Unit III :
Introduction To Compilers (TOPICS)

Phase structure of Compiler and entire compilation process

Lexical Analyzer: The Role of the Lexical Analyzer, Input Buffering.

Specification of Tokens, Recognition Tokens,

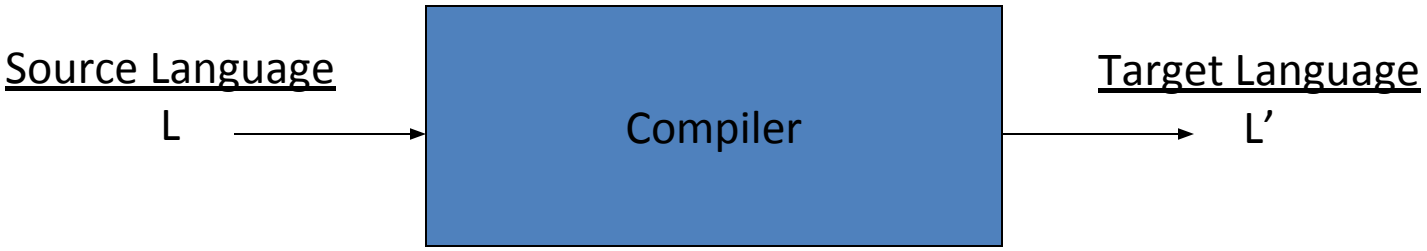
Design of Lexical Analyzer using Uniform Symbol Table,

Lexical Errors.

LEX : LEX Specification, Generation of Lexical Analyzer by LEX.

Phase Structure of Compiler

What is a Compiler?

