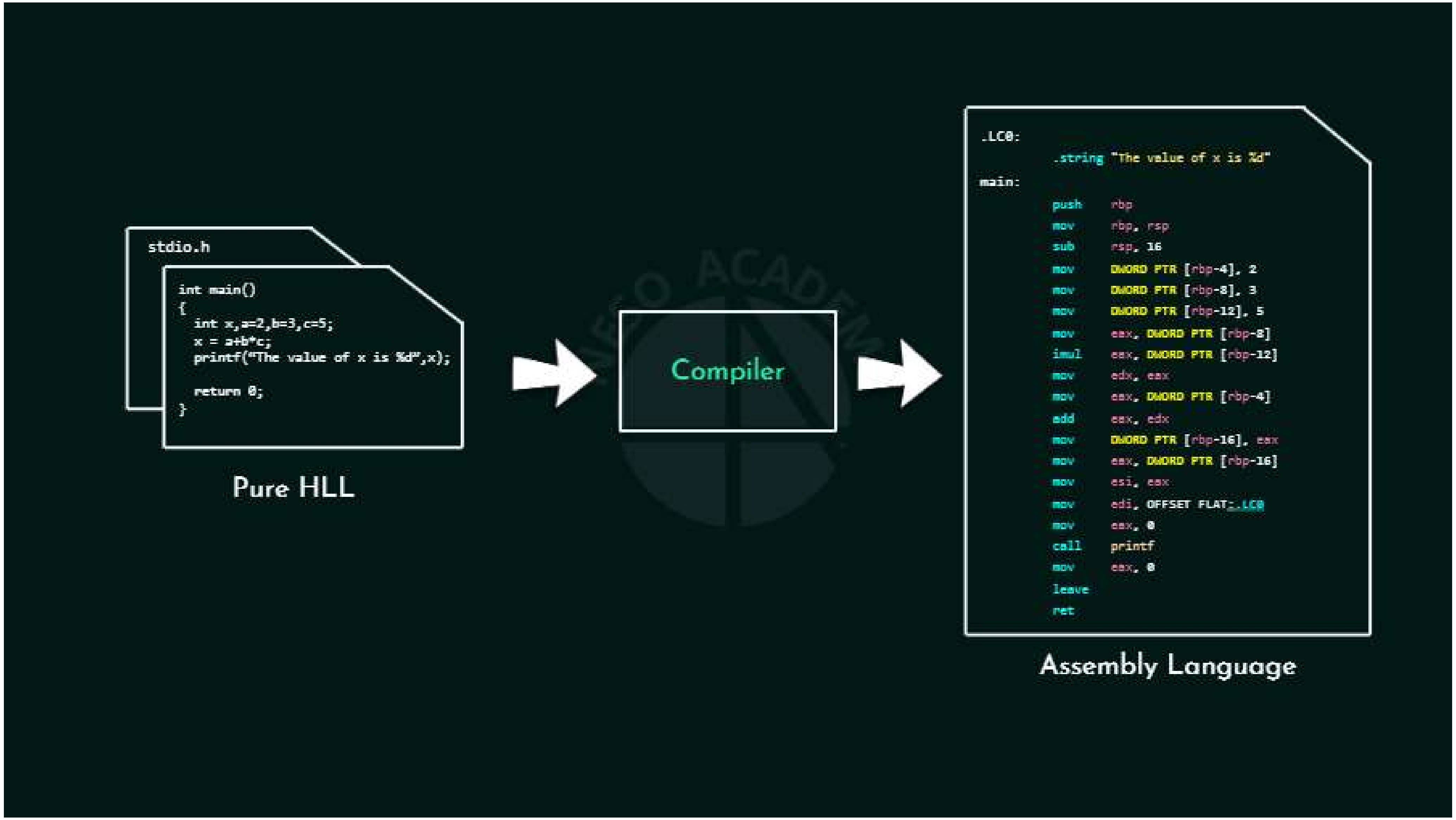
- College and University Scholars.
- Any competitive exam (GATE / NET / NIELIT etc.) aspirants.
- Any Computer Science admirer.

Outcome

- ☆ Overview of various phases of Compiler
- ☆ Tools to implement different phases


#include<stdio.h>
int main()
{
int x,a=2,b=3,c=5;
x = a+b*c;
printf("The value of x is %d",x);
return 0;
}
Source Code / HLL Code
Language Translator
0010101010010101011
0100101010110101010
1010010100111111001
0101010101010101010
1110101001010011111
0010101010010101101
Machine Code


#include<stdio.h>
int main()
{
int x,a=2,b=3,c=5;
x = a+b*c;
printf("The value of x is %d",x);
return 0;
}
Source Code / HLL Code
Preprocessor
Compiler
Assembler
Linker/Loader
001010101010010101011
010010101010110101010
101001010100111111001
010101010101010101010
111010100101010011111
001010101010010101101
Machine Code

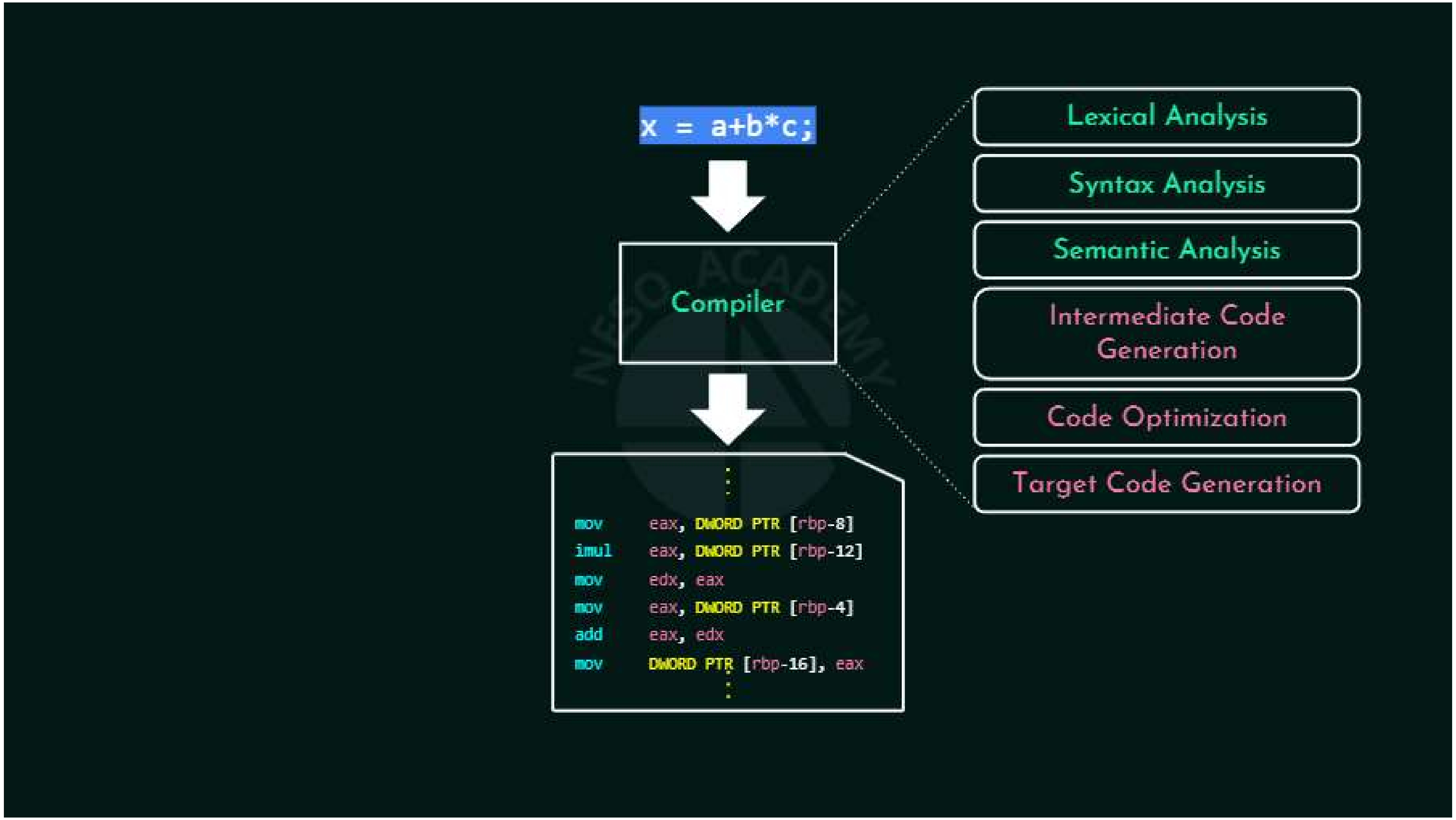

stdio.h
int main()
{
int x,a=2,b=3,c=5;
x = a+b*c;
printf("The value of x is %d",x);
return 0;
}
Pure HLL
Compiler
.LC0:
.string "The value of x is %d"
main:
push rbp
mov rbp, rsp
sub rsp, 16
mov DWORD PTR [rbp-4], 2
mov DWORD PTR [rbp-8], 3
mov DWORD PTR [rbp-12], 5
mov eax, DWORD PTR [rbp-8]
imul eax, DWORD PTR [rbp-12]
mov edx, eax
mov eax, DWORD PTR [rbp-4]
add eax, edx
mov DWORD PTR [rbp-16], eax
mov eax, DWORD PTR [rbp-16]
mov esi, eax
mov edi, OFFSET FLAT:.LC0
mov eax, 0
call printf
mov eax, 0
leave
ret
Assembly Language

stdio.h

```
int main()
{
  int x,a=2,b=3,c=5;
  x = a+b*c;
  printf("The value of x is %d",x);

  return 0;
}
```

Pure HLL


x = a+b*c;
Compiler
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
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Target Code Generation

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:

