

Introduction to Compiler Design | Neso Academy

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

CHAPTER - 1

Introduction to Compiler Design

Neso Academy

Introduction to Compiler Design Neso Academy CHAPTER-1



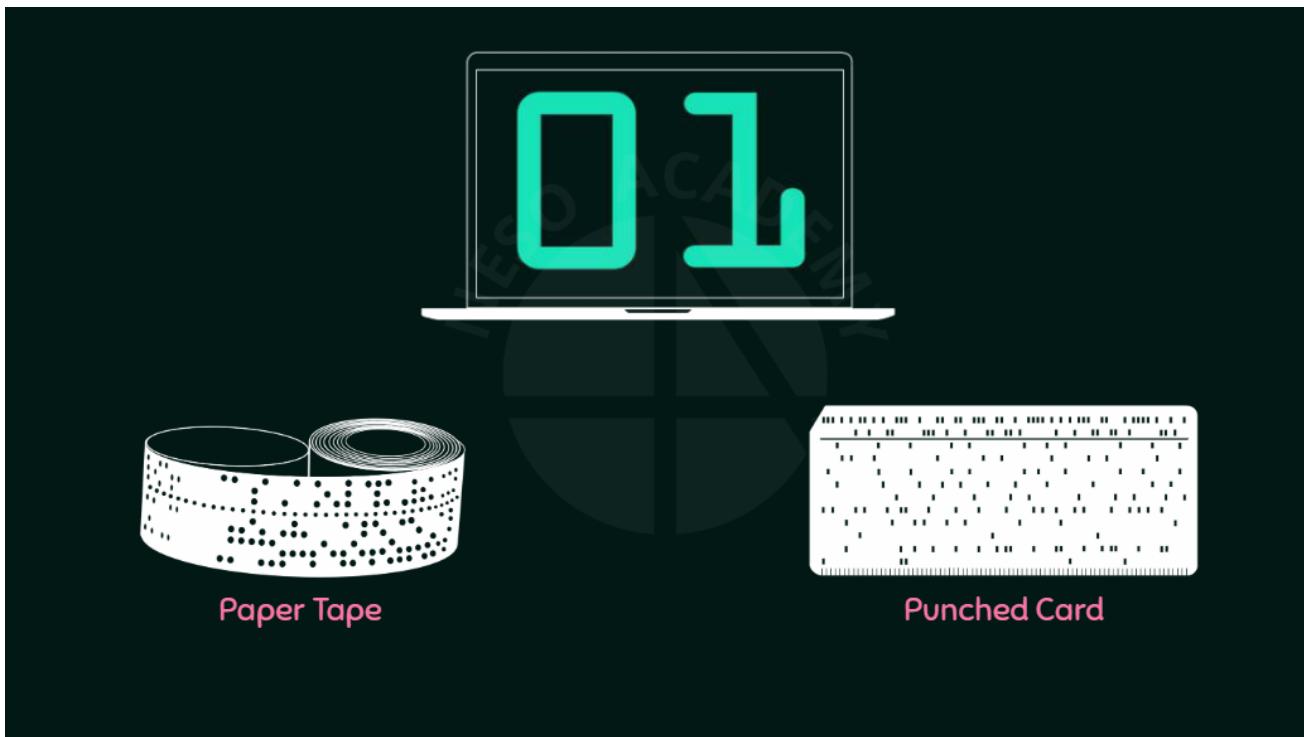
```
#include<stdio.h>
int main() {
    printf("Hello World");
    return 0;
}
```

The diagram illustrates the compilation process. On the left, a blue rectangular box contains a C program with the code: #include<stdio.h>, int main() { printf("Hello World"); return 0; }. A green horizontal bar extends from the bottom of this box. To the right of the bar is a blue gear icon. Further to the right, the same C code is shown again, followed by its binary representation: 010000000100, 000011000100, 001000000100, 000010100000, 011010010100.

Compiler Design

Introduction

Compiler Design Introduction



Paper Tape Punched Card

Punched Card:

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

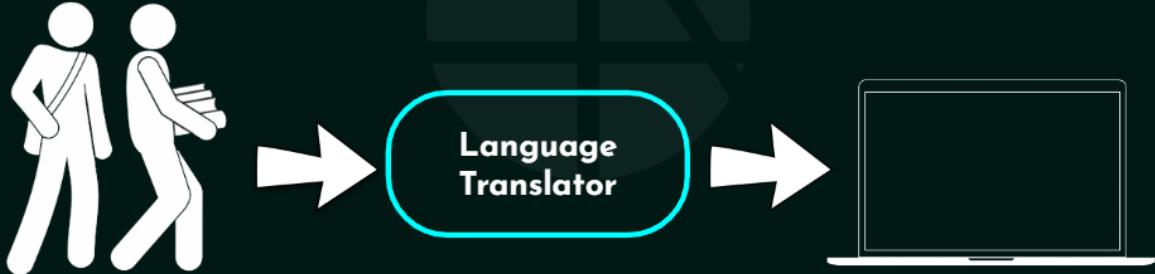
Stick figures carrying books are positioned to the left of the ASCII table, and a laptop icon is to the right.

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	C -	1000011
u -	1110101	a -	1100001
n -	1101110	r -	1110010
c -	1100011	d -	1110011
h -	1101000		
e -	1100101		
d -	1100100		



Punched Card: ASCII
P - 1010000 u - 1110101 n - 1101110 c - 1100011 h - 1101000 e - 1100101 d - 1100100
LanguageTranslator

Language Translator:

i. Assembler:

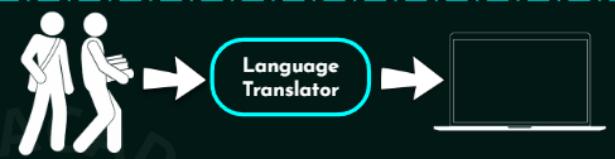
```
MOV R1, 02H  
MOV R2, 03H  
ADD R1, R2  
STORE X, R1
```

Assembly Language

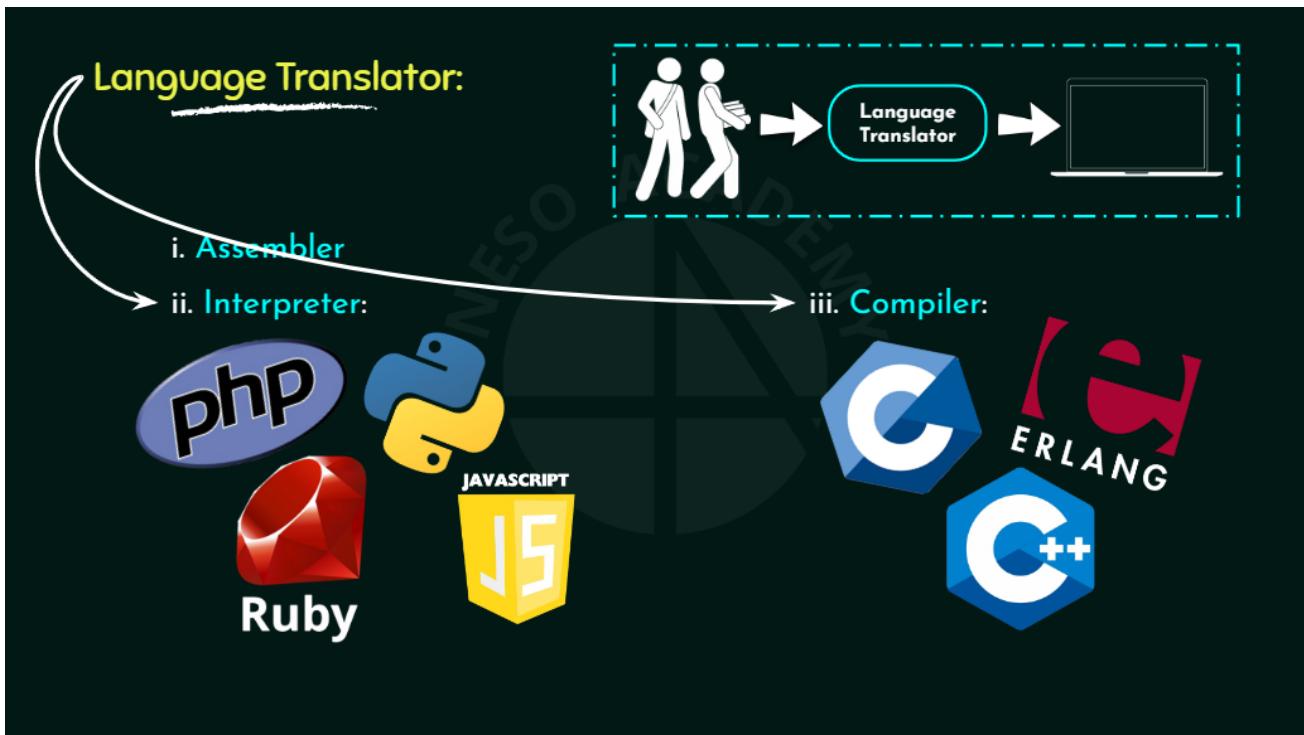
Assembler

```
0110100101010  
0010101010010  
01001111100101  
01010101010101
```