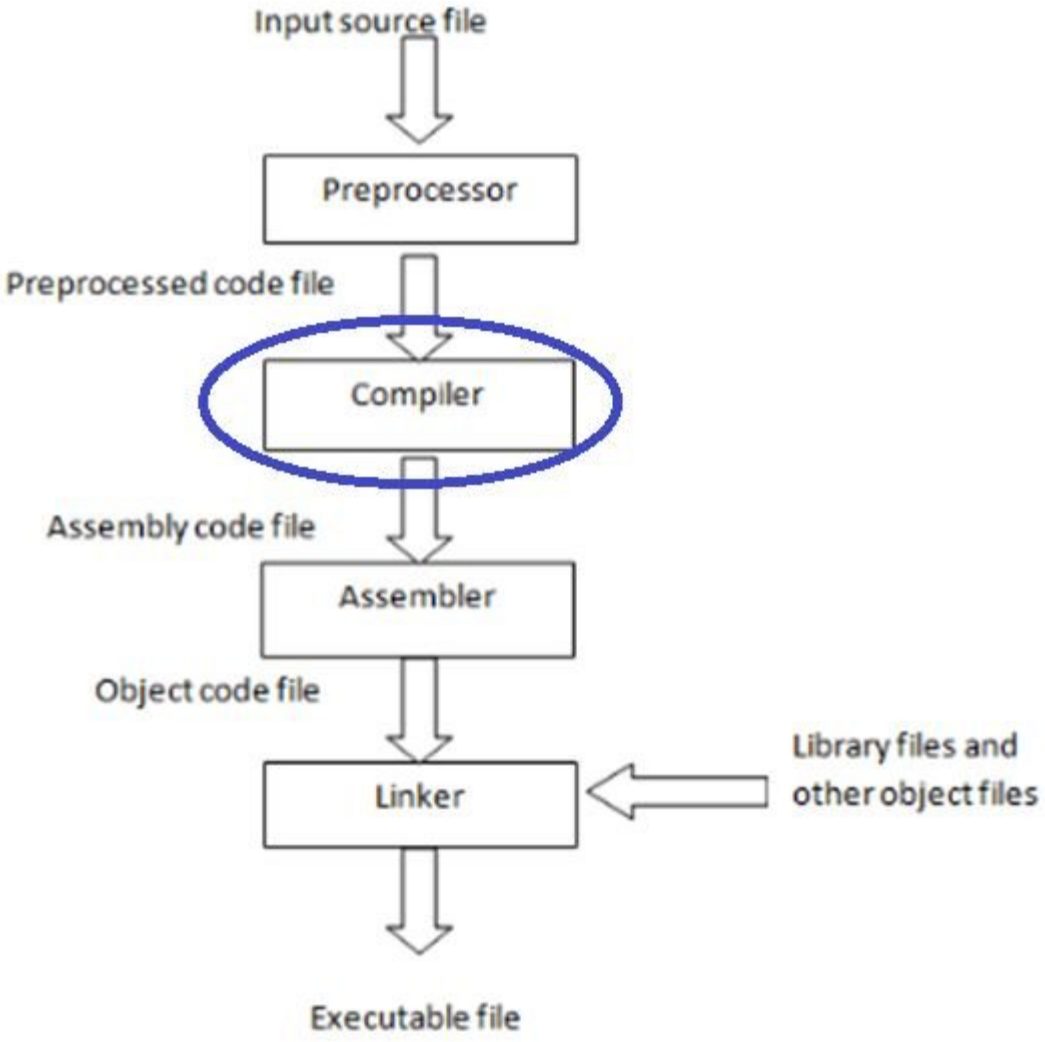


Java
C++
FORTRAN

JVM
Intel Pentium
MIPS
IBM's CELL

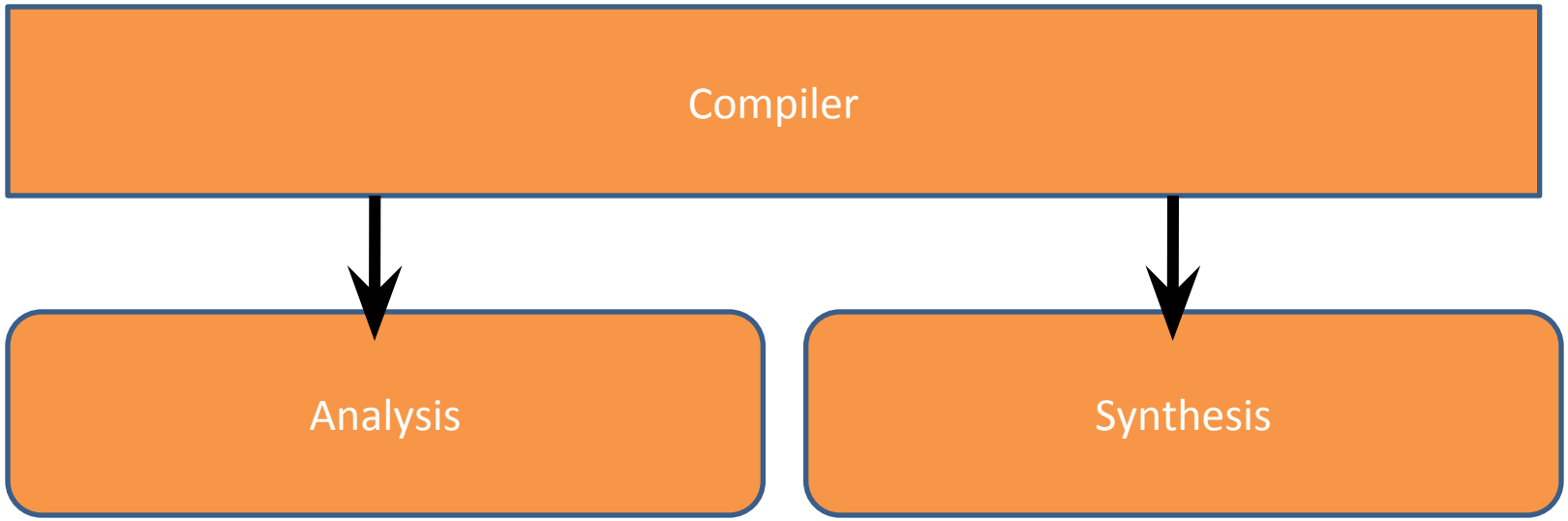
A compiler translates text from a source language, L, to
A target language, L'.

