COMPILER DESIGN
NESO ACADEMY

Introduction

0, 1 → Binary

Paper tape, Punched card → hole indicates 0. and absence means 1.

Punched card

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

C - 1000011
a - 1100001
s - 1110010
d - 1110011

ASCII

Language Translator

Language Translators :

(i) Assembler

```
MOV R1, 02H          Assembler          01101001010 10
MOV R2, 03H    ------------------>       00101010100 10
ADD R1, R2                              0100 111110010 1
STORE X, R1                             0101010101010 1
```

[Assembly language] [Machine code]

(ii) Interpreter :
one line at a time
eg Php, Python, Ruby, JS

(iii) compiler :
all at a time
eg C, C++, Erlang

(< (speed))

__/__/__

C Language : Middle Level Language

✓ Direct Memory Access through pointers.
✓ Bit Manipulation through Bitwise operator
✓ Writing Assembly code within "C code

But initially : 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 code]

Source code / HLL code → Language Translator → Machine code

Internal Architecture of Language Translator

① Preprocessor ② compiler
③ Assembler ④ Linker / Loader

① Preprocessor

* removes preprocessing directives
* removes all comments

o/p :

```
stdio.h
int main()
{;
}
```

[comments are removed]

↓ pure HLL

__/__/____

② compiler
↳ Generates the equivalent assembly code for the given HLL code.

Pure HLL $\xrightarrow{\text{compiler}}$ Assembly Language

③ Assembler
↳ Generates relocatable machine code for the given assembly language code.

Assembly Language $\xrightarrow{\text{Assembler}}$ Relocatable Machine code

Explanation of "relocatable":
A program turns into a process only during execution. only during execution, it is allocated to the main memory. Before execution, we cannot know where exactly in the RAM the process will be allocated. so for the time-being, the sequence of machine codes is given a relocatable address like i, i+1, i+2 ....

④ Linker / Loader
↳ Generate the absolute machine code that is actually executable.

i+0 : 001010101001 → 0x000004B8 : 001010101001
i+1 : 010110100110 → 0x000004B9 : 010110100110

Relocatable Machine code → Absolute Machine code

↓

process loaded into RAM and ready for execution

__/__/__

Internal Architecture of compiler :

Analysis Phase
1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis

Synthesis Phase
4. Intermediate code Generation
5. code optimization
6. Target code Generation

Front-End : 1, 2, 3, 4
Back-End : 5, 6

Backend : Varies with OS.
eg if a C compiler is already designed for Windows, and we want to build a new Mac compiler → only backend needs to be modified !

other components :
1. Symbol Table Manager
2. Error Handler

✓ Information gathered in analysis are stored in the symbol table and used by the synthesis phase.

✓ The Error Handler deals with error detection and recovery.

Syllabus :

Pre-requisite : Theory of computation

- ✓ Introduction
- ✓ Syntax Analyzer
- ✓ Top Down Parsers
- ✓ Bottom Up parser
- ✓ Syntax Directed Translation Schemes
- ✓ Intermediate code Generation
- ✓ Runtime Environment & code optimization

Different Phases of compiler

outcome:

1. overview of various phases of compiler
2. tools to implement different phases

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

↳ visualize how this statement is converted into assembly language through all phases of the compiler. (5)

(1) lexical analyzer

x = a + b * c ;

↓

Lexical analyzer

↓

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

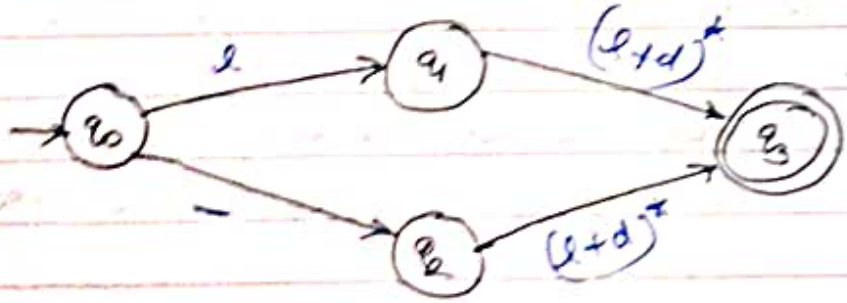
tokens → meaning of the given lexemes

Lexems: similar to words with only one small difference words have individual meaning. But group of lexems convey meaning in their entirety

token recognition is done through regex.