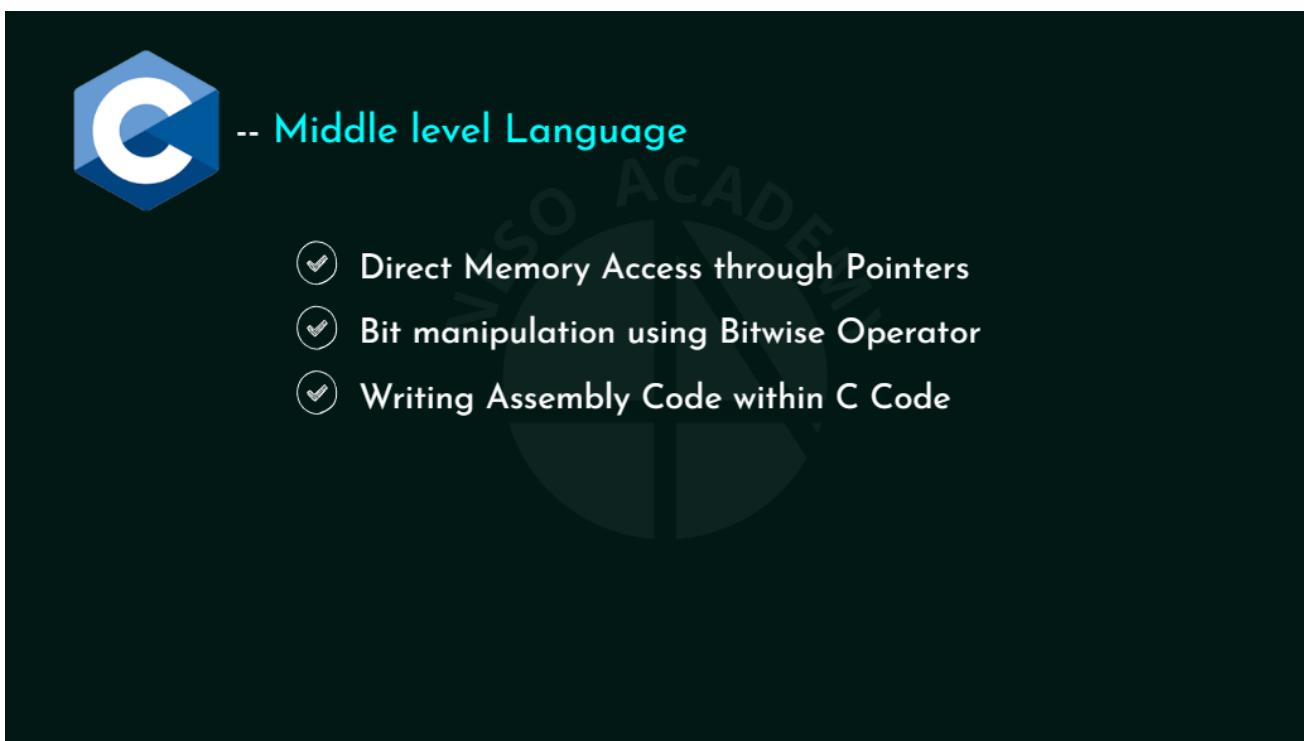
Machine Code



Language Translator:i. Assembler:AssemblerAssemblyLanguage MOV R1, 02HMOV R2, 03HADD R1, R2STORE X, R1MachineCode01101001010100010101001001001111001010101010101Language Translator



Language Translator:i. Assemblerii. Interpreter:iii. Compiler:Language Translator



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



-- High Level Language

```
#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;
}
```

x = a + b * c

-- High Level Language#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;} x=a+b*c



-- 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

-- 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



-- 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
10110100101010
11010101010010
10011110010101
01010101010101
11010101010010
10011110010101
```

Machine Code

-- High Level Language Source Code / HLL Code
#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; }
Language Translator Machine
Code 001010101001010110100101011010101001111100101010101010110101010010100111110010101

Language Translator – Internal Architecture

Language Translator

Language Translator - Internal Architecture Language Translator

Language Translator – Internal Architecture



Language Translator - Internal Architecture Preprocessor Compiler Assembler Linker/Loader

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



```
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 Source Code / HLL Code
#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;}
Preprocessor
int main() { int x,a=2,b=3,c=5; x = a+b*c; printf("The value of x is %d",x); return 0;}
stdio.h
Pure HLL

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



```
.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

int main() { int x,a=2,b=3,c=5; x = a+b*c; printf("The value of x is %d",x); return 0;}stdio.hPure HLL Language Translator - Internal ArchitectureCompiler.LC0:.string "The value of x is %d"main:push rbpmov rbp, rspsub rsp, 16mov DWORD PTR [rbp-4], 2mov DWORD PTR [rbp-8], 3mov DWORD PTR [rbp-12], 5mov eax, DWORD PTR [rbp-8]imul eax, DWORD PTR [rbp-12]mov edx, eaxmov eax, DWORD PTR [rbp-4]add eax, edxmov DWORD PTR [rbp-16], eaxmov eax, DWORD PTR [rbp-16]mov esi, eaxmov edi, OFFSET FLAT:.LC0mov eax, 0call printfmov eax, 0leaveretAssembly Language

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