Language Processing System



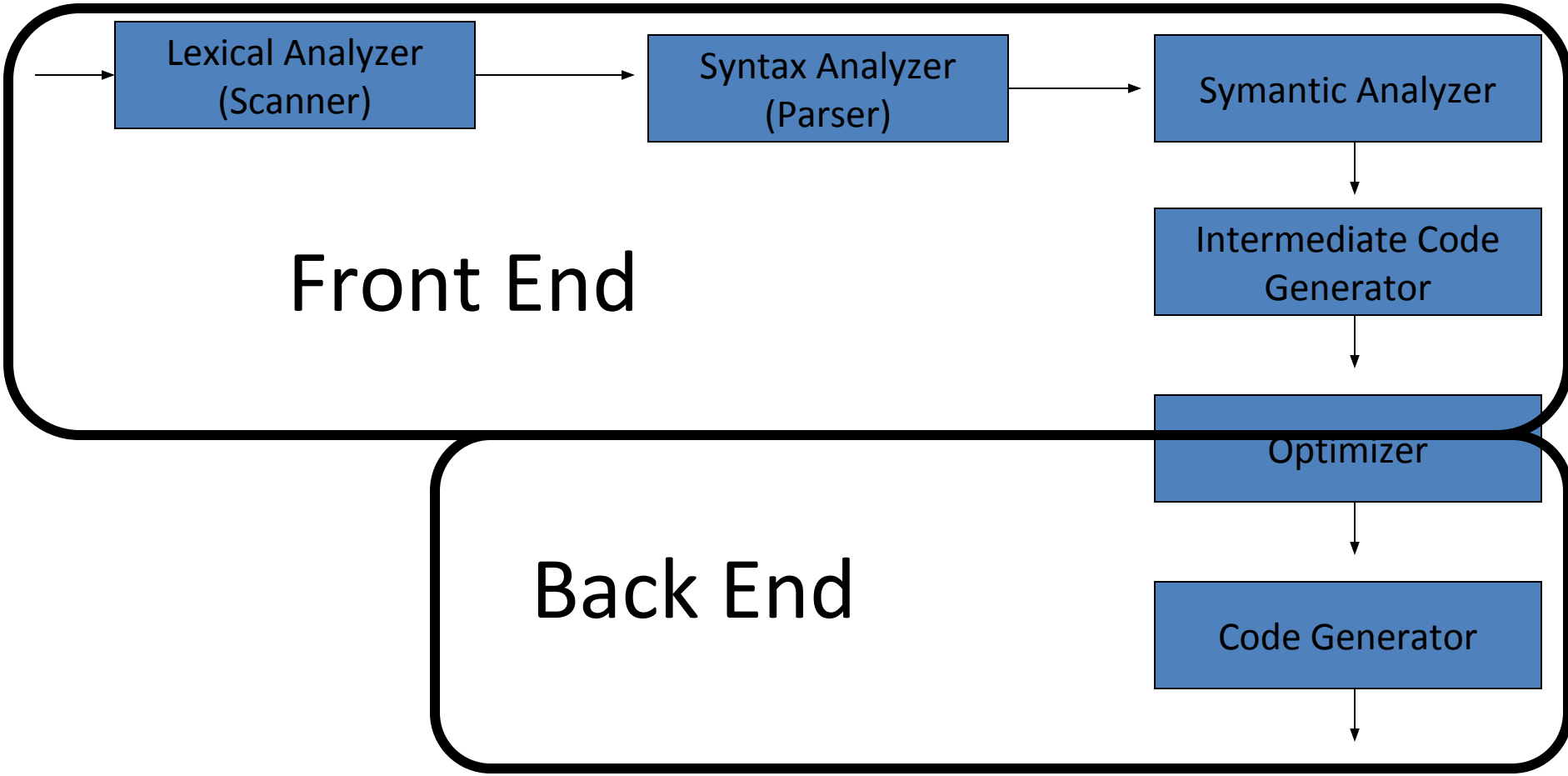
The Structure of a Compiler

- Any compiler must perform two major tasks

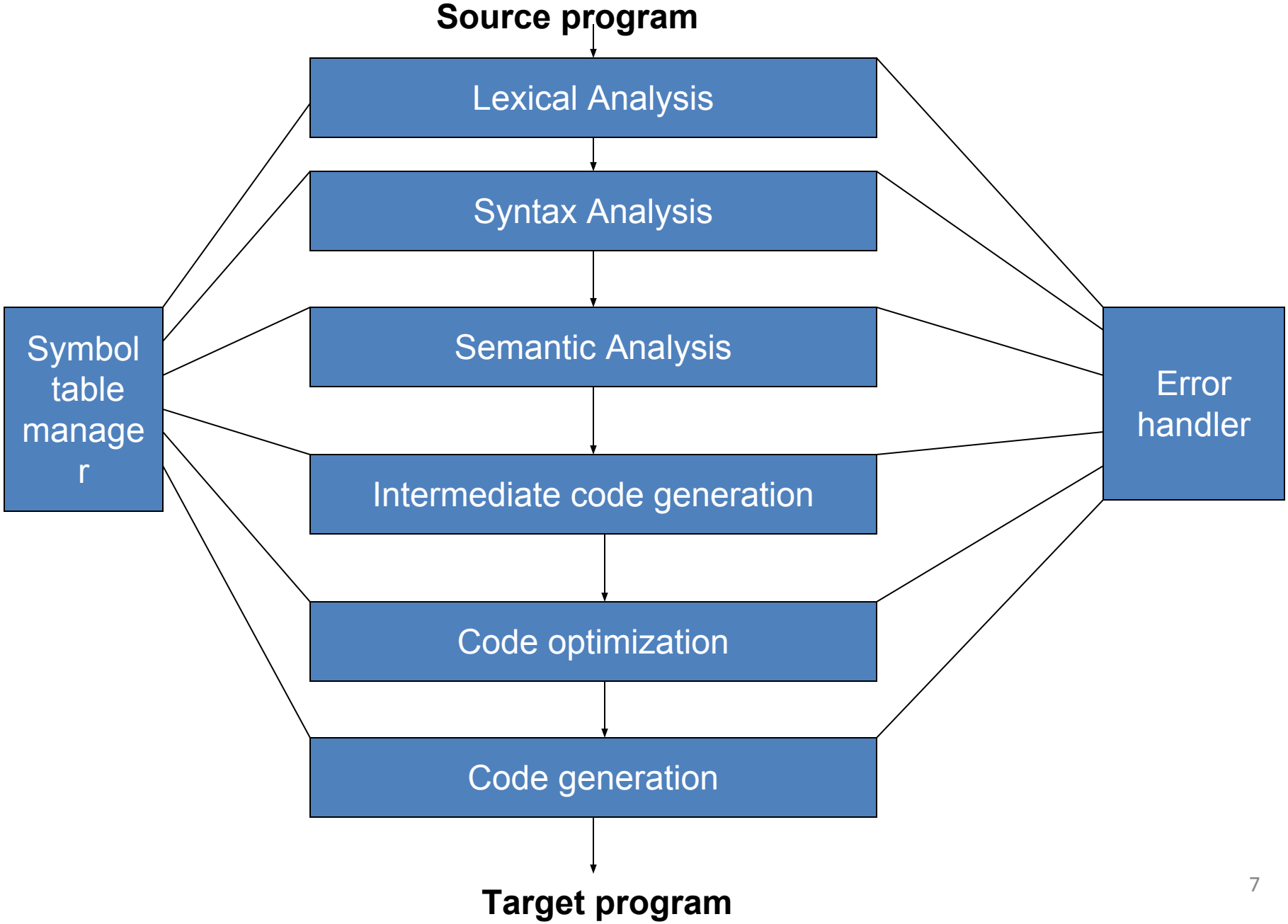


  - ***Analysis*** of the source program
  - ***Synthesis*** of a machine-language program