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

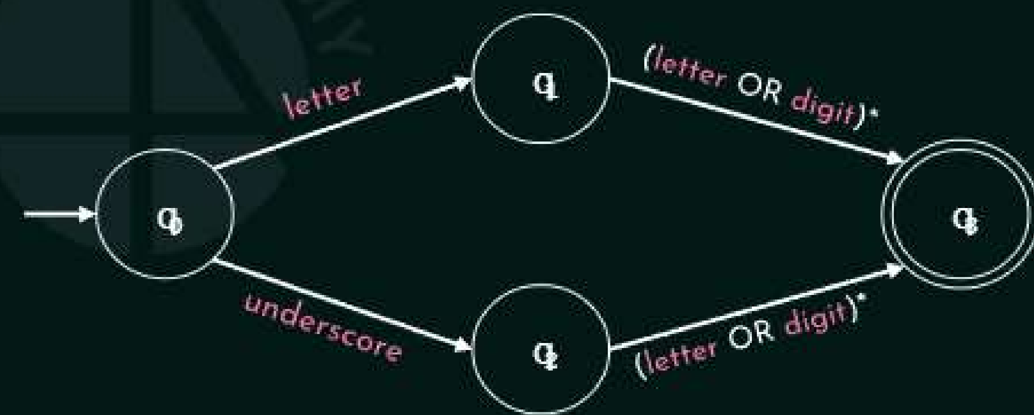
E.g. Regex for identifier:

$l(l+d)^*|\_(l+d)^*$

l : letter

d : digit

_ : underscore



Syntax Analyzer:

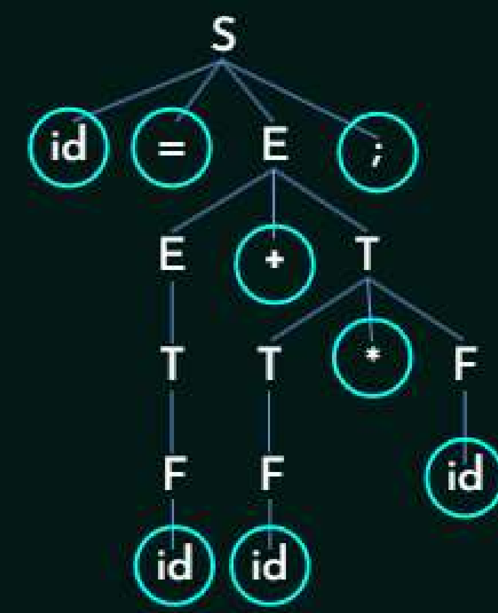
Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

```
x = a+b*c;
```

Syntax Analysis

$S \rightarrow id = E\ ;$
$E \rightarrow E + T | T$
$T \rightarrow T * F | F$
$F \rightarrow id$

```
id = id + id * id ;
```



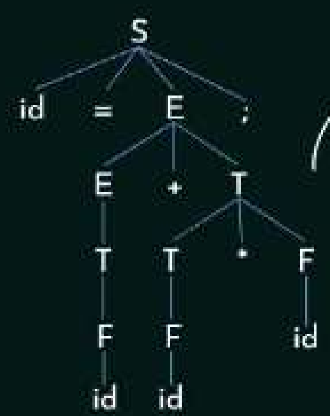
Parse Tree

Semantic Analyzer:

Parse Tree

Intermediate Code Generator:

Semantically Verified Parse Tree → Intermediate Code Generation → Intermediate Code




```
id = id + id * id ;
```

```
x = a+b*c;
```

Three Address Code (TAC)

1. t_0 = b * c;
2. t_1 = a + t_0;
3. x = t_1;

Code Optimizer:

Intermediate Code → Code Optimization → Optimized Code

Three Address Code (TAC)

1. $t_0 = b * c;$
2. $t_1 = a + t_0;$
3. $x = t_1;$

Three Address Code (TAC)

1. $t_0 = b * c;$
2. $x = a + t_0;$

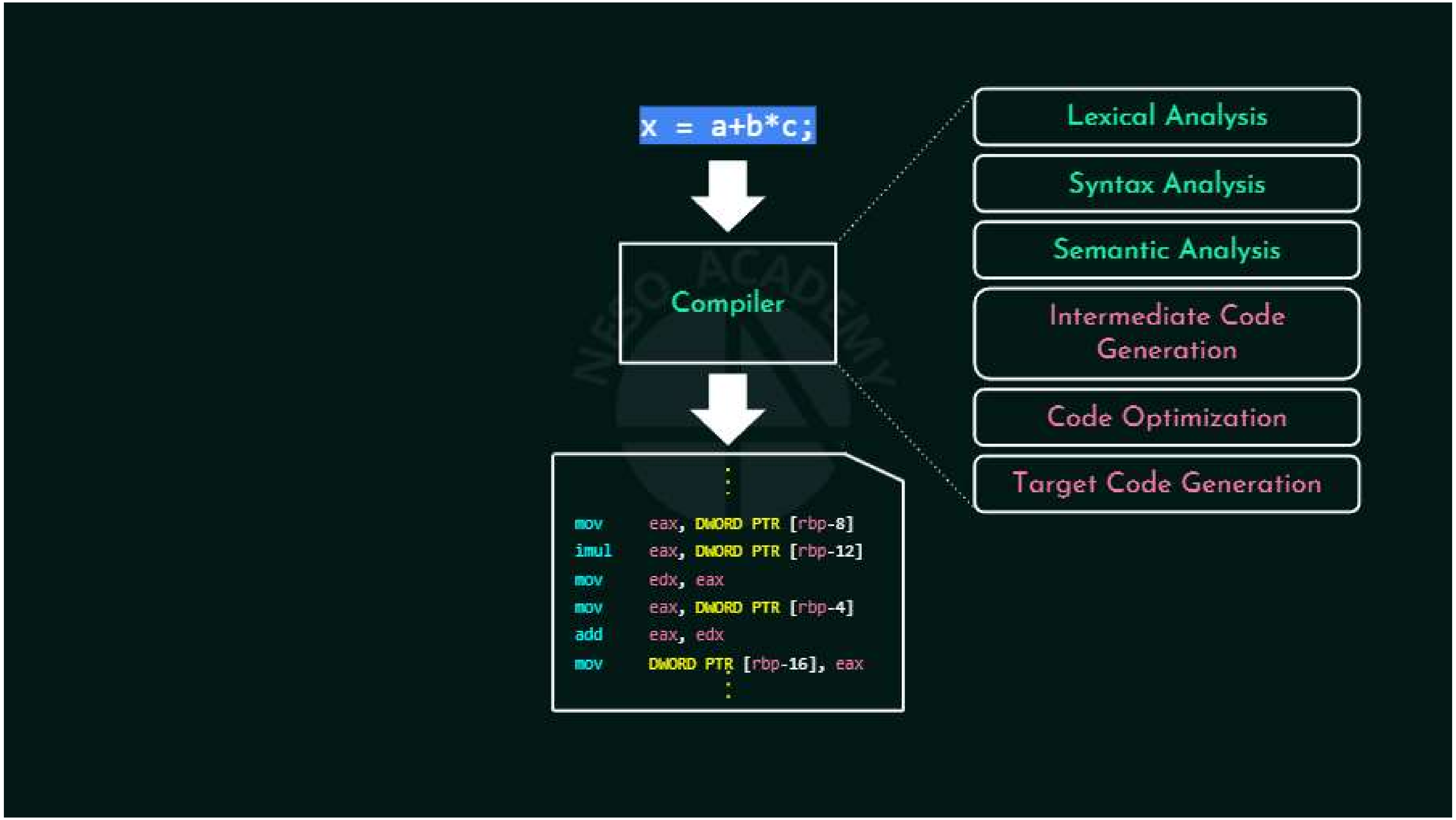
Target Code Generator:

Optimized Code → Target Code Generation →

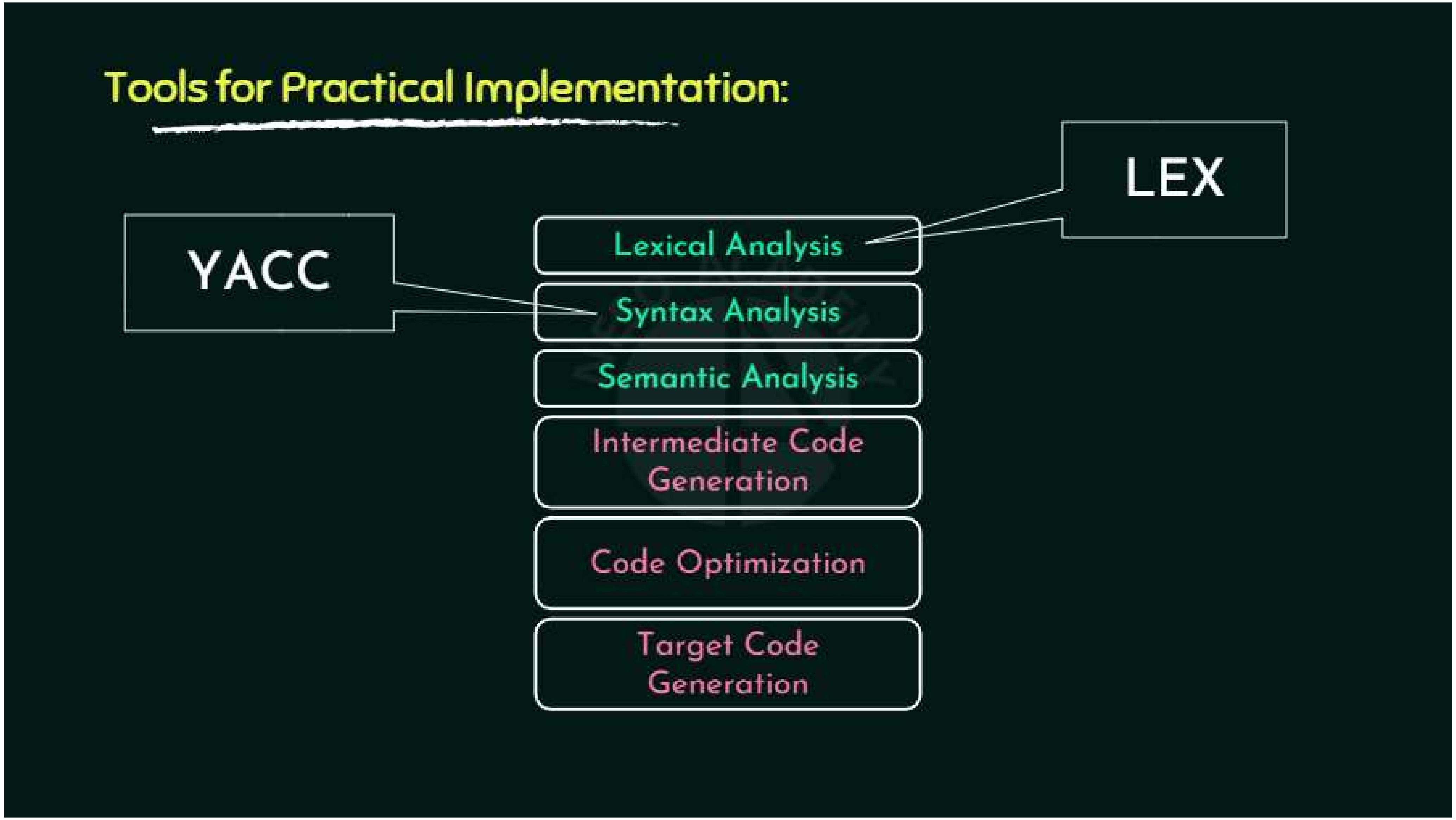
```
:
mov     eax, DWORD PTR [rbp-8]
imul    eax, DWORD PTR [rbp-12]
mov     edx, eax
mov     eax, DWORD PTR [rbp-4]
add     eax, edx
mov     DWORD PTR [rbp-16], eax
:
```