```
i+0:001010101001
i+1:0101101001100
i+2:10101101010101
i+3:01000101001101
i+4:11100101010101
i+5:01010101010111
:
```

Relocatable
Machine Code

Language Translator - Internal ArchitectureAssembler.LC0:.string "The value of x is %d"main:push rbpmov rbp, rspsub rsp, 16mov DWORD PTR [rbp-4], 2mov DWORD PTR [rbp-8], 3mov DWORD PTR [rbp-12], 5mov eax, DWORD PTR [rbp-8]imul eax, DWORD PTR [rbp-12]mov edx, eaxmov eax, DWORD PTR [rbp-4]add eax, edxmov DWORD PTR [rbp-16], eaxmov eax, DWORD PTR [rbp-16]mov esi, eaxmov edi, OFFSET FLAT:.LC0mov eax, 0call printfmov eax, 0leaveretAssembly Language Relocatable Machine Codei+0:001010101001i+1:0101101001100i+2:10101101010101i+3:01000101001101i+4:11100101010101i+5:010101010111

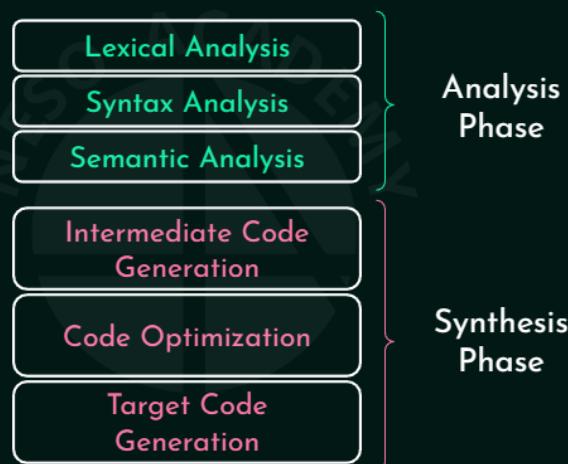
Language Translator – Internal Architecture



Linker/Loader Language Translator - Internal Architecture Relocatable Machine

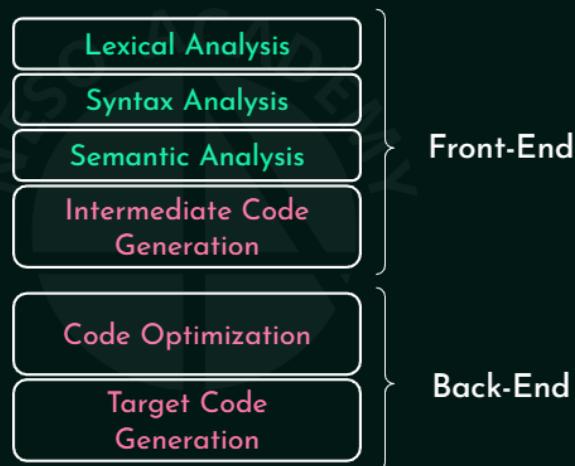
Code i+0:001010101001
Machine Code 0x000004B8: 001010101001
Code i+1:0101101001100
Machine Code 0x000004B9: 0101101001100
Code i+2:10101101010101
Machine Code 0x000004BA: 10101101010101
Code i+3:0100101001101
Machine Code 0x000004BB: 0100101001101
Code i+4:11100101010101
Machine Code 0x000004BC: 11100101010101
Code i+5:01010101010111
Machine Code 0x000004BD: 01010101010111

Compiler – Internal Architecture



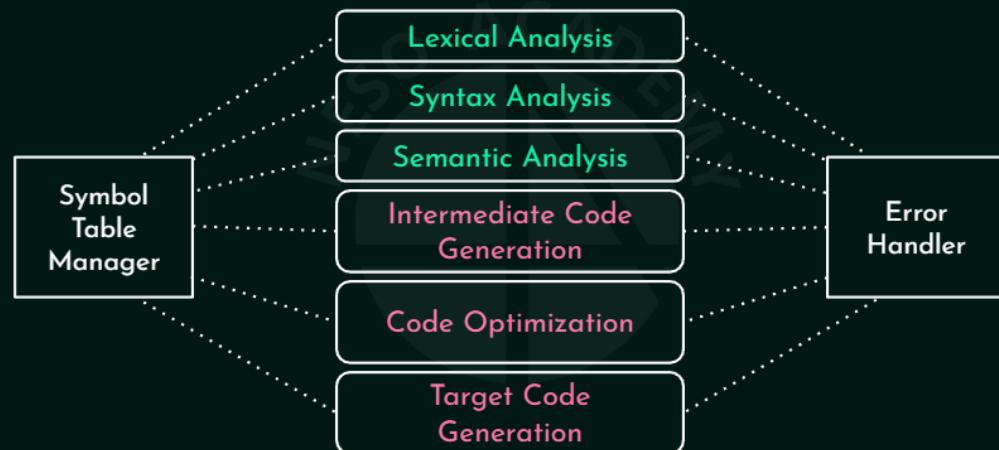
Compiler - Internal Architecture Lexical Analysis Syntax Analysis Semantic Analysis Intermediate Code Generation Code Optimization Target Code Generation Analysis Phase Synthesis Phase

Compiler - Internal Architecture



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

Compiler - Internal Architecture



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

Syllabus:IntroductionSyntax AnalyzerParsers - Top downParsers - Bottom upSyntax Directed Translation SchemesIntermediate Code GenerationRuntime Environment & Code Optimization

Prerequisite:

- Flag icon Knowledge of Theory Of Computation.

Target Audience:

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

Prerequisite:Knowledge of Theory Of Computation.Target Audience:College and University Scholars.Any competitive exam (GATE / NET / NIELIT etc.) aspirants.Any Computer Science admirer.



Compiler Design

Different Phases of Compiler

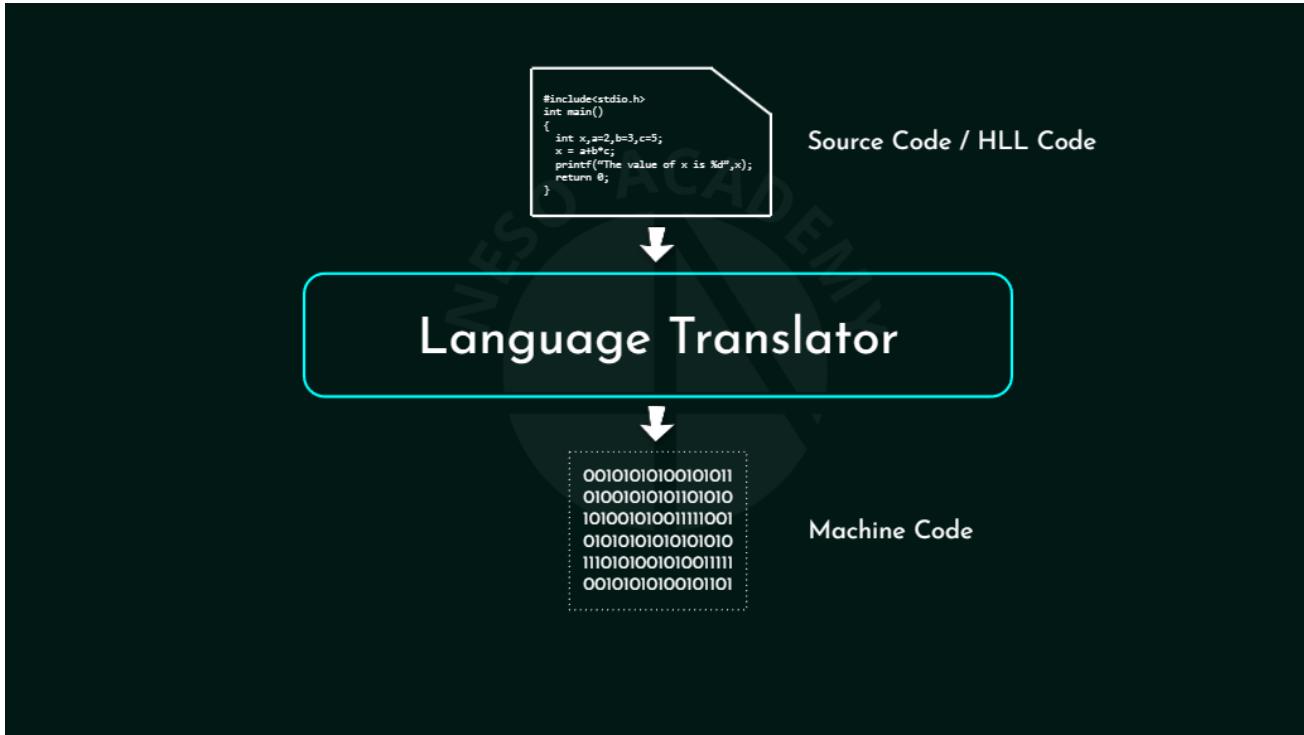
Compiler Design
Different Phases of Compiler



Outcome

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

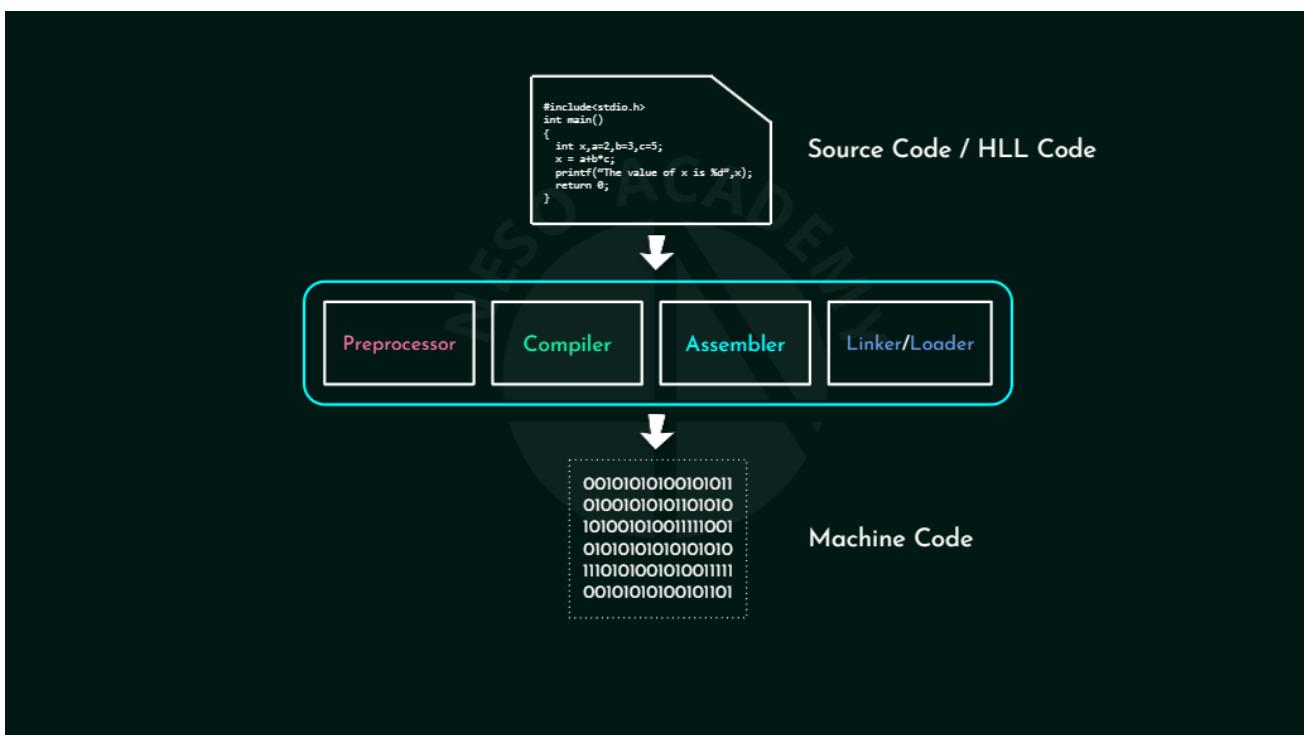
Outcome
Tools to implement different phases
Overview of various phases of Compiler



Machine

Source Code / HLL 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; }

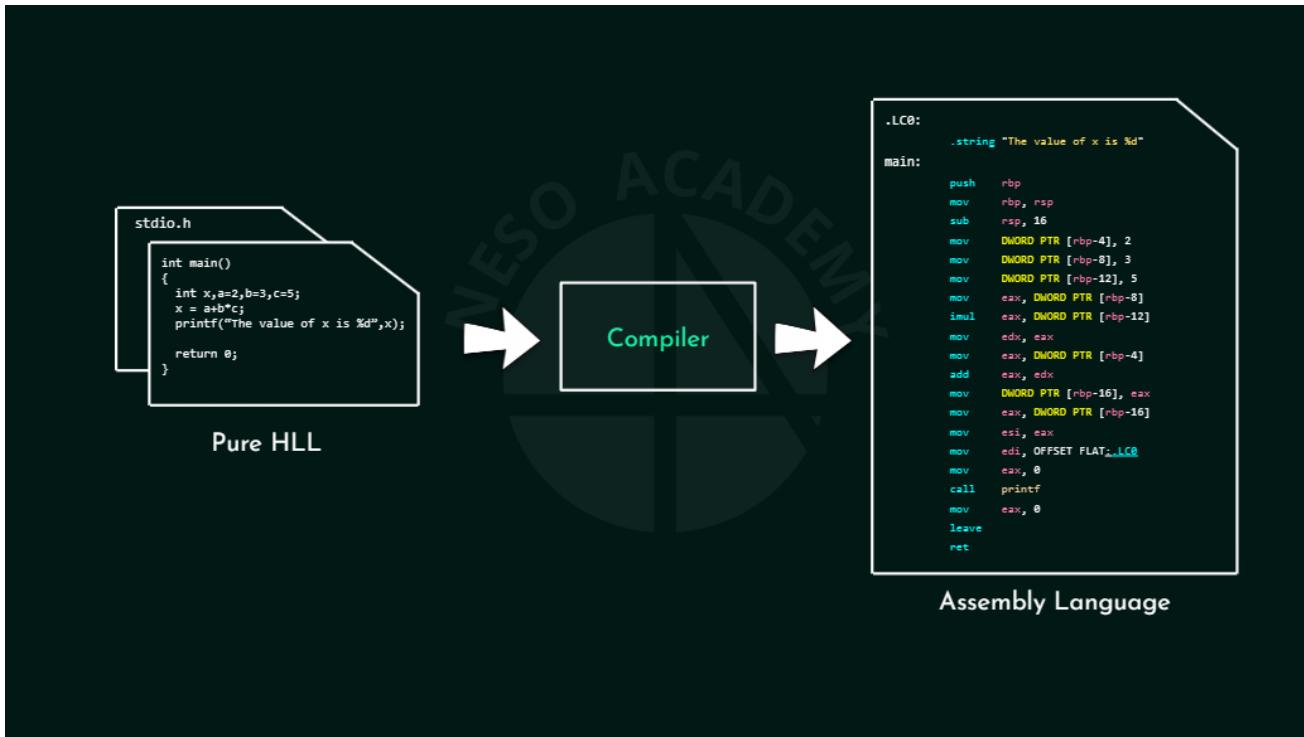
0;}Language Translator



Preprocessor Compiler Assembler Linker/Loader Source Code / HLL Code Machine

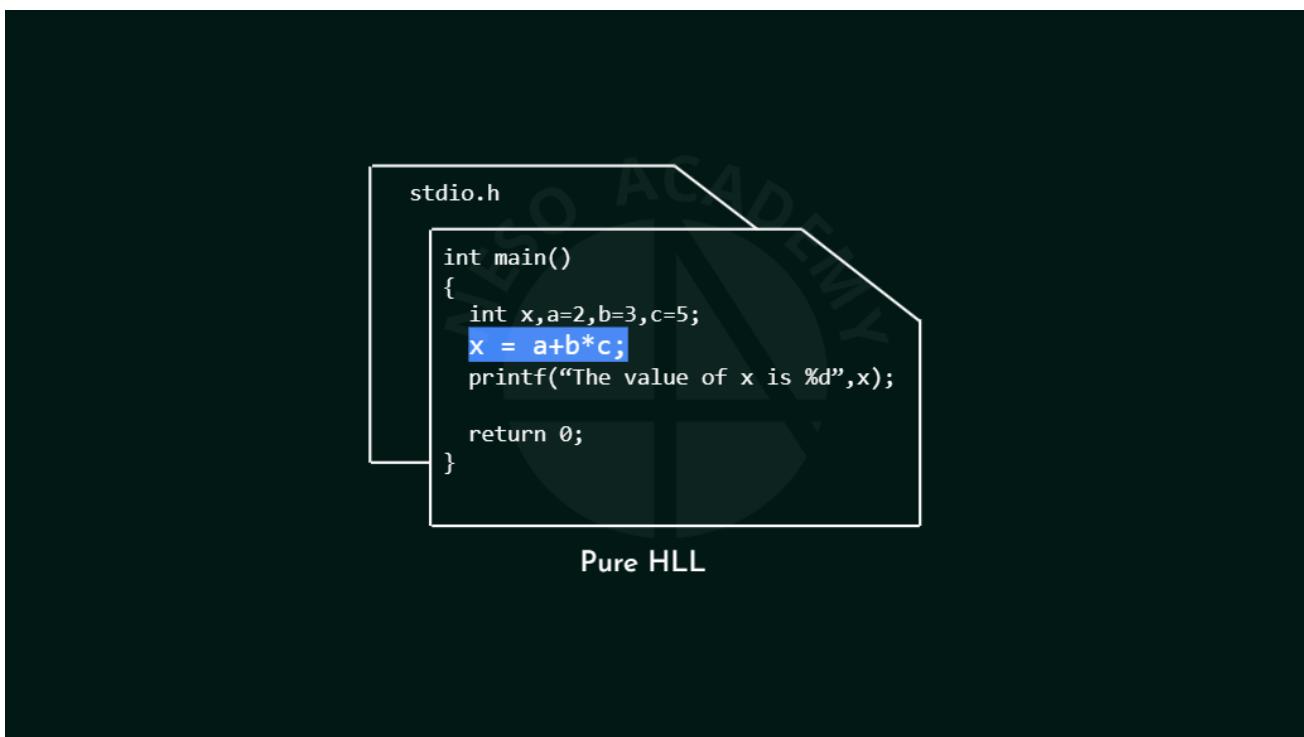
Code00101010100101011010010101011010101010010100111110010101010101010111010100101001111100101010100101101