Compiler Modularity



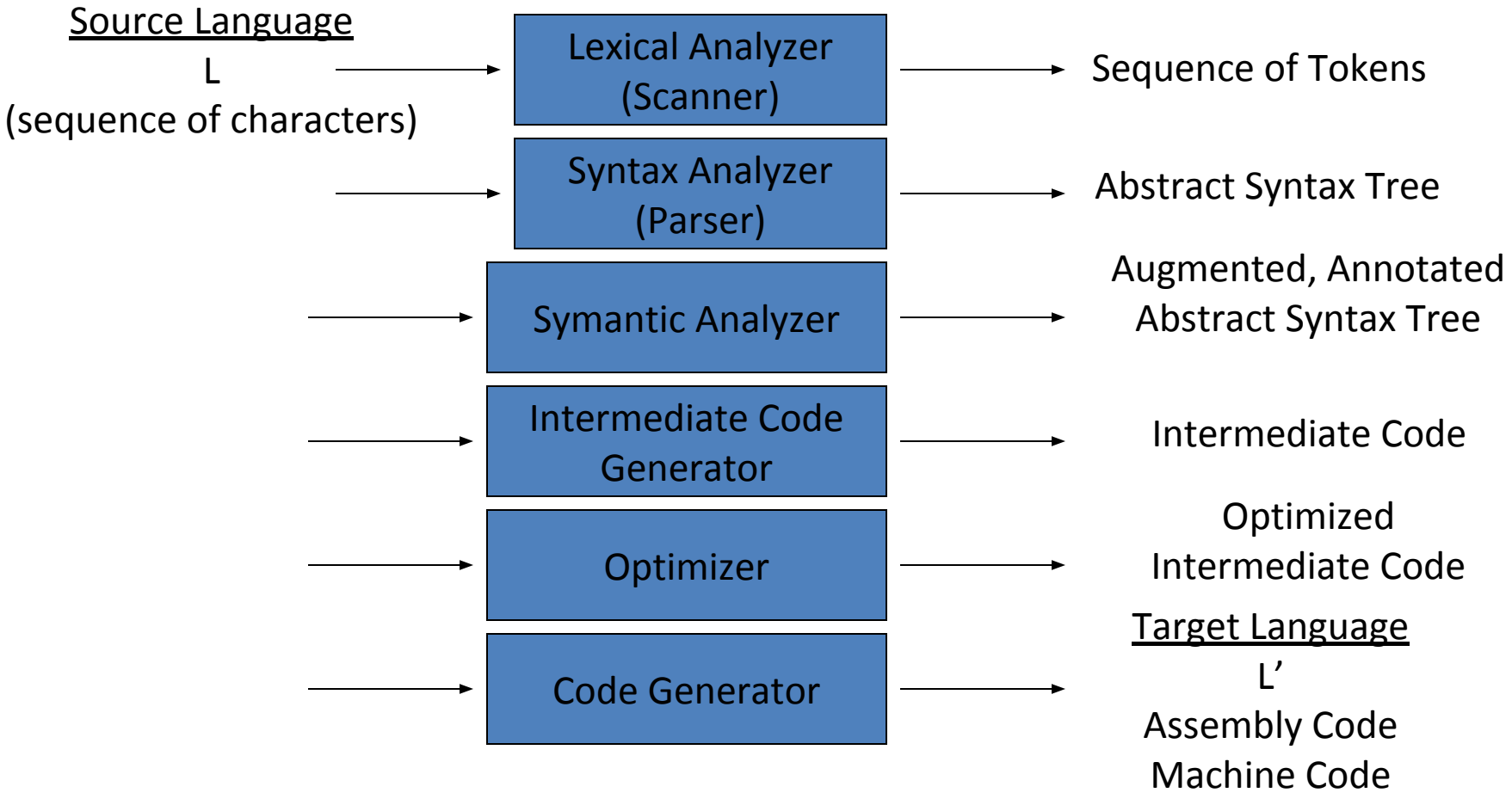
The phases of a compiler



Phases

- A typical real-world compiler usually has multiple phases
- The front-end consists of the following phases:
  - **Scanning**: a scanner groups input characters into tokens
  - **parsing**: a parser recognizes sequences of tokens according to some grammar and generates *Abstract Syntax Trees* (ASTs)
  - **semantic analysis**: performs type checking and translates ASTs into IRs
  - **optimization**: optimizes IRs
- The back-end consists of the following phases:
  - **instruction selection**: maps IRs into assembly code
  - **code optimization**: optimizes the assembly code using control-flow and data-flow analyses, register allocation, etc
  - **code generation**: generates machine code from assembly code

What is a Compiler?



Entire Compilation process

Lexical Analysis

2 Portions of Scanner/Lexical Analyzer Process

- **Scanning** - It consists of simple processes that do not require the tokenization of the input such as deletion of comments, compaction of consecutive white space characters into one
- **Lexical Analysis-** This is the more complex portion where the scanner produces sequence of tokens as output

The Scanner/Lexical Analyzer

- Reads characters from the source program (TEXT SCANNING)
- Groups characters into **lexemes (**sequences of characters that "go together“)
  - stream of characters are convert it into meaningful LEXEMES (called TOKENS)
- Lexemes corresponds to **tokens**; scanner returns next token (plus maybe some additional information) to the parser
- Scanner also discovers lexical errors (e.g., erroneous characters such as # in java)

Lexical Analysis Basic Tasks

- Basic tasks in lexical analysis
  - Recognizing basic elements
  - Removal of white spaces and comments
  - Recognizing literals or constants
  - Recognizing identifiers and keywords
  - Generating Tokens (So also called TOKENIZER)

Lexical analysis terms

- Lexemes
  - A sequence of characters in the source program that is matched by the pattern for a token.
- Pattern
  - A rule describing the set of lexemes that can represent a particular token in source program.
- Attributes
  - The lexical analyzer collects information about tokens into their associated attributes

Tokens, Patterns and Lexemes

- **Token** is pair consisting of a **token name** and an **optional attribute value**. The token name is an abstract symbol representing the kind of lexical unit, eg., a **particular keyword** or **an identifier**
  - Sample token names – KEY_TYP1, ID
- **Pattern** is a description of the form that the lexemes of a token may take.
  - In case of a **keyword as a token** the pattern is **just a sequence of characters** that form the keyword
- **Lexeme** is a **sequence of characters** in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token

Example Lexemes and Tokens

Lexeme:	**;**	**=**	**index**	**tmp**	**21**	**64.32**
Token:	**SEMI-COLON**	**ASSIGN**	**IDENT**	**IDENT**	**INT-LIT**	**INT-FLOAT**

Source Code:

position = initial + rate * 60 ;

Corresponding Tokens:

IDENT ASSIGN IDENT PLUS IDENT TIMES INT-LIT SEMI-COLON

Example Lexemes and Tokens

Lexeme:	**;**	**=**	**index**	**tmp**	**21**	**64.32**
Token:	**SEMI-COLON**	**ASSIGN**	**IDENT**	**IDENT**	**INT-LIT**	**INT-FLOAT**

Source Code:

position = initial + rate * 60 ;

Corresponding Tokens:

IDENT ASSIGN IDENT PLUS IDENT TIMES INT-LIT SEMI-COLON

Tokens

- Constants or literals (e.g. 4.5, 3, 900)
- Identifiers (e.g. A, a, num)
- Operator symbols (e.g.=, +,*)
- Keywords (e.g. for, if, do, goto)
- Punctuation symbols (e.g. {,], (, comma)

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

Find the number of tokens

```
int main()
{ // 2 variables
int a1, b2;
a1 = 10;
return 0; }
```