Regex for identifier $l(l+d)^* \mid \_(l+d)^*$

l → letter d → digit _ → underscore



② Syntax Analyzer

Depends on context-free grammars

$x = a + b * c;$

These are the rules that the parser will use to form the parse tree

Production

$S \rightarrow id = E;$

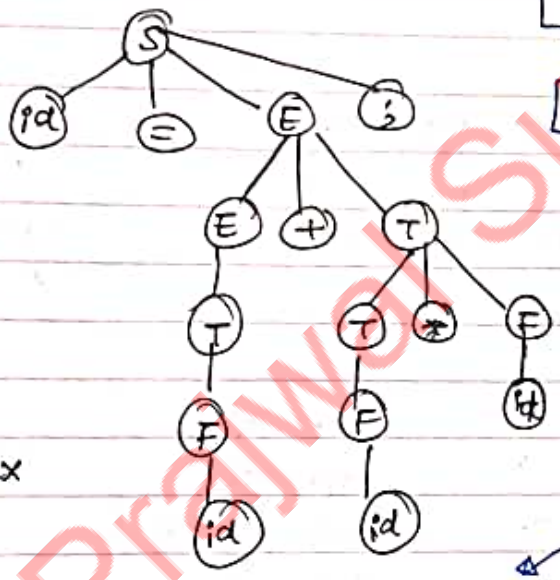
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

id, =, ;, +, * → terminals

S, E, T, F → non-terminals



Traversal: top to bottom, left to right

Yield of parse tree:

id = id + id * id ;

(If not, there is some syntax error in the statement.)

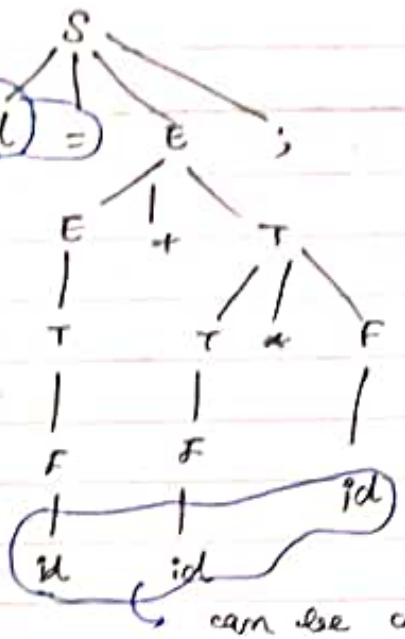
since the yield of the parse tree and the expression are the same, the syntax analyzer will not produce any errors.

③ Semantic Analyzer

Parse Tree $\xrightarrow{SA}$ semantically verified parse tree

semantic Analyzer is responsible for:

* type checking
* error bound checking
* correctness of scope resolution



(cannot be a constant as it is followed by an = operator)

can be constants

Semantic analyzer detects
- ✓ type mismatch errors
- ✓ undeclared variable
- ✓ misuse of reserved words
- ✓ multiple declaration of variables within a single scope
- ✓ accessing an out of scope variable
- ✓ mismatch between actual & formal parameters

Semantic Analyzer checks for "meaningfulness" of the parse tree and verifies that.

④ Intermediate code Generator :

semantically verified parse Tree → Intermediate code

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

① t0 = b * c;
② t1 = a + t0;
③ x = t1;

TAC
Three Address code
[At most 3 addresses of variables are referenced in one line]

Till this phase
↓
FRONT-END

After this backend : To generate target code specific to a platform, we need to only modify the next 2 phases.

⑤ code optimizer

Intermediate code → (code optimizer) → optimized code

code optimization can be

✓ Machine Dependent (OR) ✓ Machine Independent

① t0 = b * c;
② t1 = a + t0;
③ x = t1;

→ (merge)

① t0 = b * c;
② x = a + t0;

⑥ Target code Generator

optimized code → (Target code Generator) → Assembly code segment

3 Address code
① t0 = b * c;
② x = a + t0;

→ assembly level code

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

↳ mov: moving

eax: extended version of ax register, a combination of ah and al registers

eax stores 32 bits { ah: stores 16 higher order bits
al: stores remaining 16 lower order bits

↳ eax: accumulator register

DWORD PTR: data word pointer, scaling factor -8

a : %rbp - 4
b : %rbp - 8
c : %rbp - 12
x : %rbp - 16

imul → pneumonic for signed multiplication

edx → dh + dl
↳ (another accumulator register)

Tools for Practical Implementation

LEX → Lexical analysis phase
standard Lexical Analysis Generator in Unix Based systems. It reads an input stream specifying the lexical analyzer and writes the source code which implements the lexical analyzer for the same. C programming language.