```
#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;}
```



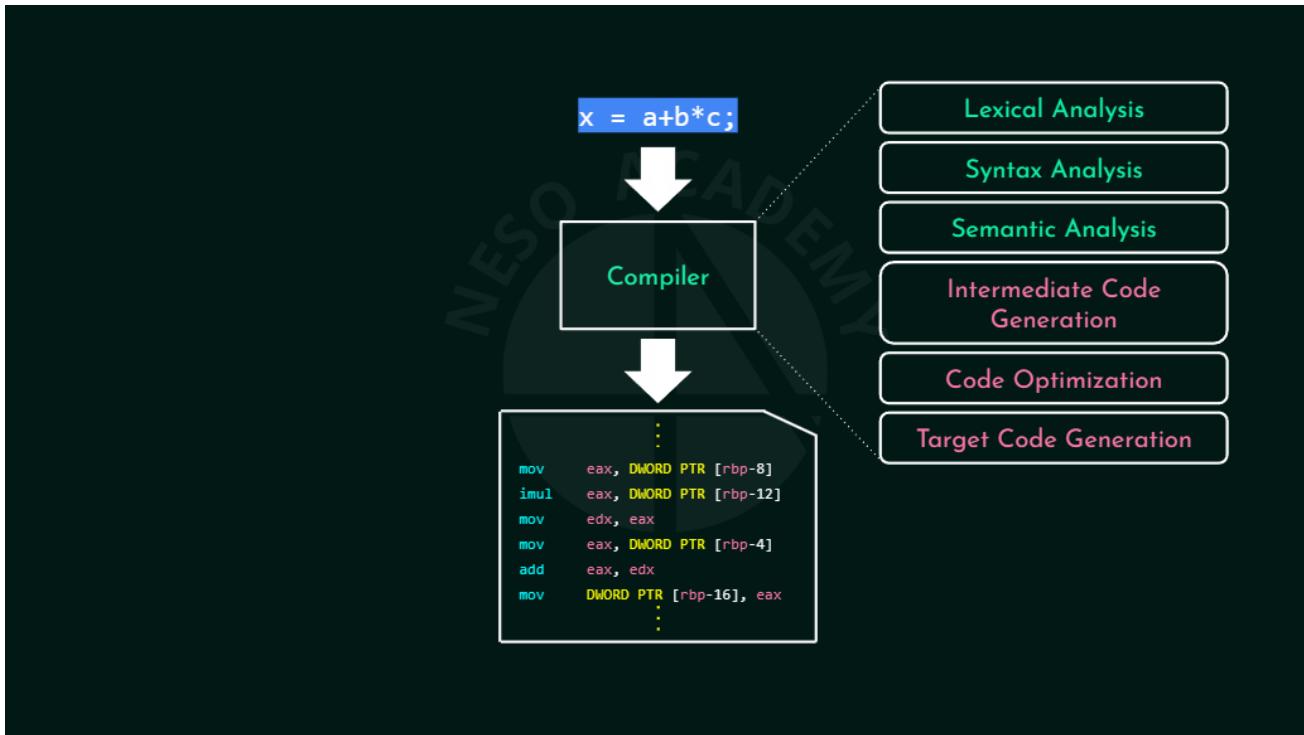
Pure HLL

```
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;}.LC0:.string "The value of x is %d"main:push rbpmov rbp, rsp, 16mov DWORD PTR [rbp-4], 2mov DWORD PTR [rbp-8], 3mov DWORD PTR [rbp-12], 5mov eax, DWORD PTR [rbp-8]imul eax, DWORD PTR [rbp-12]mov edx, eaxmov eax, DWORD PTR [rbp-4]add eax, edxmov DWORD PTR [rbp-16], eaxmov eax, DWORD PTR [rbp-16]mov esi, eaxmov edi, OFFSET FLAT:.LC0mov eax, 0call printfmov eax, 0leave retAssembly Language Compiler
```

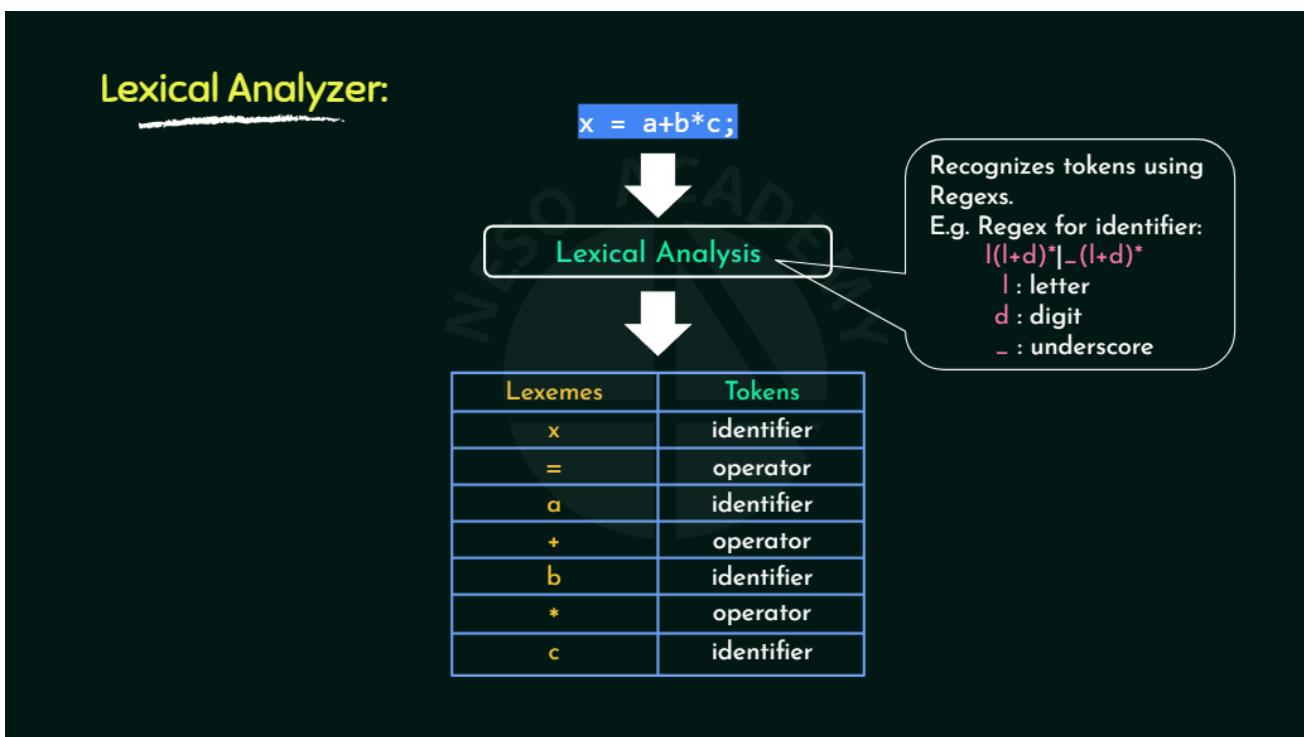


Pure HLL