Three Address Code (TAC)

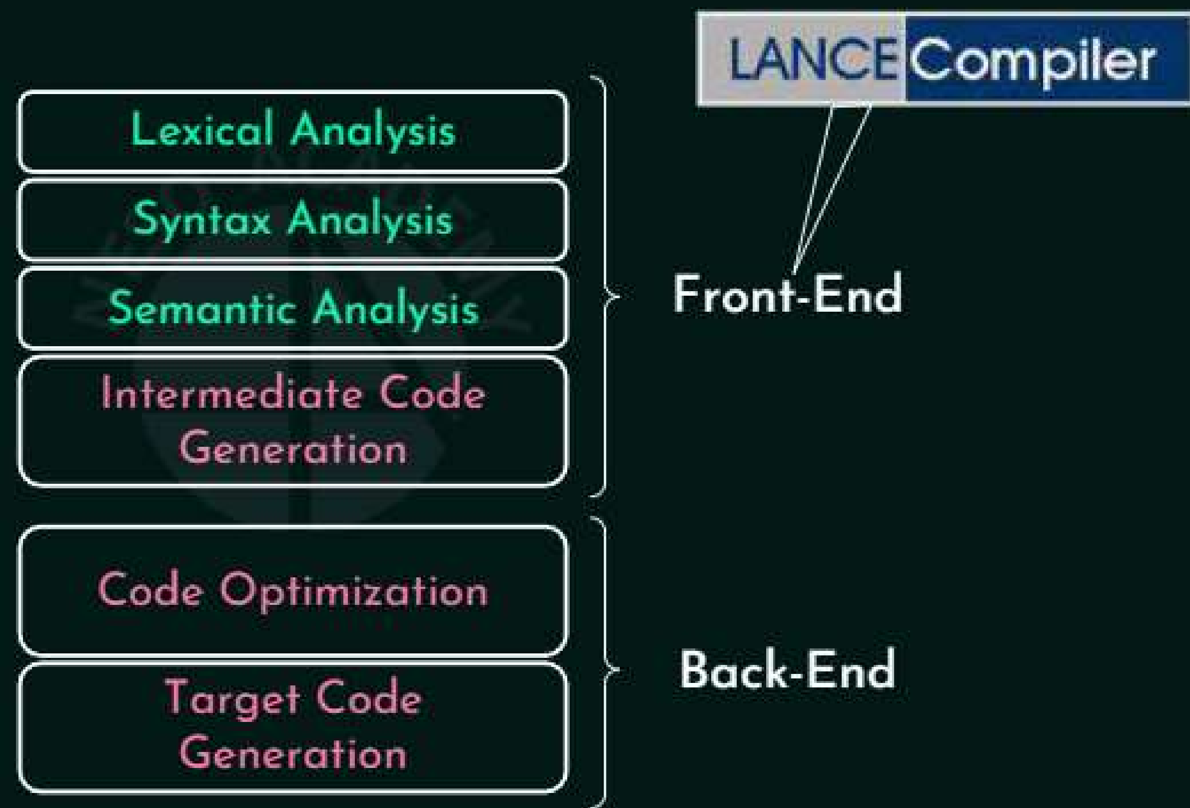
1. $t_0 = b * c;$
2. $x = a + t_0;$



x = a+b*c;
Compiler
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Target Code Generation
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


Tools for Practical Implementation:
LEX
YACC
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Target Code Generation

Tools for Practical Implementation:

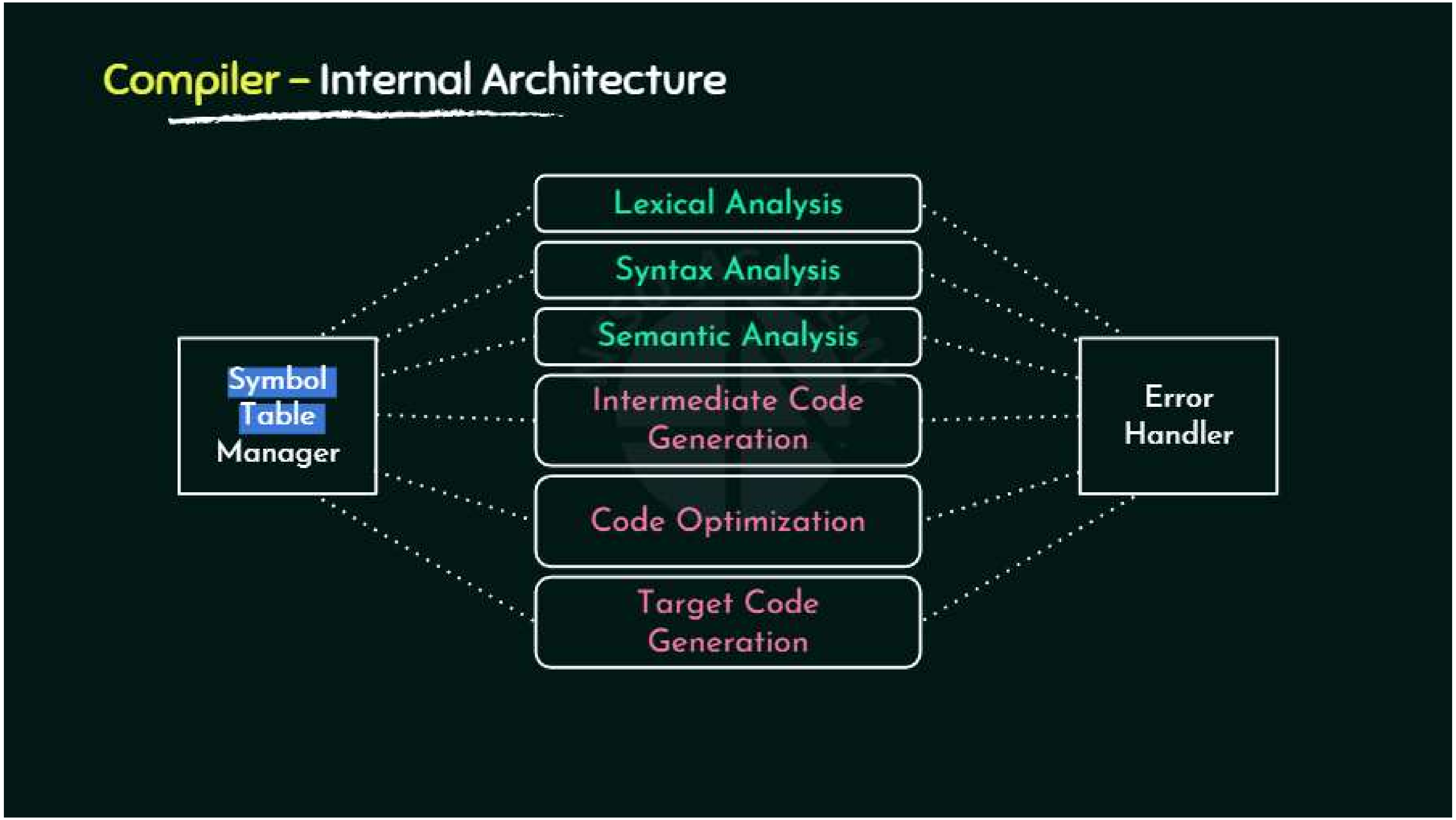


Summary

- ☆ Overview of various phases of Compiler
- ☆ Tools to implement different phases