YACC → Yet Another compiler compiler
↳ Syntax Analysis Phase
It is an LA-LR parser generator. It is commonly used with LEX to implement syntax analysis.

LANCE compiler → The entire front-end for a C language for an embedded processor.

Summary

- ✓ overview of various phases of compiler
- ✓ Tools to implement different phases.

Symbol Table

outcome:

- ✓ Usage of symbol table by various phases
- ✓ Entries of symbol table
- ✓ operations on symbol table

Symbol Table:

- ✓ Data structure which is created and maintained by compilers in order to store information about the occurrence of various entities such as:
  - → variables and function names
  - → objects
  - → classes
  - → Interfaces

Symbol Table - usage by Phases

1) Lexical Analysis
   - creates entry for identifiers

   Lexical Analyzer is a scanner. It scans the entire source code line by line. During the scanning, whenever it encounters any identifier, it creates an entry for that in the symbol table.

2) Syntax Analysis
   - Adds information regarding attributes like type, scope, dimension, line of reference, line of usage, etc

3) Semantic Analysis
↳ Using available information, checks semantics and updates the symbol table.

4) Intermediate code Generation
↳ Available information helps in adding temporary variables' information.

5) code optimization
↳ Available information is used in Machine Dependent optimization.

6) Target code Generation
↳ Generates the target code using address information of identifiers.

Symbol Table - Entries

① Name - stores name of identifiers
② Type - stores datatype of identifiers
③ size - specifies size of the identifier
④ Dimension - for 1D and multi-dimensional arrays, it stores dimension; for primitive datatypes, value = 0
⑤ Line of declaration - line number of the source code where the identifier has been declared is stored
⑥ Line of usage - line number of the source code where the identifier has been used is stored
[If used in many places, a linked list representation is used]
⑦ Address - stores address info of the identifier

(platform dependent)

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

Name	Type	size	Dimension	LOD	LOU	Address
count	int	2	0	-	-	-
x	char	12	1	-	-	-

As none of the attributes are of a fixed size, it is not possible to know how much space is required for the symbol table before it is actually created.

↓

If we choose a size which turns out to be smaller, we will be able to save the space wastage, however we won't be able to store all the entries.

↓

On the other hand, a bigger size will lead to wastage of space.

↓

considering these circumstances, the best soln would be to dynamically allocate the size of the symbol table during compile time

Symbol Table - operations

① Non-Block structured Language: eg Fortran

contains single instance of the variable declaration.

operations: Insert() Lookup()

It is continued particularly because of the use of unstructured control flow using goto statements.

② Block structure languages
- variable declaration may happen multiple times
- operations
  Insert() Lookup() set() Reset()

summary
- ✓ Usage of symbol table by various phases
- ✓ Entries of symbol table
- ✓ operations on symbol table

Symbol Table - solved PYQs

outcome :
- ✓ GATE 2021 Q on ST
- ✓ ISRO 2016 Q on ST

GATE 2021

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)

Intermediate code can be of 2 forms :
linear form AND the Tree form
↓ 3AC ↓ AST

CFG : It is a representation using Graph notation of all the paths that might be travelled through a program during its execution.