```
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;}
```



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



Lexical Analysis
 Lexical Analyzer:
 x = a+b*c;
 Lexemes
 Tokens
 identifier = operator
 identifier + operator
 identifier * operator
 identifier
 Recognizes tokens using RegExs. E.g.
 Regex for identifier: $l(l+d)^*|_(l+d)^*$
 l : letter
 d : digit
 _ : underscore

Lexical Analyzer:

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

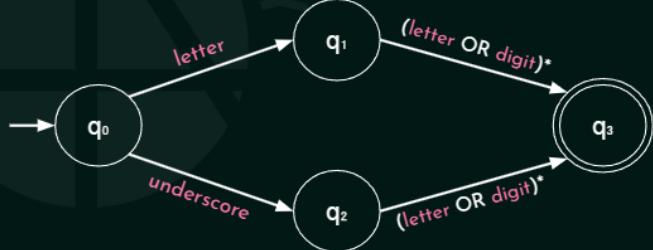
E.g. Regex for identifier:

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

l : letter

d : digit

_ : underscore



Lexical Analyzer:
E.g. Regex for identifier: $|(l+d)^*|_(l+d)^*$ | : letter d : digit _ : underscore
Lexemes Tokens x identifier = operator a identifier + operator b identifier * operator c identifier

Syntax Analyzer:

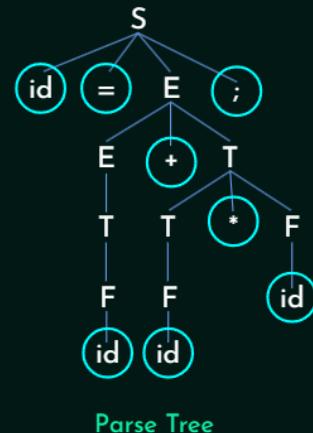
Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

Syntax Analysis

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

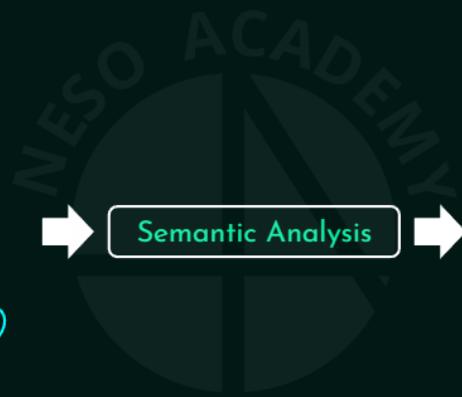
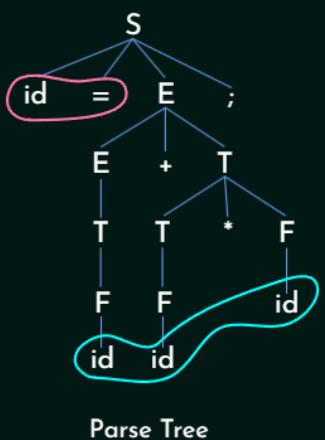
$x = a+b*c;$

$id = id + id * id ;$



Syntax Analysis:
Syntax Analyzer:
Lexemes Tokens x identifier = operator a identifier + operator b identifier * operator c identifier
 $x = a+b*c;$; S $\rightarrow id = E ;$ E $\rightarrow E + T | T$ T $\rightarrow T * F | F$ F $\rightarrow id$

Semantic Analyzer:



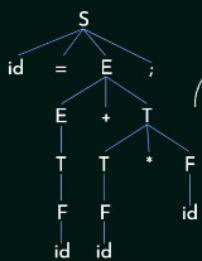
Semantic Analysis Semantic Analyzer: Sid=E;E+TT*FTFFididid Parse Tree Semantically Verified Parse Tree

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. $x = t_1;$

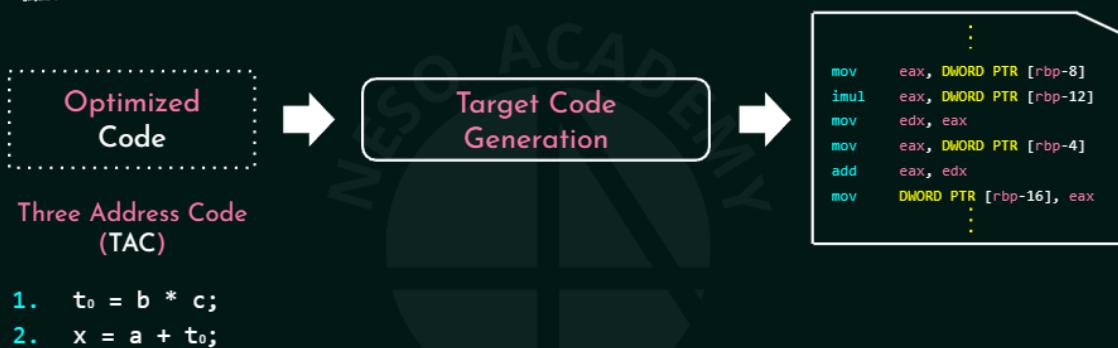
Intermediate Code Generator: Semantically Verified Parse Tree Intermediate Code Generation Intermediate Code $x = a+b*c;$; $id = id + id * id ; \Leftarrow$ Three Address Code (TAC) 1. $t_0 = b * c;$ 2. $t_1 = a + t_0;$ 3. $x = t_1;$

Code Optimizer:

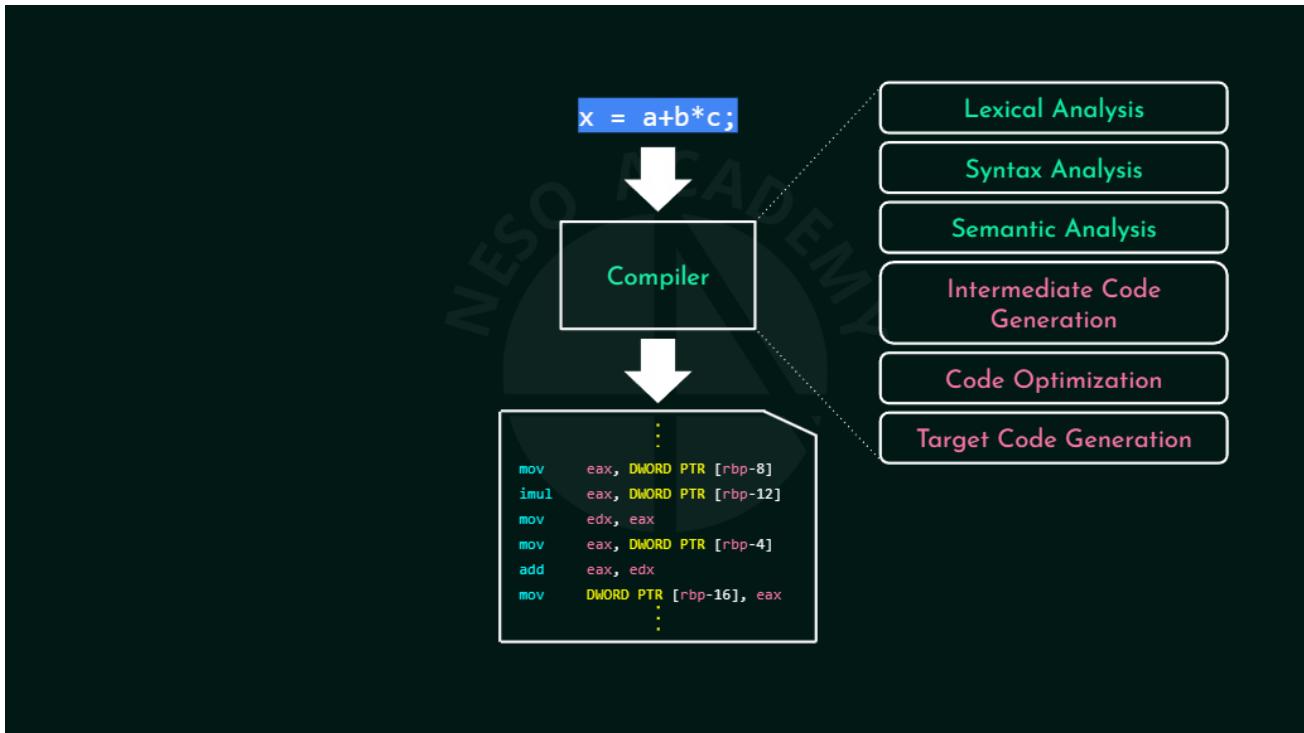


Code Optimization
Intermediate Code Three Address Code (TAC)
1. $t_0 = b * c;$; 2. $t_1 = a + t_0;$; 3. $x = t_1;$
Code Optimizer:
Optimized Code
Three Address Code (TAC)
1. $t_0 = b * c;$; 2. $x = a + t_0;$

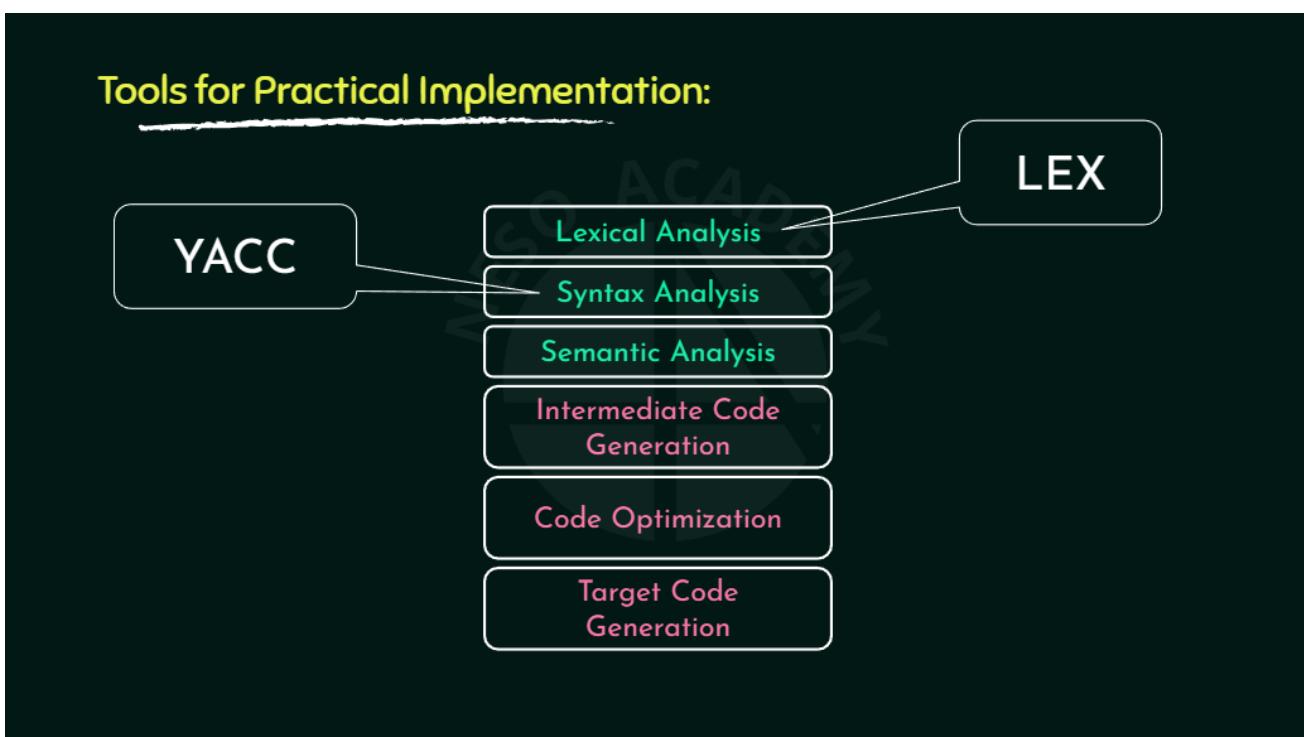
Target Code Generator:



Optimized Code Three Address Code (TAC)
1. $t_0 = b * c;$; 2. $x = a + t_0;$
Target Code Generator:
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

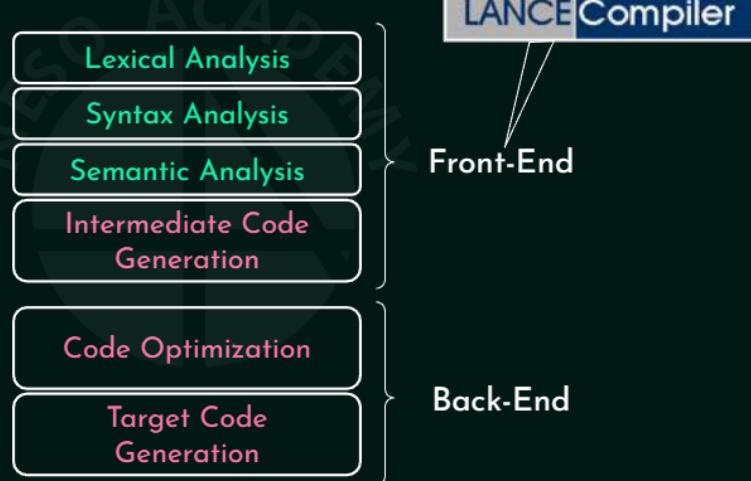


mov eax, DWORD PTR [rbp-8]
imul eax, DWORD PTR [rbp-12]
mov edx, eax
mov eax, DWORD PTR [rbp-4]
add eax, edx
mov DWORD PTR [rbp-16], eax
x = a+b*c; Compiler
Lexical Analysis Syntax Analysis Semantic Analysis Intermediate Code Generation Code Optimization Target Code Generation

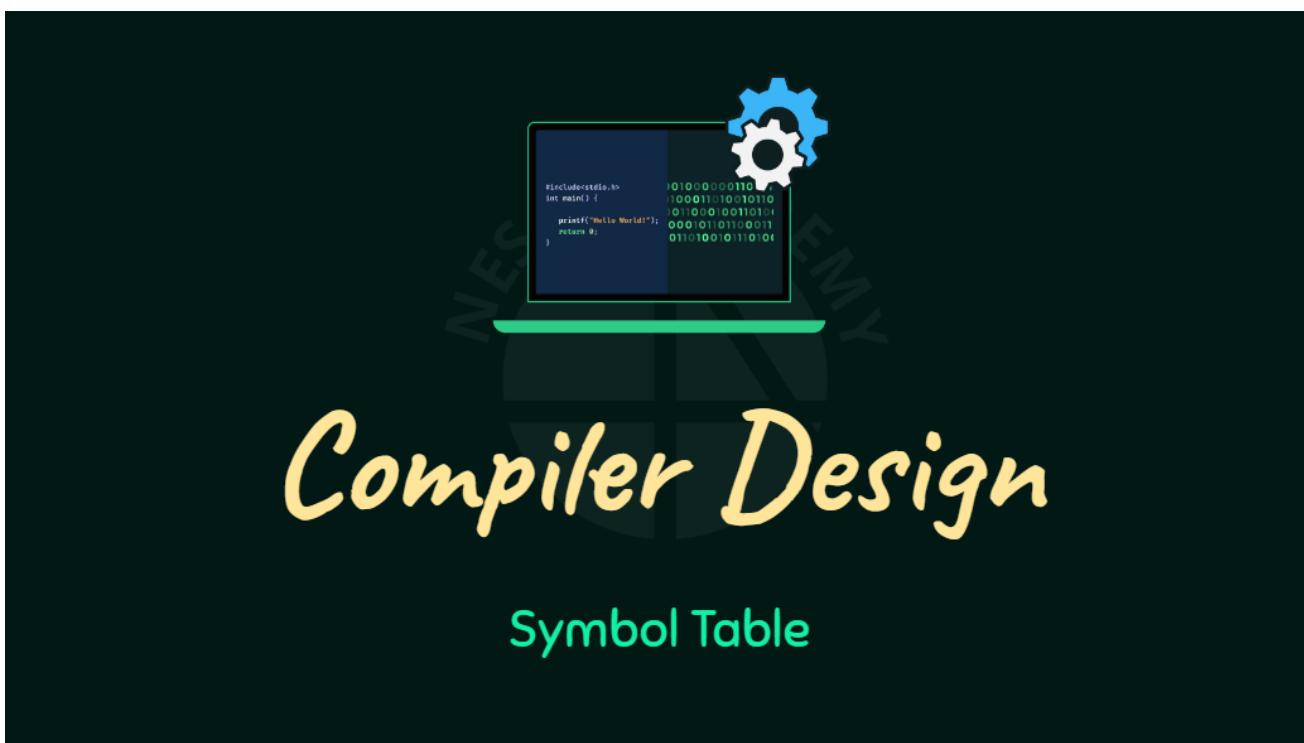


Lexical Analysis Syntax Analysis Semantic Analysis Intermediate Code Generation Code Optimization Target Code Generation Tools for Practical Implementation: LEX YACC

Tools for Practical Implementation:



Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Target Code Generation
Front-End
Back-End
Tools for Practical Implementation:



Compiler Design
Symbol Table

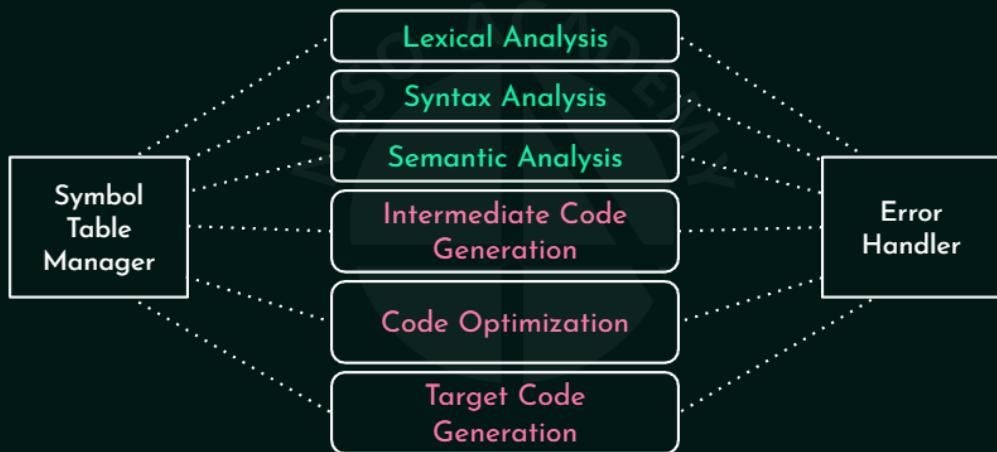


Outcome

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

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

Compiler – Internal Architecture

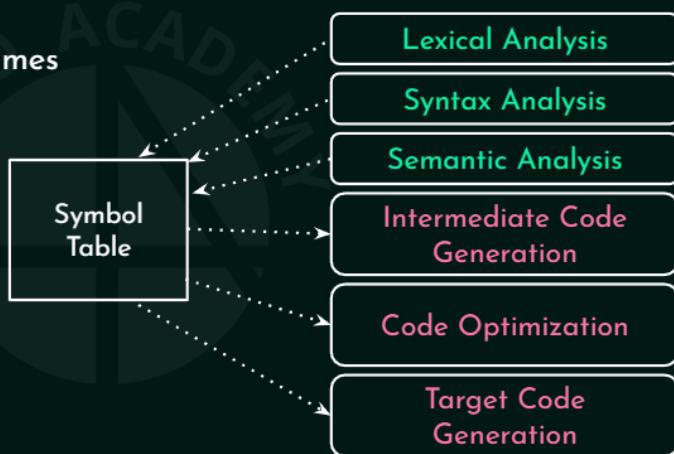


Compiler - Internal Architecture
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Target Code Generation
Symbol Table Manager
Error Handler

Symbol Table:

-- Data Structure

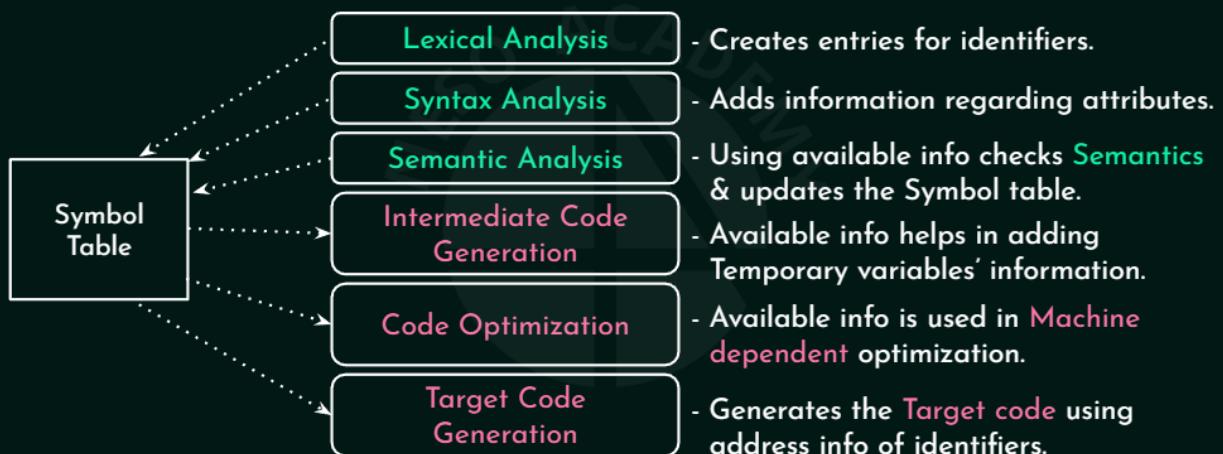
- Variable & Function names
- Objects
- Classes
- Interfaces



Symbol Table: Lexical Analysis Syntax Analysis Semantic Analysis Intermediate Code Generation Code Optimization Target Code Generation
Symbol Table -- Data Structure- Variable & Function names- Objects- Classes- Interfaces

Symbol Table – Usage by Phases

Usage



Symbol Table - Usage by Phases Lexical Analysis Syntax Analysis Semantic Analysis Intermediate Code Generation Code Optimization Target Code Generation
Symbol Table 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