Outcome

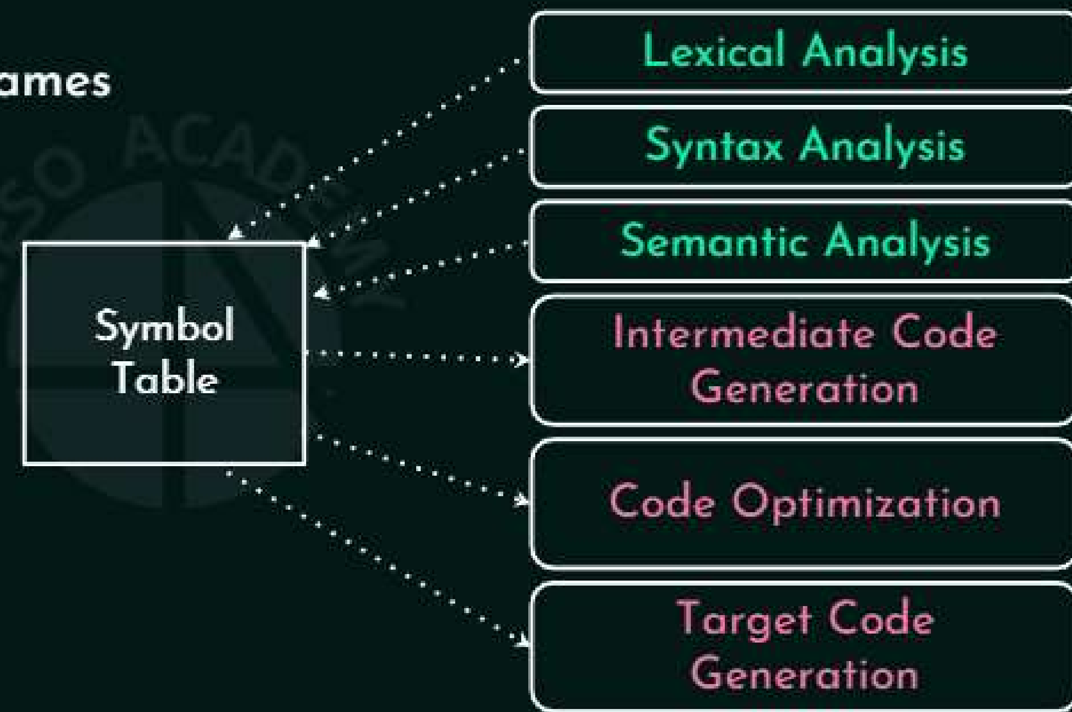
- ☆ Usage of Symbol Table by various phases
- ☆ Entries of Symbol Table
- ☆ Operations on Symbol Table


Compiler – Internal Architecture
Symbol Table Manager
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Target Code Generation
Error Handler

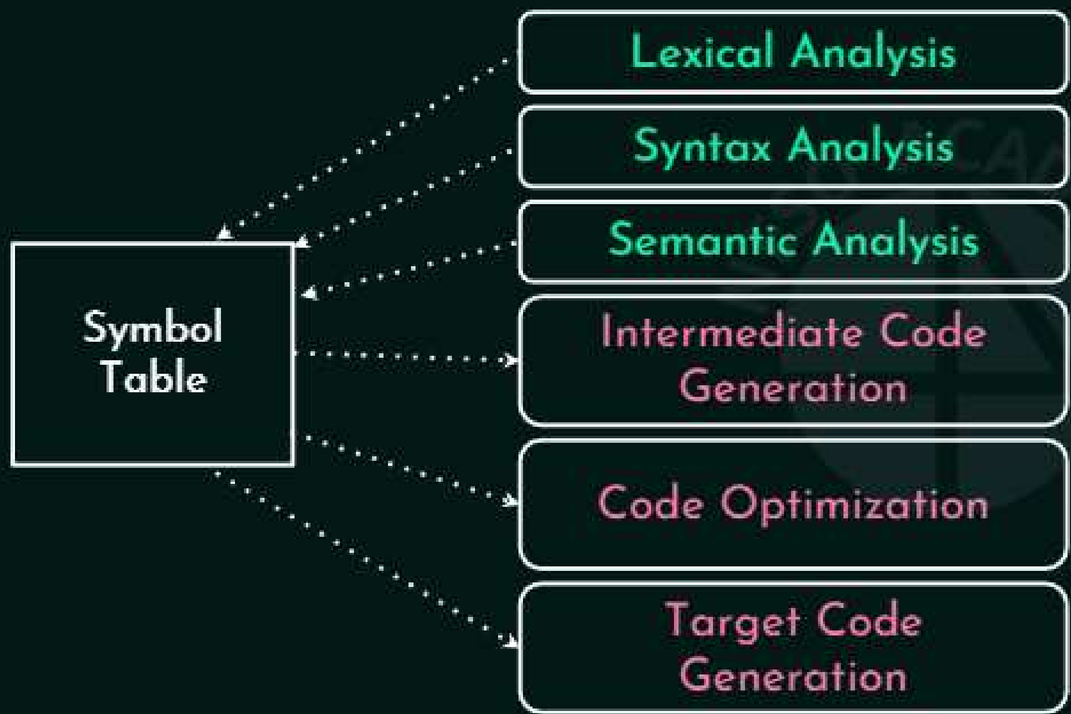
Symbol Table:

-- Data Structure

- Variable & Function names
- Objects
- Classes
- Interfaces



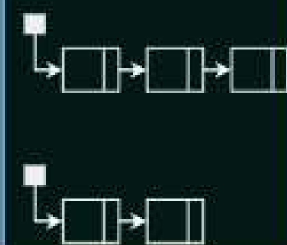
Symbol Table – Usage by Phases



Usage

- Creates entries for identifiers.
- Adds information regarding attributes.
- Using available info checks Semantics & updates the Symbol table.
- Available info helps in adding Temporary variables' information.
- Available info is used in Machine dependent optimization.
- Generates the Target code using address info of identifiers.

Symbol Table – Entries

Name	Type	Size	Dimension	Line of Declaration	Line of Usage	Address
						

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	--	--	--

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()

Summary

- ☆ Usage of Symbol Table by various phases
- ☆ Entries of Symbol Table
- ☆ Operations on Symbol Table

Outcome

- ☆ GATE 2021 question on Symbol Table
- ☆ ISRO 2016 question on Symbol Table

Q1: In the context of compilers, which of the following is/are NOT an intermediate representation of the Source program?

GATE 2021

(A) Three Address Code

(B) Abstract Syntax Tree (AST)

(C) Symbol Table

(D) Control Flow Graph (CFG)

Q2: Access time of the Symbol table will be logarithmic if it is implemented by

ISRO 2016

(A) Linear List

(B) Search Tree

(C) Hash Table

(D) None of these

Symbol Table – Various Implementations:

Implementation	Insertion Time	Lookup Time	Disadvantages
A. Linear Lists i. Ordered Lists a. Arrays ii. Unordered Lists	O(logn), O(n), O(1) O(n)		

Symbol Table – Various Implementations:

Implementation	Insertion Time	Lookup Time	Disadvantages
A. Linear Lists i. Ordered Lists a. Arrays ii. Unordered Lists	$O(\log n + \boxed{n} + 1)$ $O(n)$		

Symbol Table – Various Implementations:

Implementation	Insertion Time	Lookup Time	Disadvantages
A. Linear Lists i. Ordered Lists a. Arrays ii. Unordered Lists	O(n)		

Symbol Table – Various Implementations:

Implementation	Insertion Time	Lookup Time	Disadvantages
A. Linear Lists i. Ordered Lists a. Arrays b. Linked Lists ii. Unordered Lists	 $O(n)$ $O(n)$ $O(1)$	 $O(\log n)$ $O(n)$ $O(n)$	i. For Ordered Lists, every insertion is preceded by lookup operation. ii. Access time is directly proportional to table size.
B. Search Tree	$O(\log_m n)$	$O(\log_m n)$	Always needs to be balanced.
C. Hash Table	$O(1)$	$O(1)$	Too many collisions increases the Time complexity to $O(n)$.

Q2: Access time of the Symbol table will be logarithmic if it is implemented by

ISRO 2016

(A) Linear List

$\boxed{\text{(B) Search Tree}}$

(C) Hash Table

(D) None of these

Summary

- ☆ GATE 2021 question on Symbol Table
- ☆ ISRO 2016 question on Symbol Table

Outcome

☆ Working principle of Lexical Analyzer

Lexical Analyzer:

```
x = a+b*c;
```

Lexical Analysis

Recognizes tokens using Regexs.
E.g. Regex for identifier:

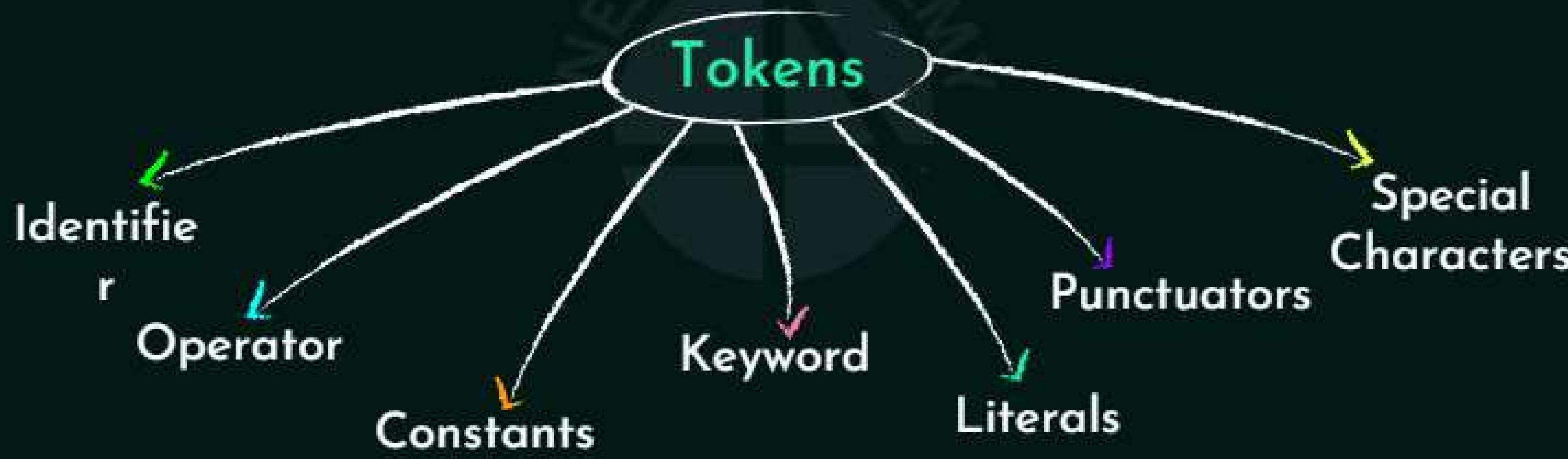
l(l+d)*|_(l+d)*

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.