'int' 'main' '(' ')' '{' 'int' 'a1' ',' 'b1' ';' 'a1' '=' '10' ';'
'return' '0' ';' '}‘

Find the number of tokens

```
int main()
{ // 2 variables
int a1, b2;
a1 = 10;
return 0; }
```

'int' 'main' '(' ')' '{' 'int' 'a1' ',' 'b1' ';' 'a1' '=' '10' ';'
'return' '0' ';' '}‘

Ans: Eighteen tokens

Find the number of tokens

```
int main() {
int a1 = 10, b1 = 20;
printf("sum is :%d",a1+b1);
return 0; }
```

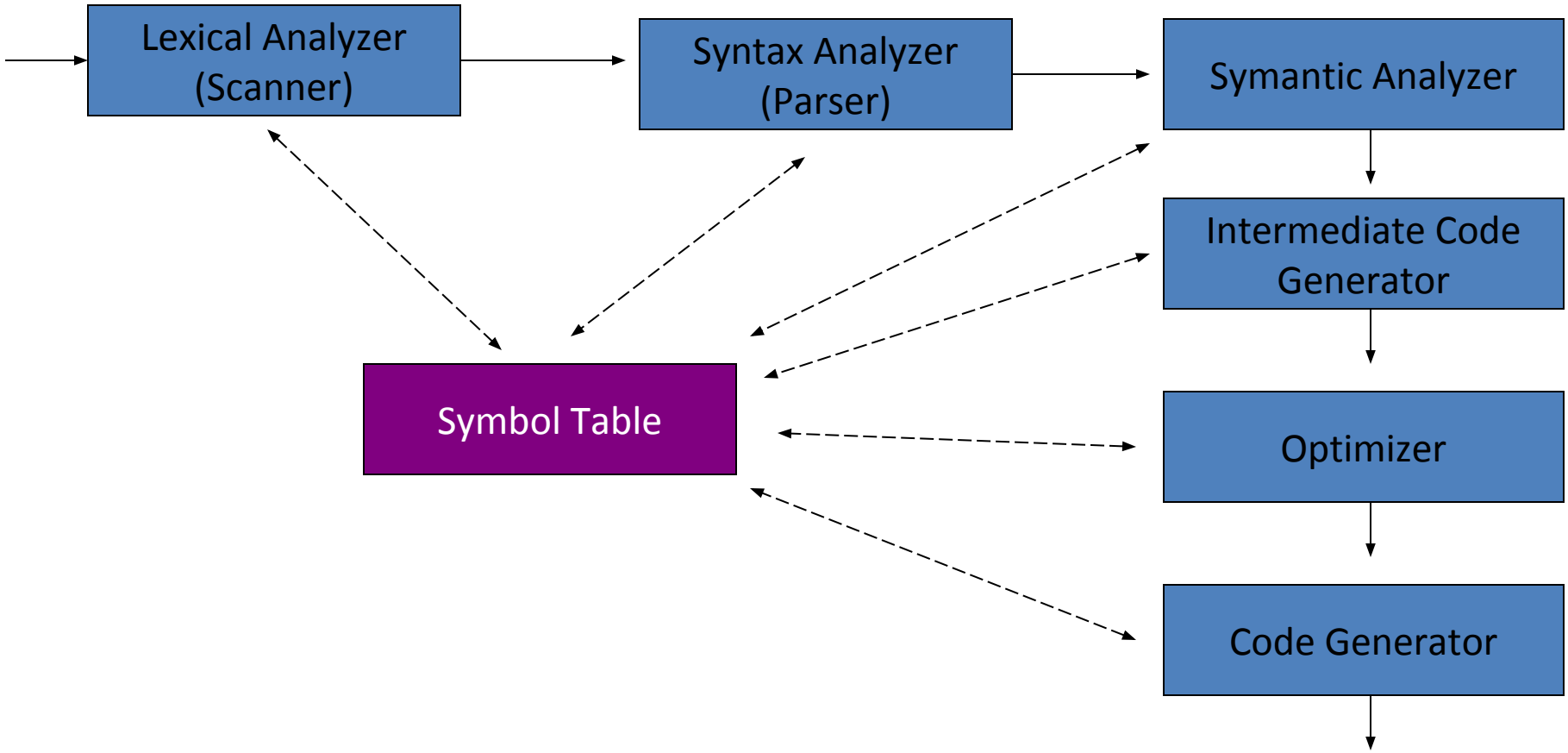
Find the number of tokens

```
int main() {
int a1 = 10, b1 = 20;
printf("sum is :%d",a1+b1);
return 0; }
```