```
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 - Entries
Name Type Size Dimension Line of Declaration Line of Usage Address
char x[] = "NESO ACADEMY"; int count; count int 20 ----- x char 121 -----

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()

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()



Compiler Design

Symbol Table – Solved PYQs

Compiler Design Symbol Table - Solved PYQs



Outcome

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

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

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

- (A) Three Address Code
- (B) Abstract Syntax Tree (AST)
- (C) Symbol Table**
- (D) Control Flow Graph (CFG)

GATE 2021

Q1:In the context of compilers, which of the following is/are NOT an intermediate representation of the Source program?
(A) Three Address Code
(B) Abstract Syntax Tree (AST)
(C) Symbol Table
(D) Control Flow Graph (CFG)
GATE 2021

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

- (A) Linear List
- (B) Search Tree**
- (C) Hash Table
- (D) None of these

ISRO 2016

Q2:Access time of the Symbol table will be logarithmic if it is implemented by
(A) Linear List
(B) Search Tree
(C) Hash Table
(D) None of these
ISRO 2016

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 $O(n)$.

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

Compiler Design

Introduction to Lexical Analyzer

Compiler Design Introduction to Lexical Analyzer

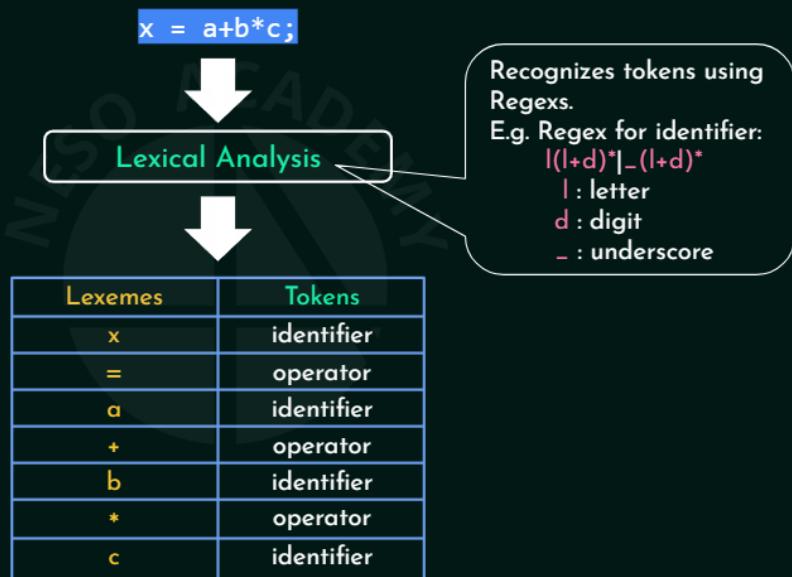


Outcome

★ Working principle of Lexical Analyzer

Outcome★Working principle of Lexical Analyzer

Lexical Analyzer:



Lexical Analysis

Lexical Analyzer

x =

a+b*c;

Lexemes

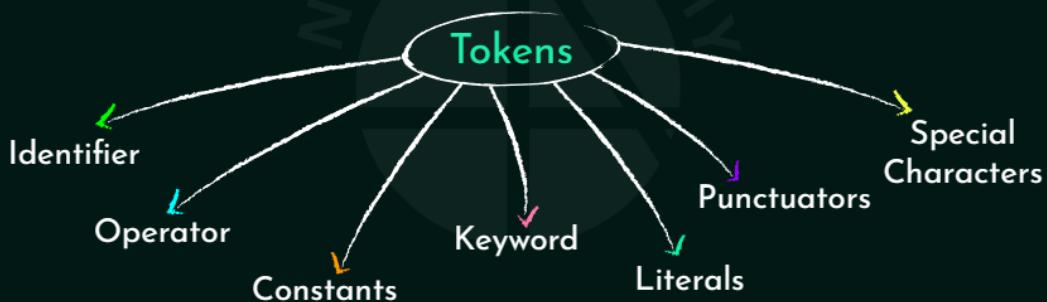
identifier=operatoridentifier+operatoridentifier*operatoridentifier

Recognizes tokens using Regexes. E.g.