__/__/__

ISRO 2016

Access Time of the symbol table will be logarithmic if it is implemented by:

(A) Linear List (C) Hash Table

✓(B) Search Tree (D) None of these

Symbol Table - various Implementations

Implementation	Insertion Time	Lookup Time	Disadvantage
(A) Linear Lists			
(i) ordered Lists			i) For ordered lists, every insertion is preceded by lookup operation
a) Arrays	O(n)	O(log n)	
b) Linked Lists	O(n)	O(n)	ii) Access time & table size
(ii) Unordered Lists	O(1)	O(n)	
(B) search tree	$O(\log_m n)$	$O(\log_m n)$	Always needs to be balanced
(C) Hash table	O(1)	a)	too many collisions increases time/complexity to O(n) (linked list)

Summary

✓ GATE 2021 ✓ ISRO 2016

Introduction to Lexical Analyzer

outcome: ✓ working principle of Lexical Analyzer

Lexical Analyzer:

✓ scans the pure HLL code line by line

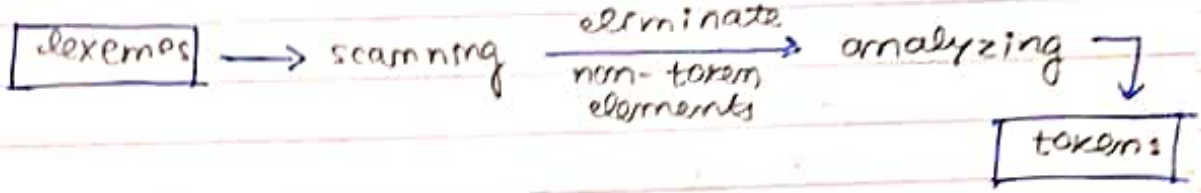
✓ takes lexemes as I/p and produces tokens

(uses DFA for pattern matching)

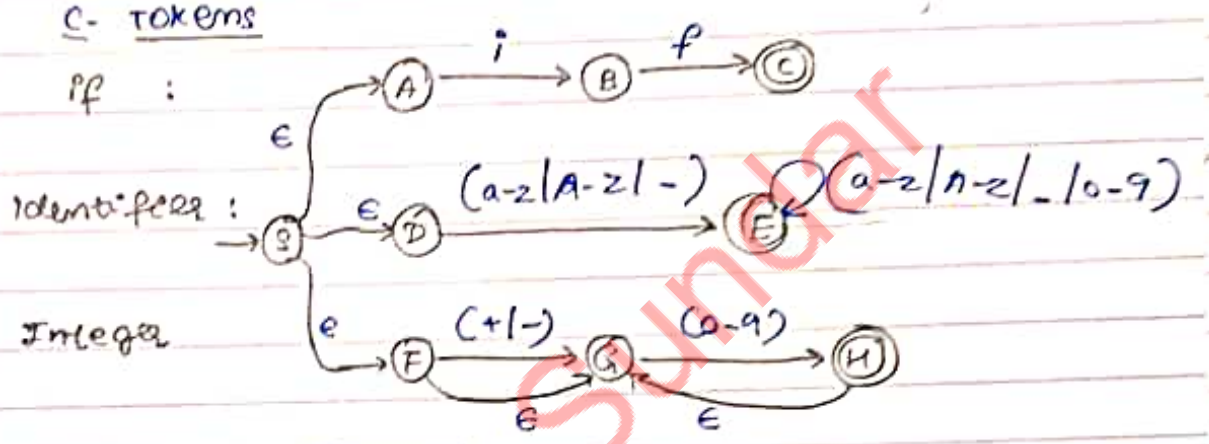
Tokens

1) identifier 2) operator 3) constants 4) keyword 5) literals 6) punctuators 7) special characters