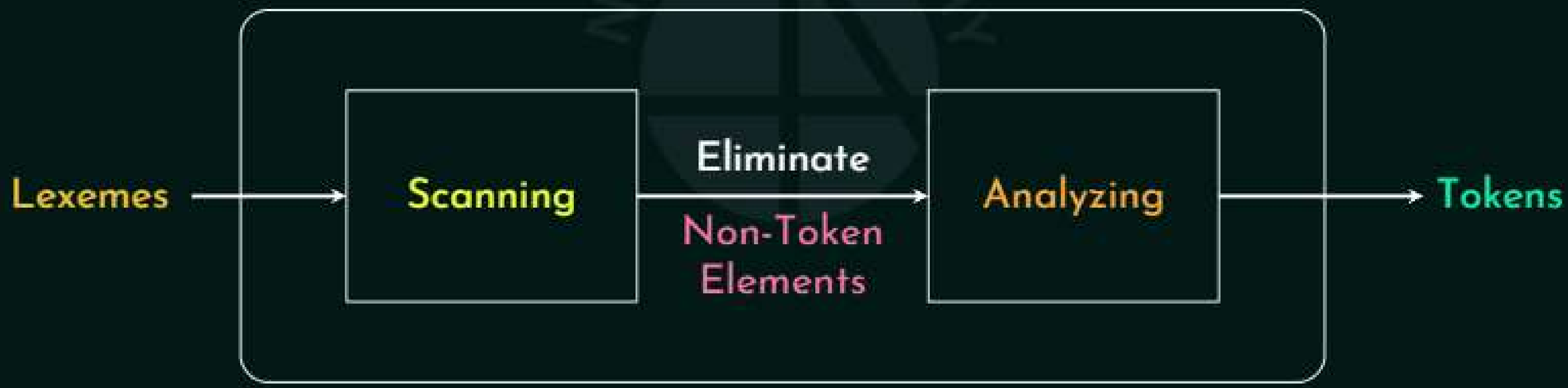
Lexical Analyzer:

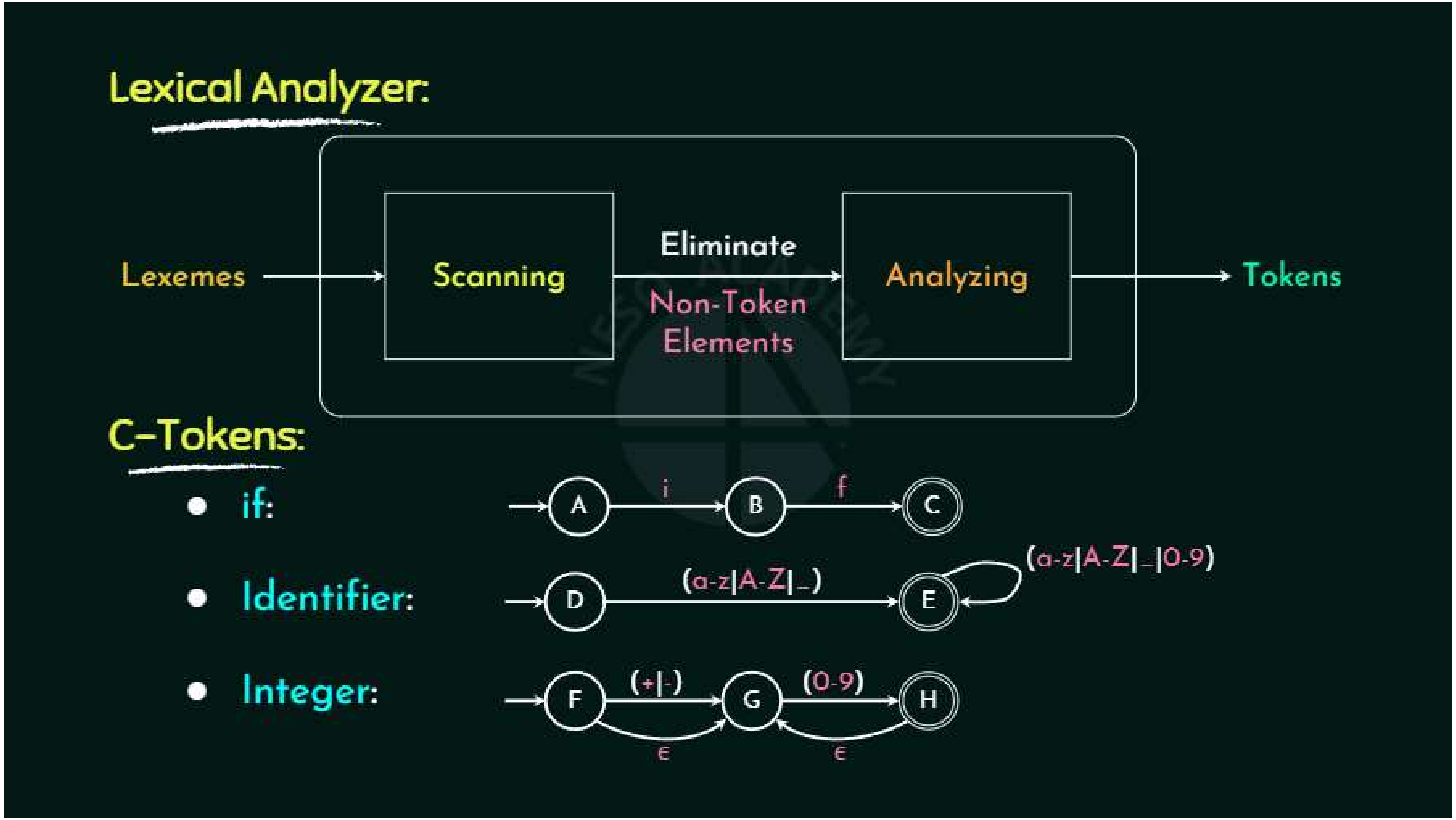
- Scans the Pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens.

Lexical Analyzer

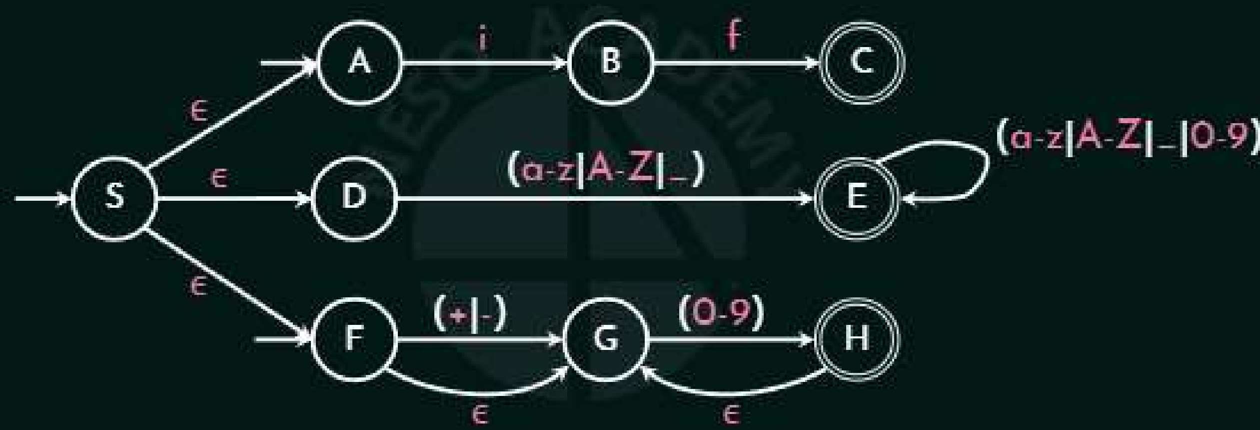
Lexical Analyzer:

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

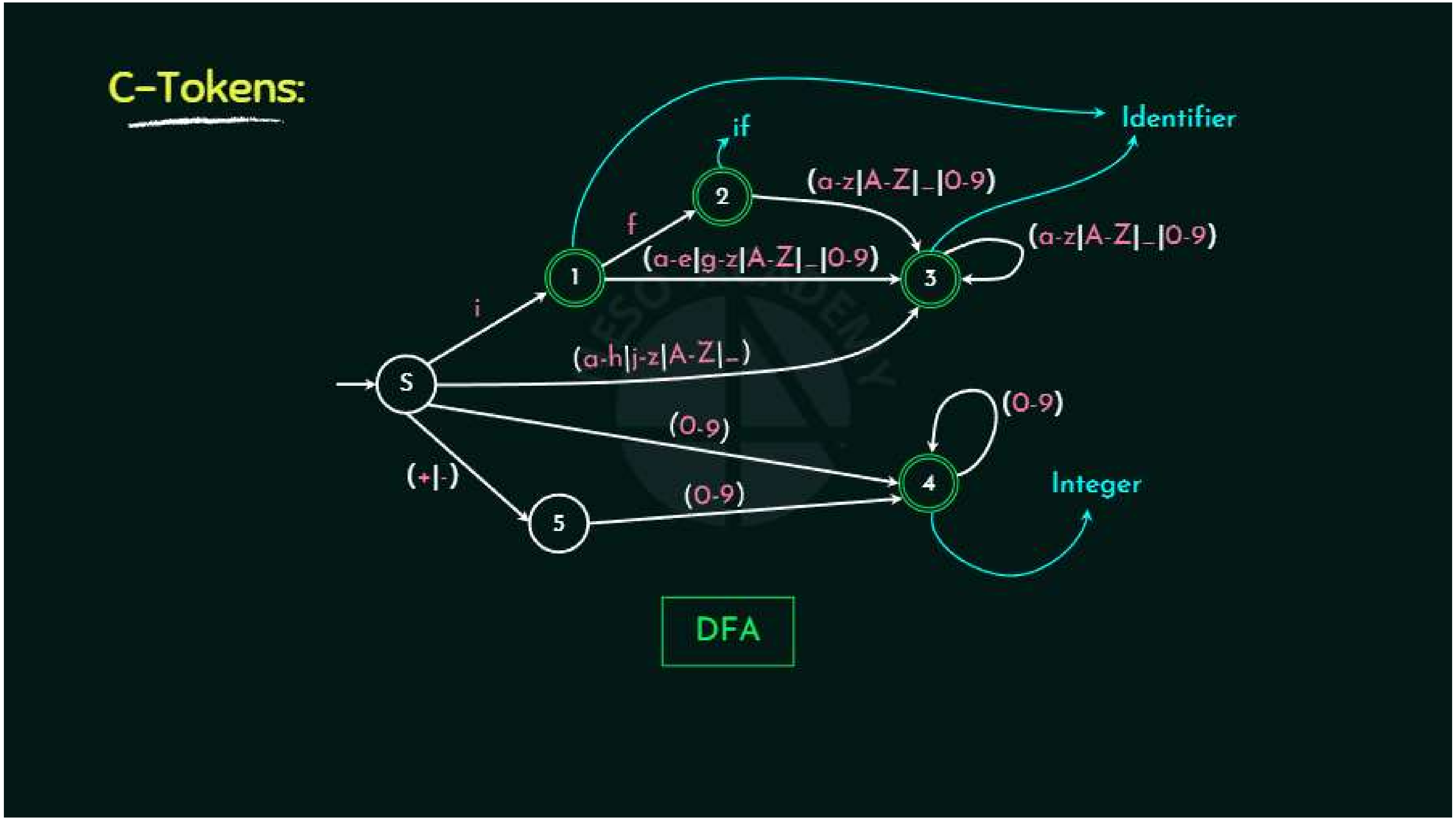



Lexical Analyzer:
Lexemes
Scanning
Eliminate
Non-Token
Elements
Analyzing
Tokens
C-Tokens:
• if:
A
i
B
f
C
• Identifier:
D
(a-z|A-Z|_)
E
(a-z|A-Z|_|0-9)
• Integer:
F
(+|-)
G
(0-9)
H
ϵ
ϵ