Regex for identifier: $l(l+d)^*|_(l+d)^*$

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:

- 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:

- 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.
Scanning
Analyzing
Eliminate Non-Token Elements
Lexemes
Tokens

Lexical Analyzer:



C-Tokens:

- **if:**

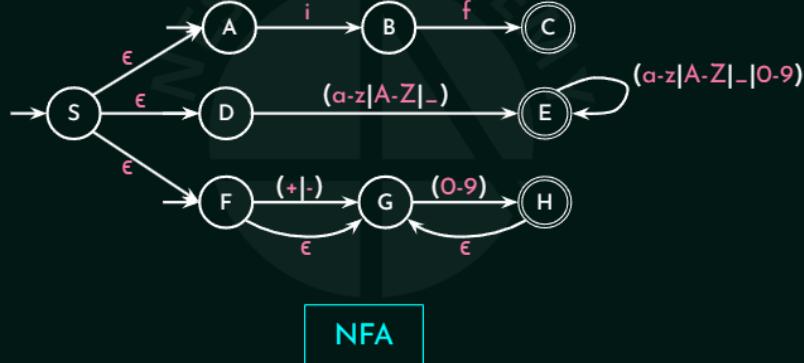
A state diagram for the 'if' token. It starts at state A, moves to B on input 'i', and to C on input 'f'. State C is a final state.
- **Identifier:**

A state diagram for identifiers. It starts at state D, moves to E on inputs '(a-z|A-Z|_)'. State E is a final state and has a self-loop for inputs '(a-z|A-Z|_)'.
- **Integer:**

A state diagram for integers. It starts at state F, moves to G on inputs '(+|-)', and to H on inputs '(0-9)'. State H is a final state. There are transitions from G to F labeled with the empty string (ϵ), and from G to H labeled with the empty string (ϵ).

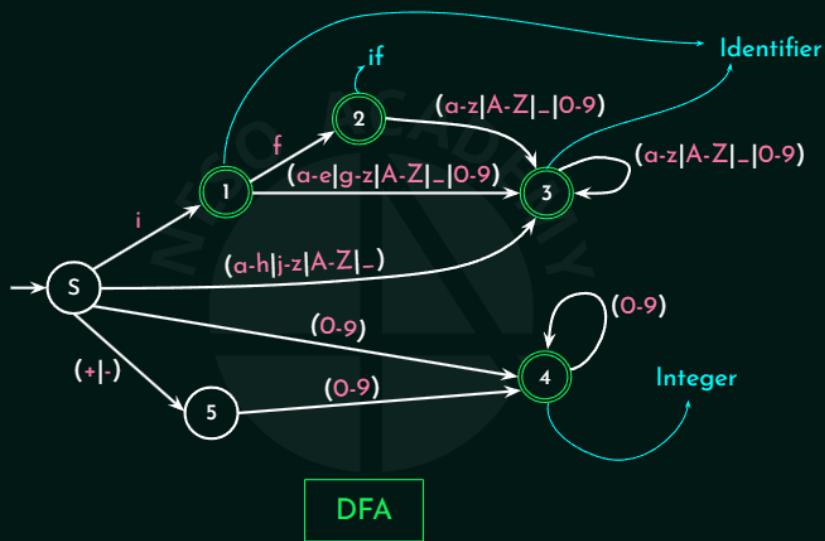
Lexical Analyzer:
Scanning
Analyzing
Eliminate Non-Token Elements
Lexemes
Tokens
C-Tokens:
• if:
• Identifier:
• Integer:
ABDiCfE(a-z|A-Z|_)(a-z|A-Z|_0-9)FG(+|-)H(0-9) $\epsilon\epsilon$

C-Tokens:



C-Tokens:ABDiCfE(a-z|A-Z|_)(a-z|A-Z|_|0-9)FG(+|-)H(0-9) $\epsilon\epsilon\epsilon\epsilon$ NFA

C-Tokens:



1SC-Tokens:2f3(a-e|g-z|A-Z|_|0-9)(a-z|A-Z|_|0-9)5(+|-)4(0-9)iDFA(a-h|j-z|A-Z|_)(0-9)(a-z|A-Z|_|0-9)ifIdentifierInteger(0-9)

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.
Scanning
Analyzing
Eliminate Non-Token Elements
Lexemes
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.
Uses DFA for pattern matching!
NFA
DFA
AmDFA

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/* Multiline 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.

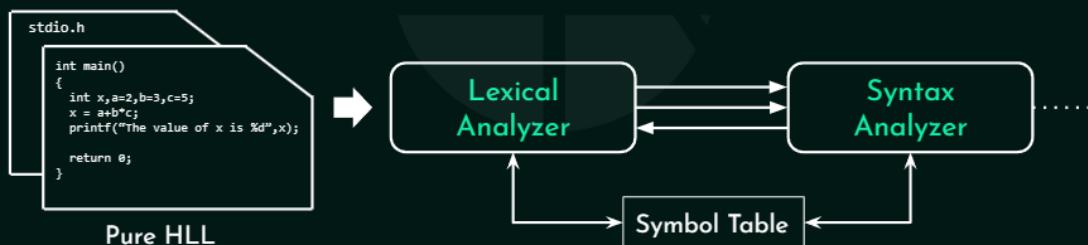
```
// Single line comment  
/* Multi  
line  
Comment*/
```

'.'	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. // Single line comment/* Multiline 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.



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.

Pure HLL
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;
}

Lexical Analyzer
Symbol Table
Syntax Analyzer

Compiler Design

Lexical Analyzer-Tokenization

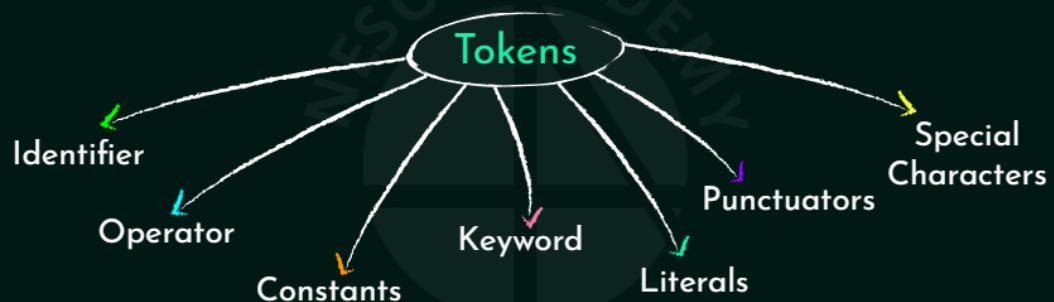


Outcome

- ☆ Count the number of tokens in a given code segment

Outcome ☆ Count the number of tokens in a given code segment

Lexical Analyzer-Tokenization:



Lexical Analyzer-Tokenization: Tokens Identifier 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;}

Lexical Analyzer-Tokenization:

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"