Answer: Total number of token: 27

STRING in “” is 1 token

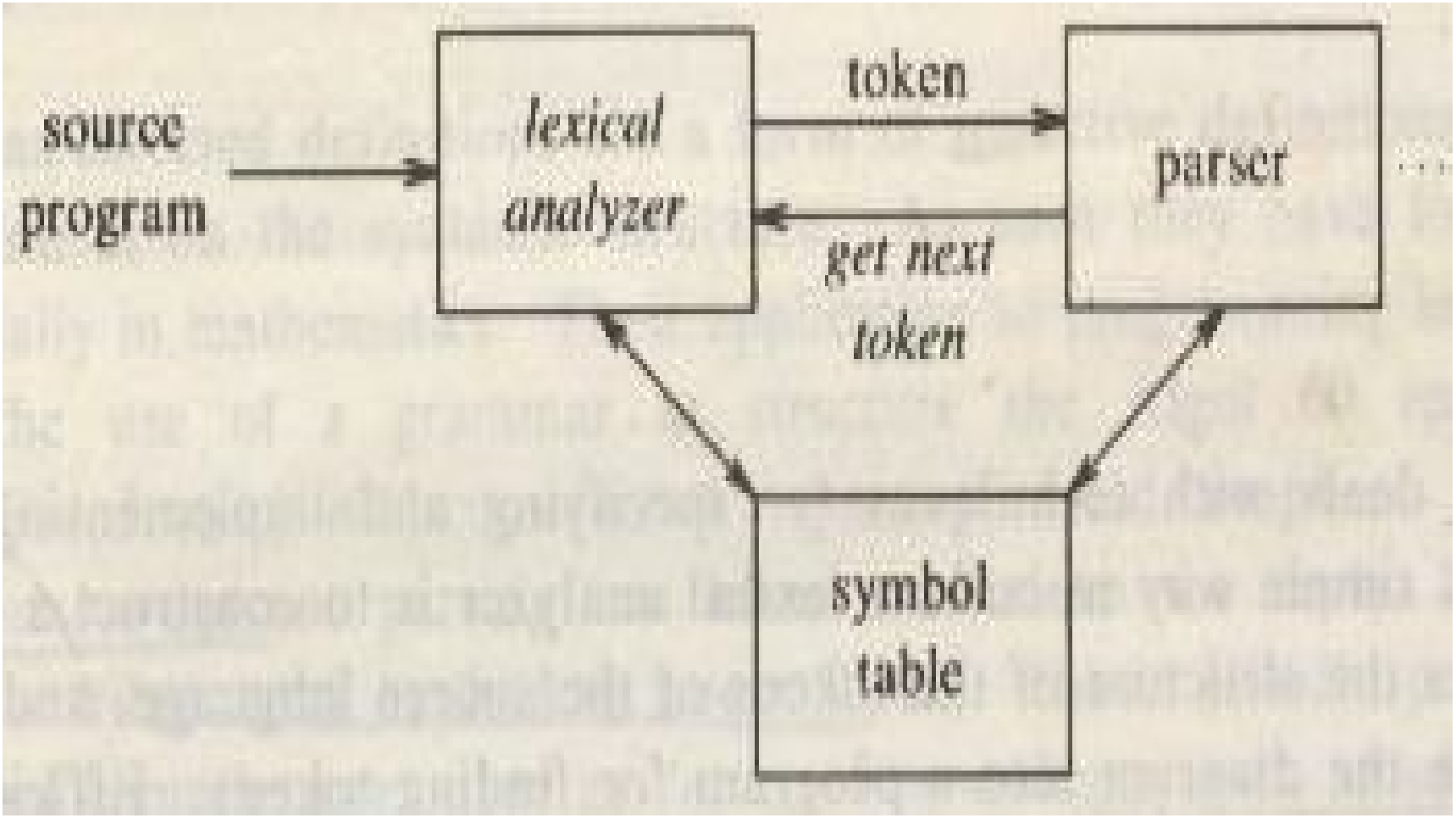
Symbol Tables



Symbol Tables

- Keep track of names declared in the program
- Separate level for each scope
- Used to analyze static symantics:
  - Variables should not be declared more than once in a scope
  - Variables should not be used before being declared
  - Parameter types for methods should match method declaration

Lexical analysis



The Parser

- Groups tokens into "grammatical phrases", discovering the underlying structure of the source program.

- Finds syntax errors.

  For example, in Java the source code

  position = * 5 ;

  corresponds to the sequence of tokens:

  IDENT ASSIGN TIMES INT-LIT SEMI-COLON

  All are legal tokens, but that sequence of tokens is erroneous.

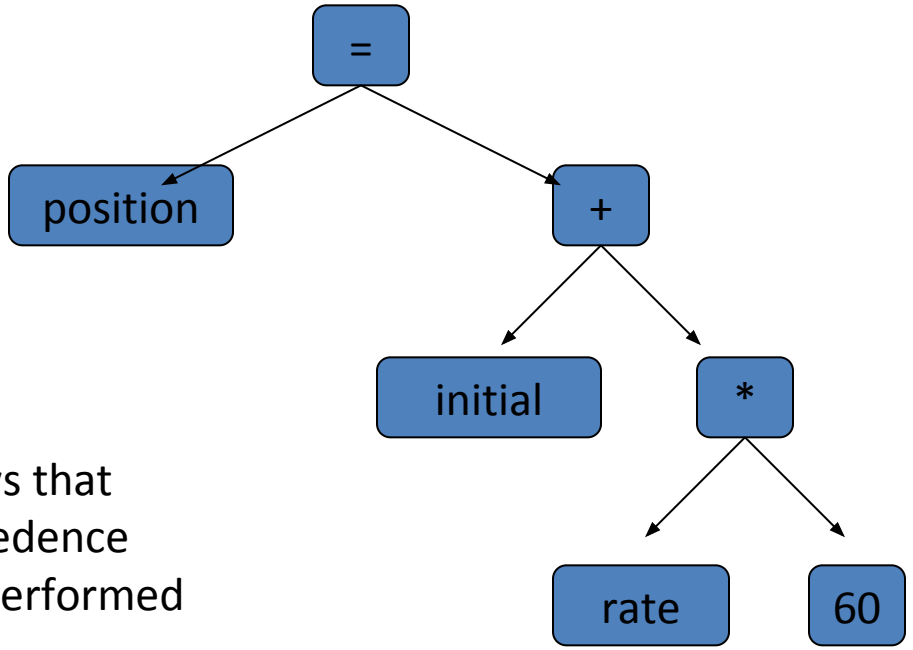
- Might find some "static semantic" errors,
  - e.g., a use of an undeclared variable,
  - variables that are declared multiple times
- Might generate code, or build some intermediate representation of the program such as an abstract-syntax tree.

Example Parse

Source Code:

position = initial + rate * 60 ;

Abstract-Syntax Tree:



- Interior nodes are **operators**.
- A node's children are **operands**.
- Each subtree forms "logical unit"

  e.g., the subtree with * at its root shows that because multiplication has higher precedence than addition, this operation must be performed as a unit (*not* initial+rate).