C-Tokens:



NFA


C-Tokens:
if
Identifier
(a-z|A-Z|_|0-9)
f
(a-e|g-z|A-Z|_|0-9)
(a-z|A-Z|_|0-9)
i
(a-h|j-z|A-Z|_)
S
1
2
3
4
5
(0-9)
(0-9)
(+|-)
(0-9)
Integer
DFA

Lexical Analyzer:

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

Uses DFA for pattern matching!

Lexical Analyzer:

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

Uses DFA for pattern matching!

Lexical Analyzer:

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

```
// Single line comment

/* Multi
line
Comment*/
```

```
int NE/*it's a comment*/SO;
```

⬇

```
int NE SO;
```

Lexical Analyzer:

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

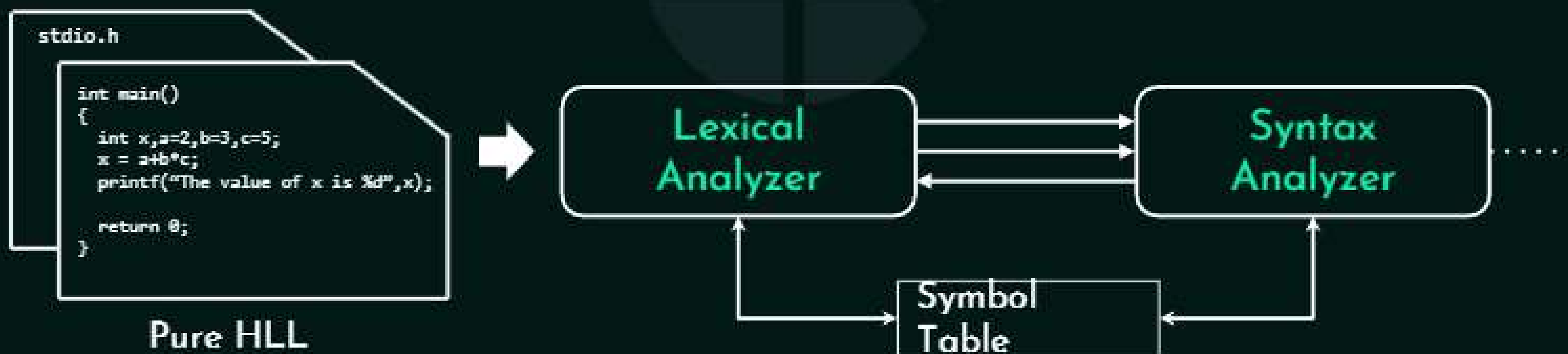
```
// Single line comment

        /* Multi
         line
        Comment*/
```

```
' '      space
'\t'     horizontal tab
'\n'     newline
'\v'     vertical tab
'\f'     form feed
'\r'     carriage
return
```

Lexical Analyzer:

- Scans the Pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens.
- Removes comments and whitespaces from the Pure HLL code.
- Helps in macro expansion in the Pure HLL code.

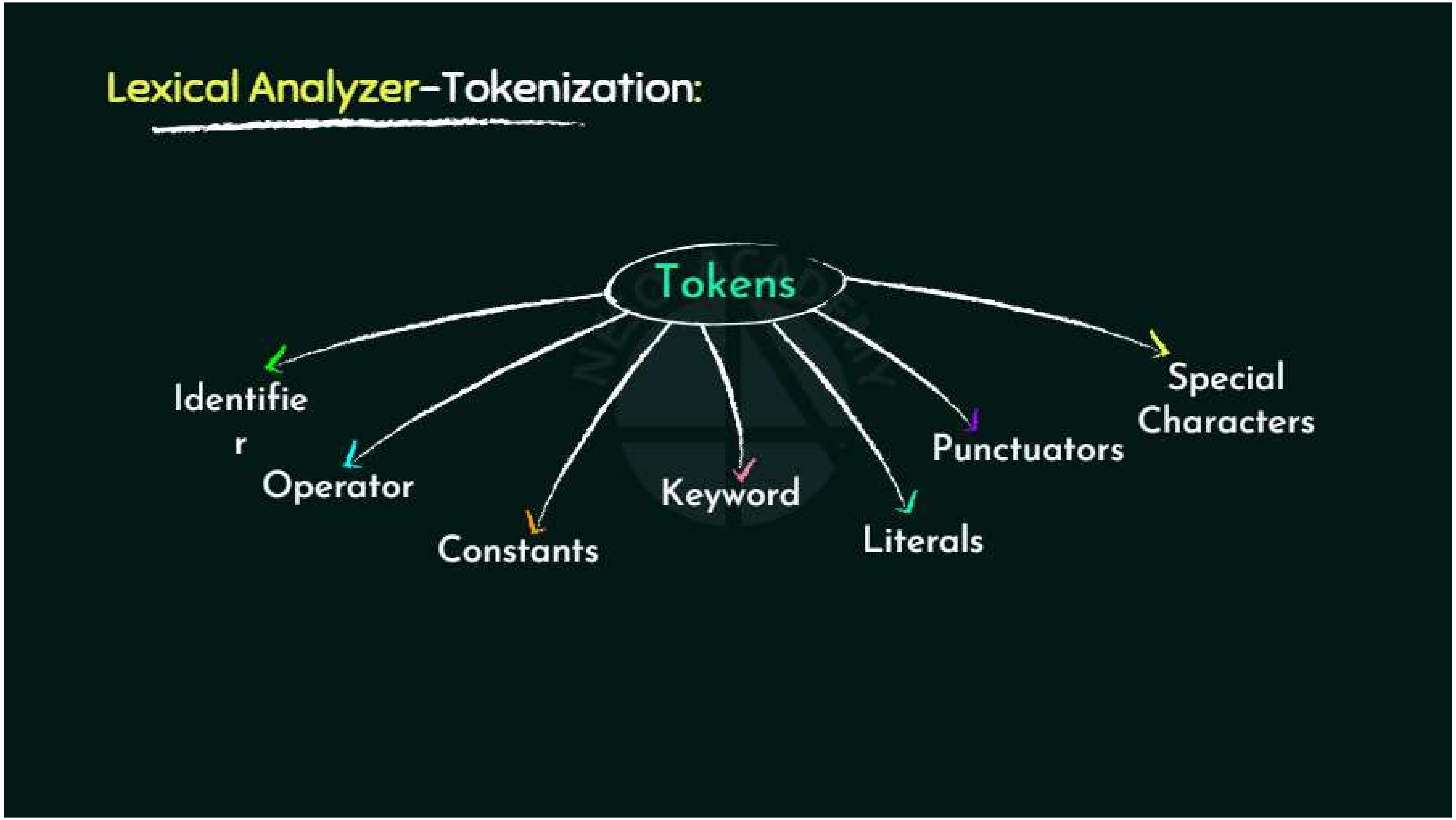


Summary

☆ Working principle of Lexical Analyzer

Outcome

☆ Count the number of tokens in a given code segment


Lexical Analyzer-Tokenization:
Tokens
Identifie
r
Operator
Constants
Keyword
Literals
Punctuators
Special
Characters

Lexical Analyzer–Tokenization:

```
int main()
{
  int x,a=2,b=3,c=5;
  x = a+b*c;
  printf("The value of x is %d",x);

  return 0;
}
```

Lexical Analyzer–Tokenization:

```
int main()
{
  int x,a=2,b=3,c=5;
  x = a+b*c;
  printf("The value of x is %d",x);

  return 0;
}
```

Tokens:

1. Keyword: int , return
2. Identifier: main , x , a , b , c , printf
3. Punctuator: (,) , { , , , ; , }
4. Operator: = , + , *
5. Constant: 2 , 3 , 5 , 0
6. Literal: "The value of x is %d"

Count: 39

Summary

☆ Count the number of tokens in a given code segment

Outcome

☆ Three Solved questions on Lexical Analyzer.

Q1: The lexical analysis for a modern computer language such as Java needs the power of which one of the following machine models in a necessary and sufficient sense?