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

    return 0;
}
```

Count: 39

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:
printf,2.Identifier: main,x,a,b,c,4.Operator: =+,*,6.Literal: "The value of x is %d"1.Keyword: intreturn,5.Constant: 23,5,0,3.Punctuator: (),{,,;},Count:39



Compiler Design

Lexical Analyzer – Solved Problems (Set 1)

Compiler Design Lexical Analyzer - Solved Problems (Set 1)



Outcome

- ☆ Three Solved questions on Lexical Analyzer.

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?

- (A) Finite state automata
- (B) Deterministic pushdown automata
- (C) Non-Deterministic pushdown automata
- (D) Turing Machine

GATE 2011

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?
(A) Finite state automata
(B) Deterministic pushdown automata
(C) Non-Deterministic pushdown automata
(D) Turing Machine
GATE 2011

Q2: The output of a lexical analyzer is

ISRO 2017

- (A) A parse tree
- (B) Intermediate code
- (C) Machine code
- (D) A stream of tokens

Q2: The output of a lexical analyzer is
(A) A parse tree
(B) Intermediate code
(C) Machine code
(D) A stream of tokens
ISRO 2017

Q3: The number of tokens in the following C statement is
printf("i=%d, &i=%x", i, &i);

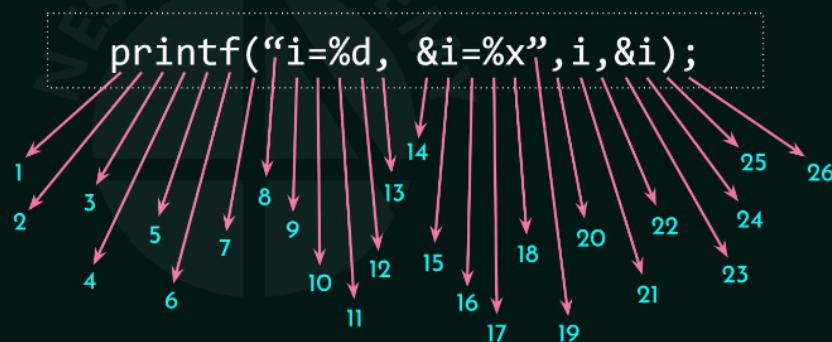
GATE
2000

- (A) 3
- (B) 26
- (C) 10**
- (D) 21

Q3: The number of tokens in the following C statement is
printf("i=%d, &i=%x", i, &i);

GATE
2000

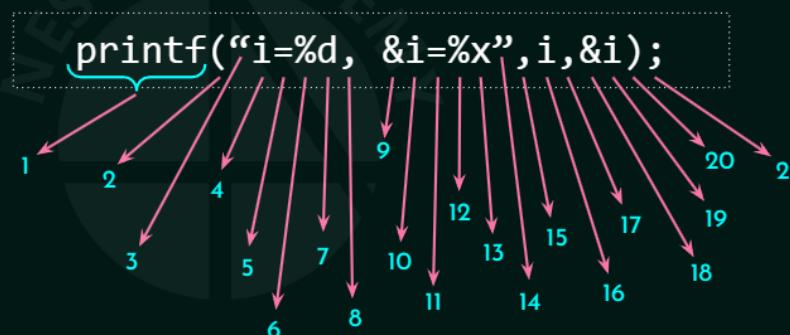
- (A) 3
- (B) 26
- (C) 10**
- (D) 21



Q3: The number of tokens in the following C statement is
`printf("i=%d, &i=%x",i,&i);`

GATE
2000

- (A) 3
- (B) 26
- (C) 10**
- (D) 21



The number of tokens in the following C statement is(A) 3(B) 26(C) 10(D) 21GATE 2000 printf("i=%d, &i=%x",i,&i);Q3:printf("i=%d,
&i=%x",i,&i);123456710111213141516171819892021



Compiler Design

Lexical Analyzer – Solved Problems (Set 2)



Outcome

- ☆ Two 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

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 analysisGATE 2011 NIELIT Scientist-B2017

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) T₁T₂T₃ (B) T₁T₁T₃ (C) T₂T₁T₃ (D) T₃T₃

GATE 2018

Q2: A lexical analyzer uses the following patterns to recognize three tokens T₁, T₂, and T₃ over the alphabet {a,b,c}. (A) T₁T₂T₃ (B) T₁T₁T₃ (C) T₂T₁T₃ (D) T₃T₃ GATE 2018 T₁: a? (b|c)*a T₂: b? (a|c)*b T₃: 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?

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) T₁T₂T₃ (B) T₁T₁T₃ (C) T₂T₁T₃ (D) T₃T₃

Q2: A lexical analyzer uses the following patterns to recognize three tokens T₁, T₂, and T₃ over the alphabet {a,b,c}. (A) T₁T₂T₃ (B) T₁T₁T₃ (C) T₂T₁T₃ (D) T₃T₃ T₁: a? (b|c)*a T₂: b? (a|c)*b T₃: 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? (b|c)*a + a(b|c)*a(a|c)*b + b(a|c)*b (b|a)*c + c(b|a)*c*****

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

$$\begin{aligned}
 T_1: a? (b|c)^*a &\longrightarrow (b|c)^*a + a(b|c)^*a \\
 T_2: b? (a|c)^*b &\longrightarrow (a|c)^*b + b(a|c)^*b \\
 T_3: c? (b|a)^*c &\longrightarrow (b|a)^*c + c(b|a)^*c
 \end{aligned}$$

(A) T₁T₂T₃

bbaacabc

 T₁ T₂ T₃

Q2: A lexical analyzer uses the following patterns to recognize three tokens T₁, T₂, and T₃ over the alphabet {a,b,c}.(A) T₁T₂T₃T₁: a? (b|c)*aT₂: b? (a|c)*bT₃: c? (b|a)*c***bbaacabcT₁T₂T₃(b|c)*a + a(b|c)*a(a|c)*b + b(a|c)*b (b|a)*c + c(b|a)*c*****

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

$$\begin{aligned}
 T_1: a? (b|c)^*a &\longrightarrow (b|c)^*a + a(b|c)^*a \\
 T_2: b? (a|c)^*b &\longrightarrow (a|c)^*b + b(a|c)^*b \\
 T_3: c? (b|a)^*c &\longrightarrow (b|a)^*c + c(b|a)^*c
 \end{aligned}$$

(B) T₁T₁T₃

bbaacabc

 T₁ T₁ T₃

Q2: A lexical analyzer uses the following patterns to recognize three tokens T₁, T₂, and T₃ over the alphabet {a,b,c}.(B) T₁T₁T₃T₁: a? (b|c)*aT₂: b? (a|c)*bT₃: c? (b|a)*c***bbaacabcT₁T₃(b|c)*a + a(b|c)*a(a|c)*b + b(a|c)*b (b|a)*c + c(b|a)*c*****T₁

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

$$\begin{array}{lll} T_1: a? (b|c)^*a & \xrightarrow{\hspace{2cm}} & (b|c)^*a + a(b|c)^*a \\ T_2: b? (a|c)^*b & \xrightarrow{\hspace{2cm}} & (a|c)^*b + b(a|c)^*b \\ T_3: c? (b|a)^*c & \xrightarrow{\hspace{2cm}} & (b|a)^*c + c(b|a)^*c \end{array}$$

(C) T₂T₁T₃

bbaacabc

 T₂ T₁ T₃

Q2:A lexical analyzer uses the following patterns to recognize three tokens T₁, T₂, and T₃ over the alphabet {a,b,c}.(C) T₂T₁T₃T₁: a? (b|c)*aT₂: b? (a|c)*bT₃: c? (b|a)*c***bbaacabc(b|c)*a + a(b|c)*a(a|c)*b + b(a|c)*b (b|a)*c + c(b|a)*c*****T₂T₃T₁

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

$$\begin{array}{lll} T_1: a? (b|c)^*a & \xrightarrow{\hspace{2cm}} & (b|c)^*a + a(b|c)^*a \\ T_2: b? (a|c)^*b & \xrightarrow{\hspace{2cm}} & (a|c)^*b + b(a|c)^*b \\ T_3: c? (b|a)^*c & \xrightarrow{\hspace{2cm}} & (b|a)^*c + c(b|a)^*c \end{array}$$

(D) T₃T₃

bbaacabc

 T₃ T₃

Q2:A lexical analyzer uses the following patterns to recognize three tokens T₁, T₂, and T₃ over the alphabet {a,b,c}.(D) T₃T₃T₁: a? (b|c)*aT₂: b? (a|c)*bT₃: c? (b|a)*c***bbaacabcT₃(b|c)*a + a(b|c)*a(a|c)*b + b(a|c)*b (b|a)*c + c(b|a)*c*****T₃

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?

bbaacabc
 $T_1 \quad T_2 \quad T_3$

bbaacabc
 $T_1 \quad T_1 \quad T_3$

bbaacabc
 $T_2 \quad T_1 \quad T_3$

bbaacabc
 $T_3 \quad T_3$

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?
bbaacabcT₃T₃bbaacabcT₁T₃T₁bbaacabcT₁T₂T₃bbaacabcT₂T₃T₁



Compiler Design

Errors and Error-Recovery in Lexical Analysis

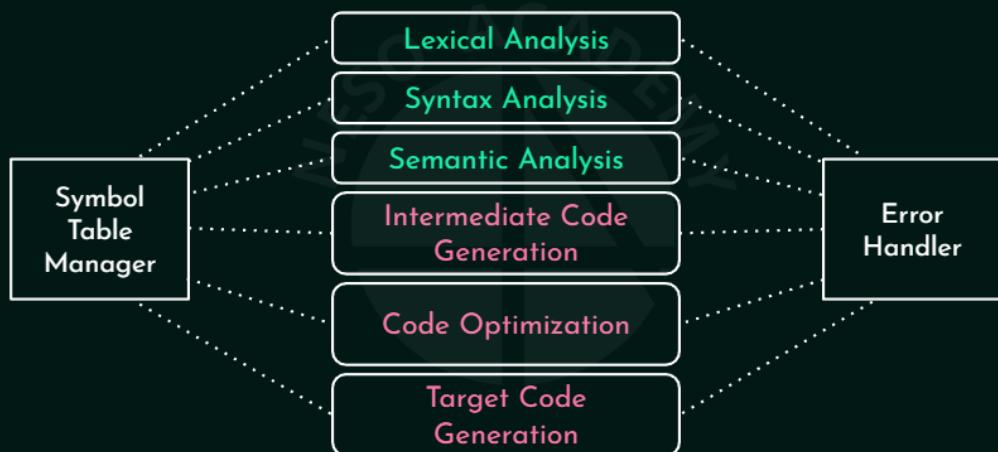


Outcome

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

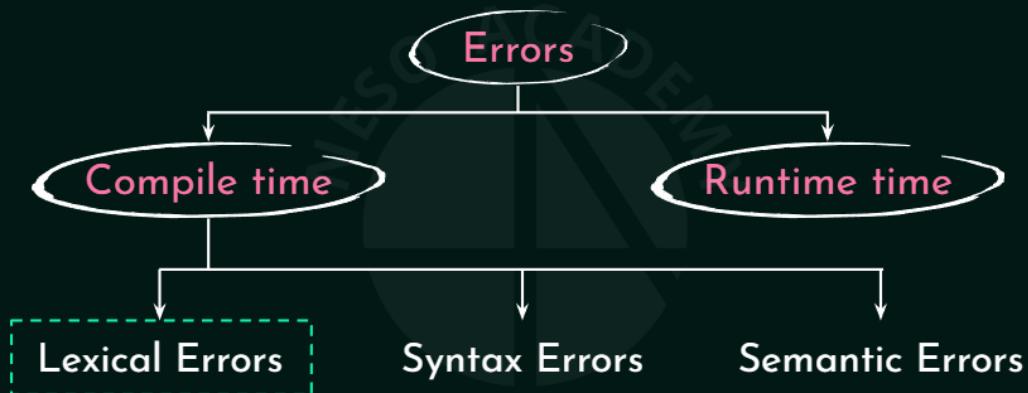
Outcome
☆Role of Error handler, especially for Lexical Analysis.
☆Types of Error.
☆Different types of Lexical Errors.
☆Error Recovery in Lexical Analysis.

Compiler – Internal Architecture



Compiler - Internal Architecture
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Target Code Generation
Symbol Table Manager
ErrorHandler

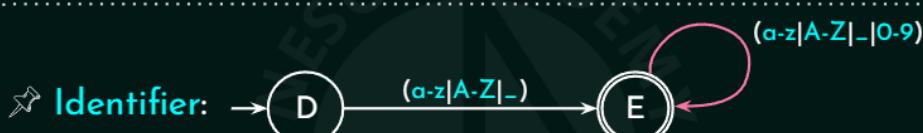
Classification of Errors:



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

Lexical Errors:

- Identifiers that are way too long.



: 31/247



: 2048



: 79

Lexical Errors:
• Identifiers that are way too long.
D(a-z|A-Z|_)E(a-z|A-Z|_|0-9)
Identifier:: 31/247: 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. 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. 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 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 ANESO;
```

int 4NESO; Lexical Error-Recovery:
•Panic-mode recovery.

Lexical Error-Recovery:

- Panic-mode recovery.

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

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. 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.



Lexical Error-Recovery:
•Panic-mode recovery.
•Transpose of two adjacent characters.
•Insert a missing character.
it NESO;int NESO;

Lexical Error-Recovery:

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



Lexical Error-Recovery:
•Panic-mode recovery.
•Transpose of two adjacent characters.
•Insert a missing character.
•Delete an unknown or extra character.
intt 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.



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.