2 functions take place in the lexical analyzer : ① scanning ② analyzing



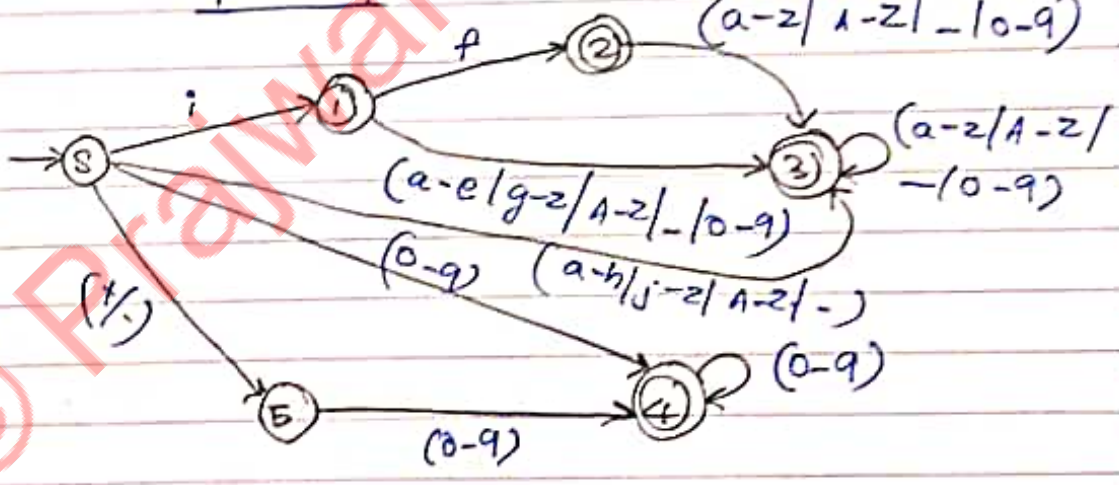
C- tokens



(DFA is implemented in the Analyzing phase)

equivalent DFA

ε-NFA



② identifies if
① and ③ identify identifiers
④ identifies integers

NFA → DFA → mDFA
(minimized DFA)

More features of Lexical analyzer.

✓ removes comments and whitespaces from the pure HLL code.

// single line comment

/* Multi - line
comment */

int NE /* its a comment */ SO;

↳ int NE SO;

lexical analyser removes the comment & replaces it with a blank whitespace. During semantic analysis, an error will be generated → so will be reported as an undeclared variable. However, for the lexical analyser NE and SO will be treated as 2 different tokens.

Whitespaces :

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

['\v' = 6 × '\n', '\f' → page breaking ASCII character]
'\r' → space created by the ENTER key

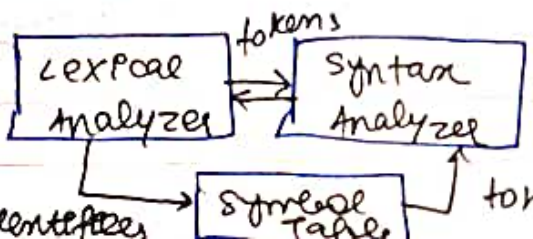
↓

All these are recognized and eliminated during the scanning phase of the lexical Analyser

Summary
✓ working principle of LA

Features

✓ helps in macro expansion in the pure HLL code.



LA hands tokens to SA. while SA parses a token, it requests for another from the LA

Lexical Analyzer - Tokenization

Outcome

✓ count the no. of tokens in a given code segment

```
int main()
{
    int x, a = 2, b = 5, c = 5;
    x = a + b * c;
    printf("The value of x is %d", x);
    return 0;
}
```

Tokens:
① keyboard : int → ①
② Identifier : main → ①
③ Punctuation : ( ) { } → ③
④ keyword : int → ①
⑤ Identifier : x → ①
⑥ Punctuator : , → ①
⑦ Identifier : a → ①
⑧ operator : = → ①
⑨ constant : 2 → ①
⋮

counting → 39

Summary :

① keywords : 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"

Note : Every occurence of the token is counted by the lexical analyzer, no matter how many time it occurs in the pure HLL code.

Summary : ✓ count no. of tokens

Lexical Analyzer - solved Problems - set 1

outcome

✓ 3 solved questions on Lexical Analyzer

Q1 GATE 2011

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

(A) Finite State Automata

(B) Deterministic Push Down Automata

(C) Non-Deterministic Push Down Automata

(D) Turing Machine

D-PDA and ND-PDA are the acceptors of D-CFL and ND-CFL [context free language]

↓

CFLs are generated from CFGs [context free grammars]. During the different phases of compilers, we have observed that in the syntax analysis phase, the syntax analyzer makes use of the CFGs in the construction of the parse trees.

N-PDAs have more expressing power than PDAs

↓

Also, since compilers are implemented on physical systems, TM can be designed that can map the HLL string into assembly language target code. So Turing Machine has the power to implement the entire compiler itself.

Q2 ISRO 2017

The output of a lexical analyzer is :

(A) A parse tree (B) Intermediate code

(C) Machine code ✓(D) stream of tokens

Q3 GATE 2000

The number of tokens in the following C statement is

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

(A) 3 (B) 26 ✓(C) 10 (D) 21

(A) 3 → no. of arguments

(B) 26 → counting each token in string

(C) 10 → correct ans; " " is one token

(D) 21 → counting printf as one string + each token in string

summary :

✓ 3 solved questions on Lexical Analyzer

Lexical Analyzer (solved Problems) - set 2

outcome :

✓ 2 solved questions on Lexical Analyzer

Q1 GATE 2011 NIELIT Scientist-B 2017

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

DF analysis is performed during code optimization on the program flow graph. So, it will not recognize tokens, rather it analyses the flow control of the tokens.