GATE 2011

(A) Finite state automata

(B) Deterministic pushdown automata

(C) Non-Deterministic pushdown automata

(D) Turing Machine

Q2: The output of a lexical analyzer is

ISRO 2017

(A) A parse tree

(B) Intermediate code

(C) Machine code

$\boxed{\text{(D) A stream of tokens}}$

Q3: The number of tokens in the following C statement is

GATE 2000

```
printf("i=%d, &i=%x",i,&i);
```

(A) 3

(B) 26

(C) 10

(D) 21

Q3: The number of tokens in the following C statement is

GATE 2000

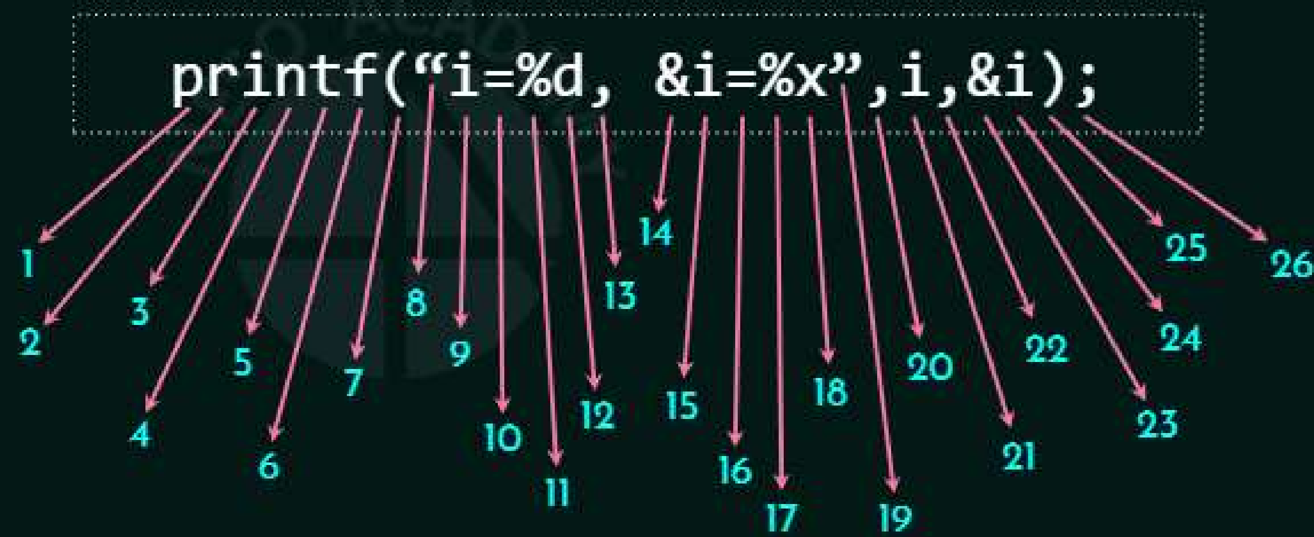
```
printf("i=%d, &i=%x",i,&i);
```

(A) 3

(B) 26

(C) $\boxed{10}$

(D) 21



Q3: The number of tokens in the following C statement is

GATE 2000

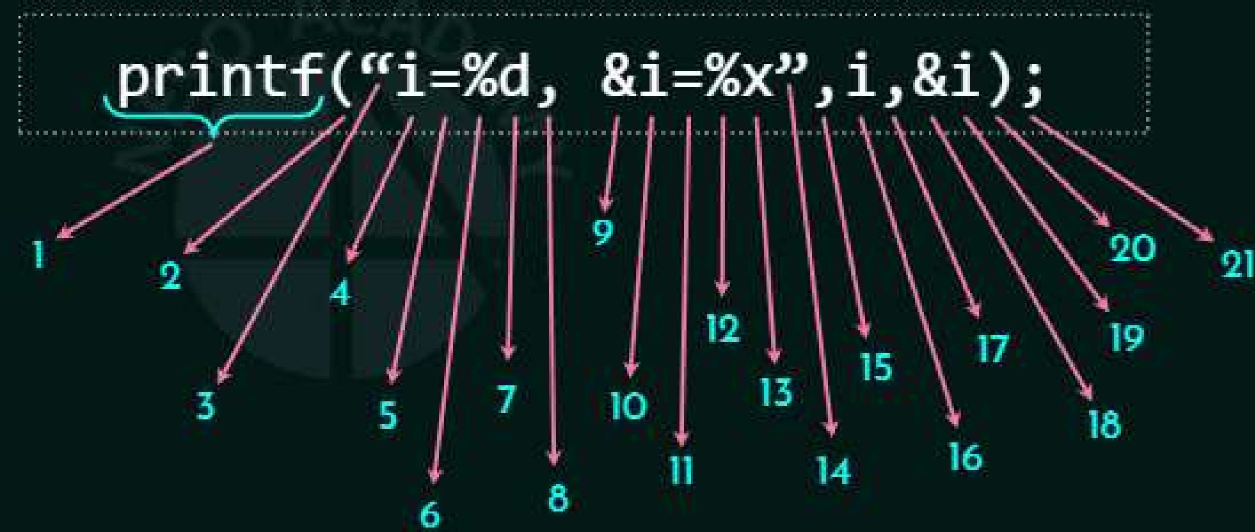
```
printf("i=%d, &i=%x",i,&i);
```

(A) 3

(B) 26

(C) 10

(D) 21



Summary

☆ Three Solved questions on Lexical Analyzer.

Outcome

☆ Two Solved questions on Lexical Analyzer.

Q1: In a compiler, keywords of a language are recognized during

(A) parsing of the program

(B) the code generation

(C) the lexical analysis of the program

(D) dataflow analysis

GATE 2011

NIELIT Scientist-B 2017

Q2: A lexical analyzer uses the following patterns to recognize three tokens T1, T2, and T3 over the alphabet {a,b,c}.

T_1: $a?\,(b|c)^*a$

T_2: $b?\,(a|c)^*b$

T_3: $c?\,(b|a)^*c$

Note that 'x?' means 0 or 1 occurrence of the symbol x. Note also that the analyzer outputs the token that matches the longest possible prefix.

If the string $bbaacabc$ is processed by the analyzer, which one of the following is the sequence of tokens it outputs?

(A) $T_1T_2T_3$ (B) $T_1T_1T_3$ (C) $T_2T_1T_3$ (D) T_3T_3

GATE 2018

Q2: A lexical analyzer uses the following patterns to recognize three tokens T1, T2, and T3 over the alphabet {a,b,c}.

T_1: $a?\,(b|c)^*a \longrightarrow (b|c)^*a + a(b|c)^*a$

T_2: $b?\,(a|c)^*b \longrightarrow (a|c)^*b + b(a|c)^*b$

T_3: $c?\,(b|a)^*c \longrightarrow (b|a)^*c + c(b|a)^*c$

If the string $bbaacabc$ is processed by the analyzer, which one of the following is the sequence of tokens it outputs?

(A) $T_1T_2T_3$ (B) $T_1T_1T_3$ (C) $T_2T_1T_3$ (D) T_3T_3

Q2: A lexical analyzer uses the following patterns to recognize three tokens T1, T2, and T3 over the alphabet {a,b,c}.

T_1: $a?\ (b|c)^* a \longrightarrow (b|c)^* a + a(b|c)^* a$

T_2: $b?\ (a|c)^* b \longrightarrow (a|c)^* b + b(a|c)^* b$

T_3: $c?\ (b|a)^* c \longrightarrow (b|a)^* c + c(b|a)^* c$

(A) $T_1 T_2 T_3$

$$\underbrace{bba}_{T_1}\underbrace{acab}_{T_2}\underbrace{c}_{T_3}$$

Q2: A lexical analyzer uses the following patterns to recognize three tokens T1, T2, and T3 over the alphabet {a,b,c}.

T_1: $a?\ (b|c)^*a$
T_2: $b?\ (a|c)^*b$
T_3: $c?\ (b|a)^*c$

Note that 'x?' means 0 or 1 occurrence of the symbol x. Note also that the analyzer outputs the token that matches the longest possible prefix.

If the string *bbaacabc* is processed by the analyzer, which one of the following is the sequence of tokens it outputs?

$\underbrace{bba}_{T_1}\underbrace{acab}_{T_2}\underbrace{c}_{T_3}$ $\quad$ $\underbrace{bba}_{T_1}\underbrace{aca}_{T_1}\underbrace{bc}_{T_3}$ $\quad$ $\underbrace{bb}_{T_2}\underbrace{aa}_{T_1}\underbrace{cabc}_{T_3}$ $\quad$ $\boxed{\underbrace{bbaac}_{T_3}\underbrace{abc}_{T_3}}$

Q2: A lexical analyzer uses the following patterns to recognize three tokens T1, T2, and T3 over the alphabet {a,b,c}.

T_1: $a?\,(b|c)^*a$

T_2: $b?\,(a|c)^*b$

T_3: $c?\,(b|a)^*c$

Note that 'x?' means 0 or 1 occurrence of the symbol x. Note also that the analyzer outputs the token that matches the longest possible prefix.

If the string $bbaacabc$ is processed by the analyzer, which one of the following is the sequence of tokens it outputs?