Parser

- Verifies that the string of token can be generated by the grammar for the source program

Classification of parser

- Top-down
  - Builds parse tree from root (top) to leaves (bottom)
  - Recursive Descent parsing
  - Predictive parsers
  - Nonrecursive predictive parsing
- Bottom-up
  - Builds parse tree from leaves (bottom) to root (up)
  - Shift-reduce parsing
  - Operator-precedence

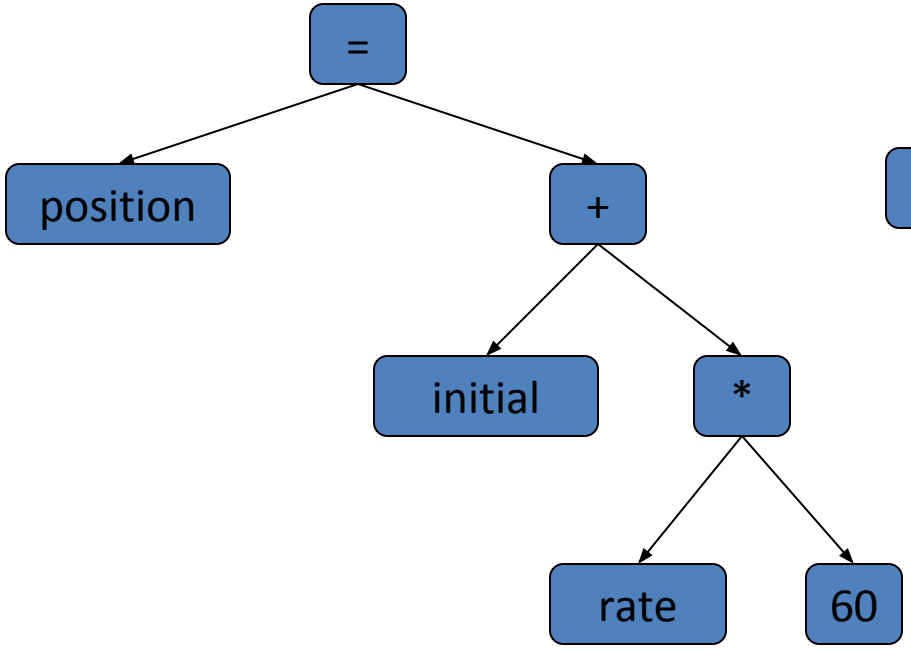
Semantic Analyzer

Semantic Analyzer

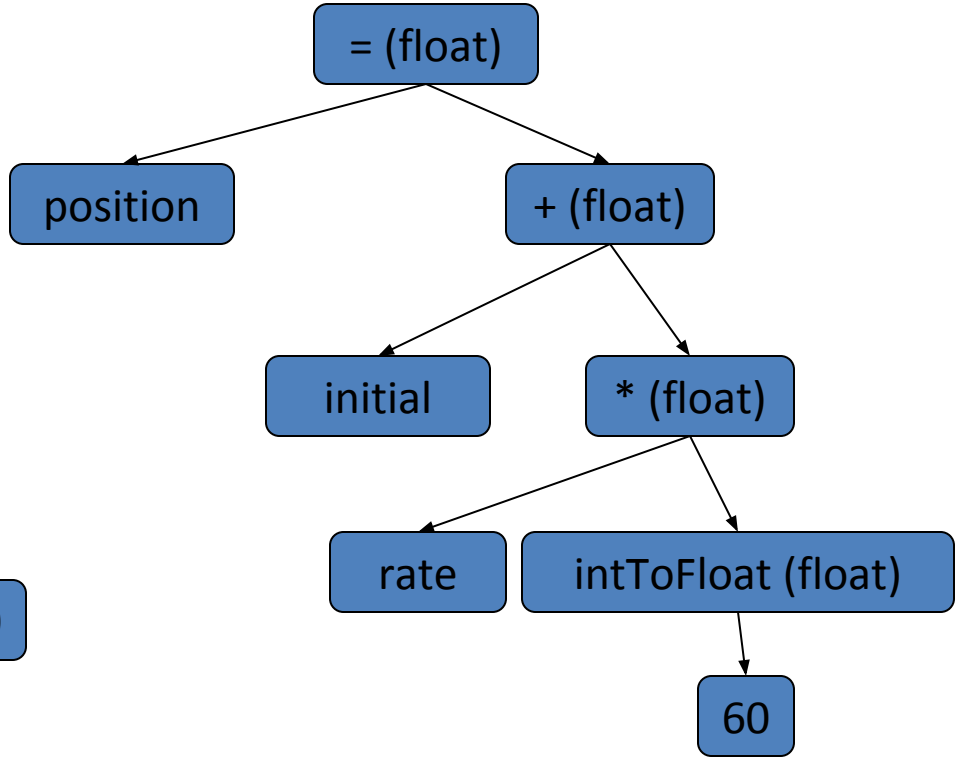
- Checks for (more) "static semantic" errors
- Annotate and/or change the abstract syntax tree

Example Semantic Analysis

Abstract-Syntax Tree:



Annotated Abstract-Syntax Tree:



Intermediate Code Generation

Intermediate Code Generator

- Translates from abstract-syntax tree to intermediate code
- One possibility is **3-address code**

  each instruction involves at most 3 operands

  Example:

  temp1 = inttofloat(60)

  temp2 = rate * temp1

  temp3 = initial + temp2

  position = temp3

Three Address Code

- Three-address code is a linearization of the tree.
- Three-address code is useful: related to machine-language/ simple/ optimizable
- Statements $x := y \text{ op } z$