Q2 : GATE 18

A lexical analyzer uses the following patterns to recognize 3 tokens T_1, T_2, T_3 over the alphabet $\{a, b, c\}$.

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

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

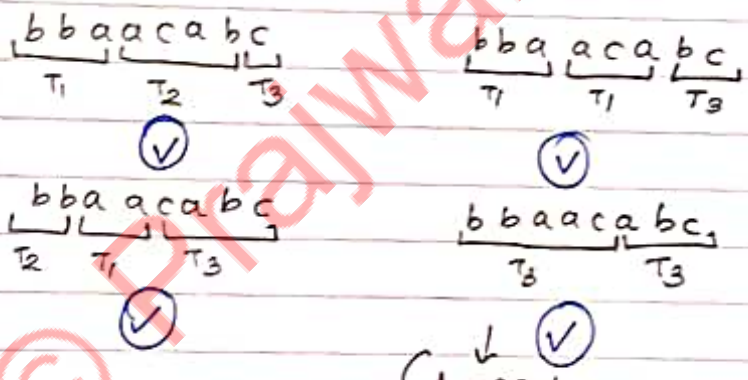
T_3 : $c\,?\,(b|a)^*\,c$

[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]

GATE 2018

which one of the following is the sequence of tokens output by the analyzer, if the string "bbaacabc" is processed ?

(A) $T_1\,T_2\,T_3$ (B) $T_1\,T_1\,T_3$ (C) $T_2\,T_1\,T_3$ (D) $T_3\,T_3$



summary

✓ 2 solved Q on LA

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

Error Handler :

- ✓ Error detection
- ✓ Error reporting [generating error reports to the user]
- ✓ Error Recovery [implementation of some recovery strategy for handling errors].

Types of Errors :

① Run-Time Errors : These take place during the execution of the program. code. Eg:- Insufficient available memory space, unexpected. errors, etc are the examples of run time errors.

② Compile-Time Errors : These occur during compile time, i.e. before execution of the program. There are 3 types of compile-time errors : ✓ lexical errors ✓ syntax errors ✓ semantic errors

Lexical Errors :

✓ Identifiers that are way too long
eg C: ANSCI allows 6 significant characters from the identifier's names. external identifiers are the identifiers that are used in macros.
Anyway ANSCI allows :
31 significant characters → internal identifiers
which are basically declared in a function block.
on the other hand microsoft c compilers allow internal identifiers compilers to have upto 247 significant characters

__/__/____

C++ For C++, both Intel and Microsoft compilers allow upto 2048 significant characters for the identifiers.

Python In python, identifiers are allowed to have upto 79 significant characters.

✓ Exceeding length of numeric constants.

int i = 45678 91;

size 2 Bytes : −32768 to 32767 only

✓ Numeric constants which are ill-formed.

int i = 4567 $91;

✓ Illegal characters that are absent from the source code.

char x[] = "NESO ACADEMY"; $

Lexical - Error - Recovery

① Panic - Mode Recovery

↳ It is the most basic way of recovery. In this, once an error has been encountered, successive characters are skipped until a valid delimeter is found.

int 4 NESO; → delimeter

error skipped

while (condition)
{
___
___
___
} → delimeter

For blocks, } is the delimeter.

Ignore in b/w → "panic mode" recovery

② Transpose of 2 adjacent characters

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

unoin → wrong positions

union ✓ valid keyword

↓

Error handler transposes o↔i and forms the correct keyword token union.

③ Insert a missing character

it NESO; → int NESO;

④ Deleting an unknown or extra character

intt NESO; → int NESO;

⑤ Replacing one character with another.

itt NESO; → int NESO;

Therefore, these are the various error recovery strategies implemented by the error handler during the lexical analysis phase

Summary :

- ✓ Role of Error Handler, especially for LA
- ✓ Types of Error.
- ✓ Different Types of Lexical Errors
- ✓ Error Recovery in LA.

— X —