(A) $T_1T_2T_3$ (B) $T_1T_1T_3$ (C) $T_2T_1T_3$ (D) $\boxed{T_3T_3}$

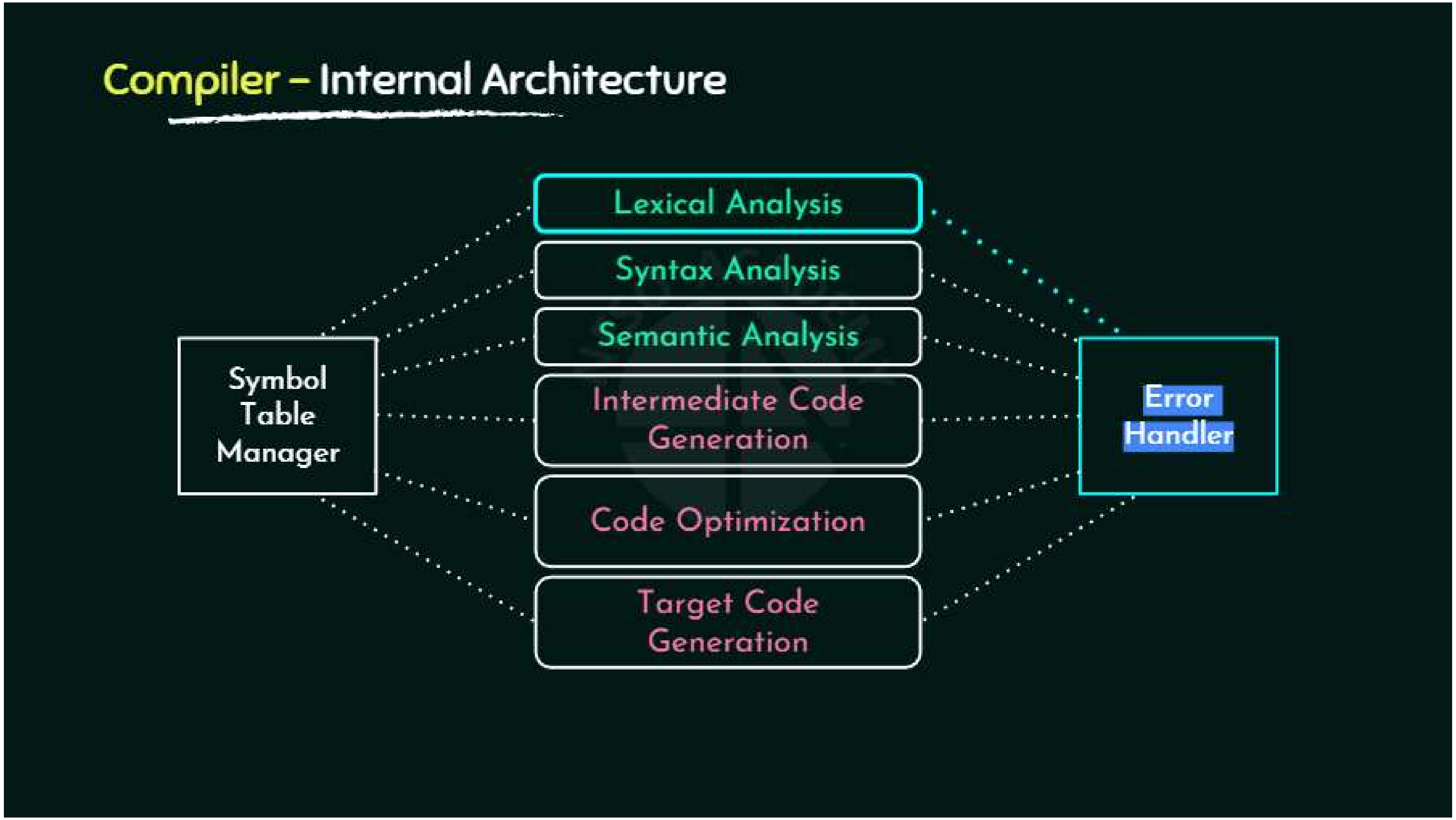
GATE 2018

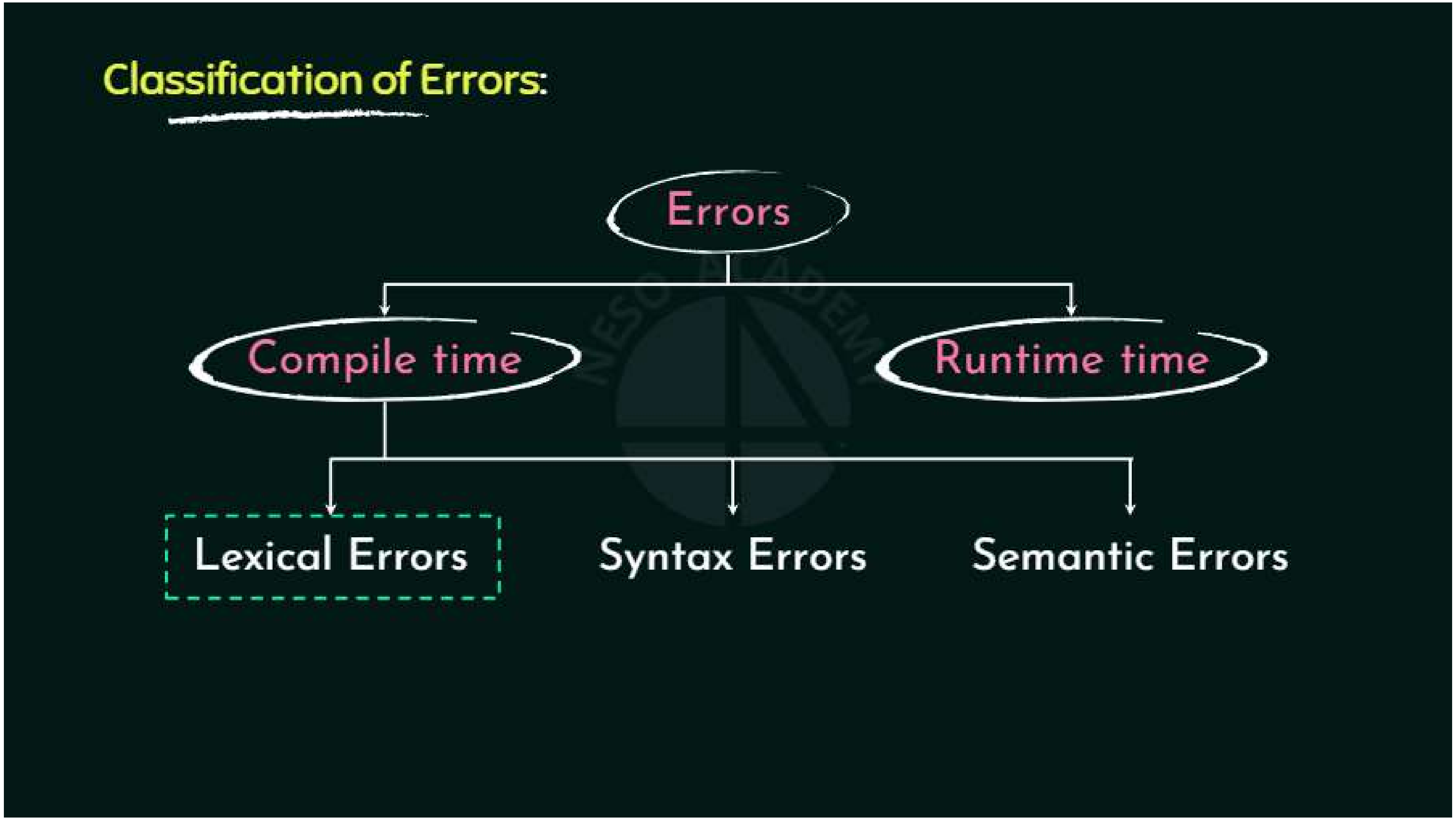
Summary

☆ Two Solved questions on Lexical Analyzer.

Outcome

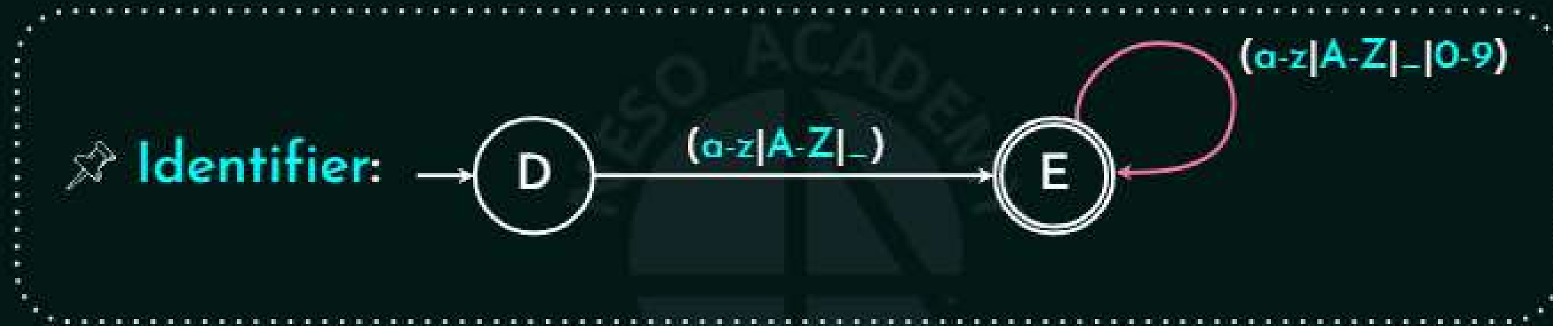
- ☆ Role of Error handler, especially for Lexical Analysis.
- ☆ Types of Error.
- ☆ Different types of Lexical Errors.
- ☆ Error Recovery in Lexical Analysis.


Compiler – Internal Architecture
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Target Code Generation
Symbol Table Manager
Error Handler


Classification of Errors:
Errors
Compile time
Runtime time
Lexical Errors
Syntax Errors
Semantic Errors

Lexical Errors:

- Identifiers that are way too long.



C : 31/247

C++ : 2048

: 79

Lexical Errors:

- Identifiers that are way too long.
- Exceeding length of numeric constants.

```
int i = 4567891;
```

Size: 2 Bytes

-32,768 to 32,767

Lexical Errors:

- Identifiers that are way too long.
- Exceeding length of numeric constants.
- Numeric constants which are ill-formed.

```
int i = 4567$91;
```

Lexical Errors:

- Identifiers that are way too long.
- Exceeding length of numeric constants.
- Numeric constants which are ill-formed.
- Illegal characters that are absent from the source code.

```
char x[] = “NESO ACADEMY”;$
```

Lexical Error-Recovery:

- Panic-mode recovery.

```
int 4NESO;
```

Lexical Error-Recovery:

- Panic-mode recovery.

```
while(condition)
{
}
```

Lexical Error-Recovery:

- Panic-mode recovery.
- Transpose of two adjacent characters.

```
unoin test
{
        int x;
        float
y;
} T1;
```

```
union test
{
        int x;
        float
y;
} T1;
```

Lexical Error-Recovery:

- Panic-mode recovery.
- Transpose of two adjacent characters.
- Insert a missing character.
- Delete an unknown or extra character.
- Replace a character with another.

```
itt NESO;   →   int NESO;
```

Lexical Error-Recovery:

- Panic-mode recovery.
- Transpose of two adjacent characters.
- Insert a missing character.
- Delete an unknown or extra character.
- Replace a character with another.

Summary

- ☆ Role of Error handler, especially for Lexical Analysis.
- ☆ Types of Error.
- ☆ Different types of Lexical Errors.
- ☆ Error Recovery in Lexical Analysis.