EXAMPLE: $x := y + z * w$

should be represented as

$t_1 := z * w$

$t_2 := y + t_1$

$x := t_2$

Example of 3-address code

a=(b*-c) + (b*-c)

t_1:=- c
t_2:=b * t_1
t_3:=- c
t_4:=b * t_3
t_5:=t_2 + t_4
a:=t_5

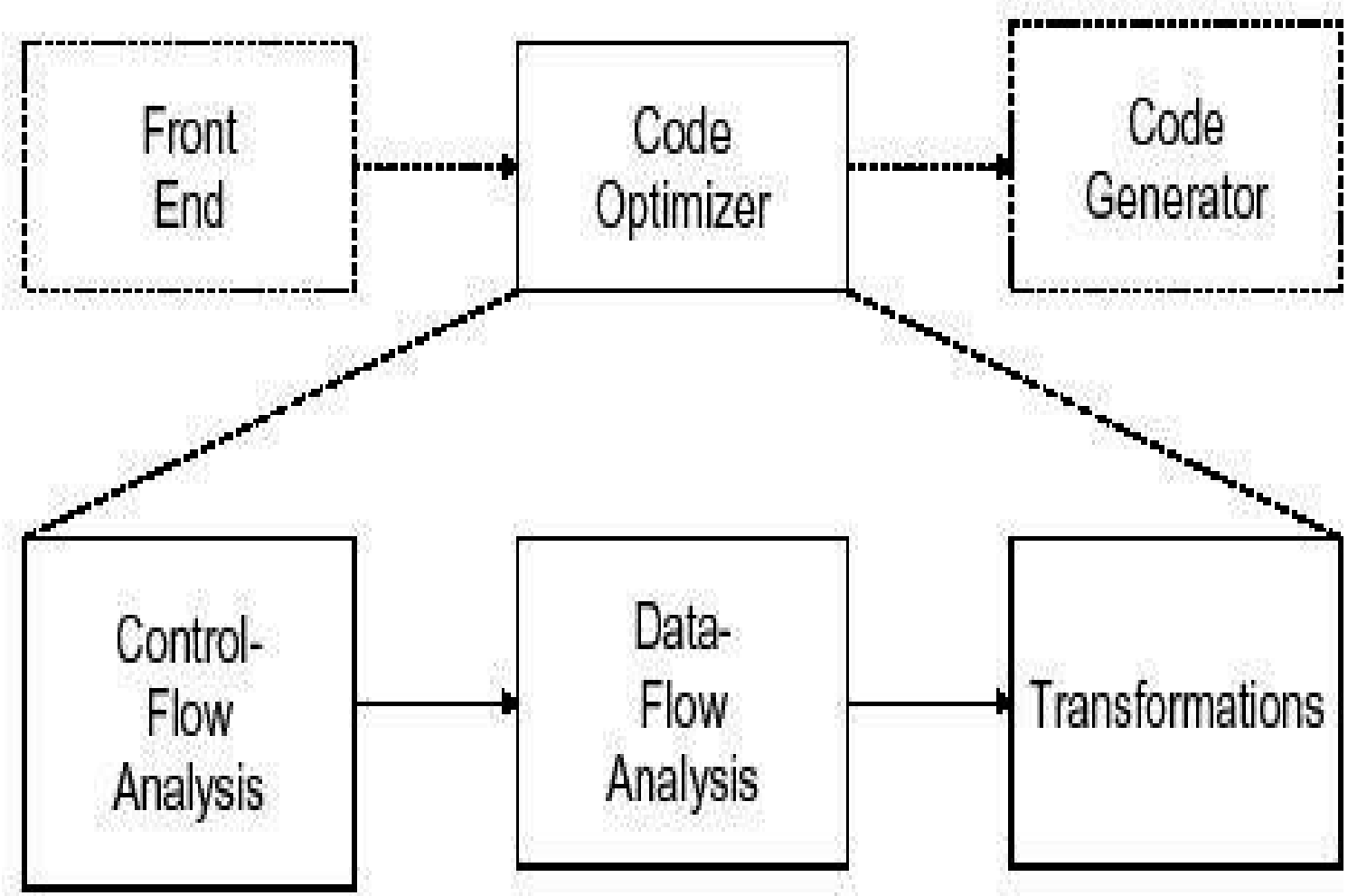
t_1:=- c
t_2:=b * t_1
t_5:=t_2 + t_2
a:=t_5

Code Optimization

Optimization

- Code produced by standard algorithms can often be made to **run faster, take less space or both**
- These improvements are achieved through transformations called optimizations
- Compilers that apply these transformations are called optimizing compilers
- **Optimizers** try to improve code to
  - Run faster, Be smaller, Consume less energy

Code Optimization and Phases



Principal sources of optimization

- Function-preserving transformations
  - **Common subexpressions**
  - **Copy propogation**
  - **Dead-code elimination**
- Loop optimization
  - **Code motion**, **Induction variables** , **Reduction in strength**

Code Generation

The Structure of a Compiler

```
position := initial + rate * 60
```

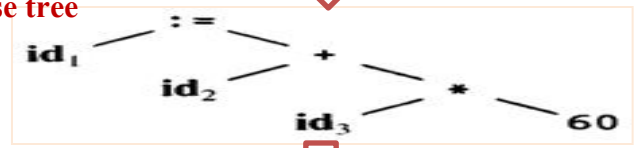
Scanner
[Lexical Analyzer]

Tokens

$id_1 := id_2 + id_3 * 60$

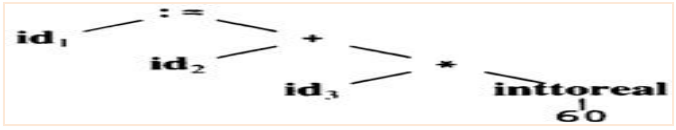
Parser
[Syntax Analyzer]

Parse tree



Semantic Process
[Semantic analyzer]

Abstract Syntax Tree w/ Attributes



SYMBOL TABLE

1	position	···
2	initial	···
3	rate	···
4		

Code Generator
[Intermediate Code Generator]

Non-optimized Intermediate Code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Code Optimizer

Optimized Intermediate Code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code Generator

Target machine code

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Unit III :
Introduction To Compilers (TOPICS)

Phase structure of Compiler and entire compilation process

Lexical Analyzer: **The Role of the Lexical Analyzer,** Input Buffering.
Specification of Tokens, Recognition Tokens, Design of Lexical Analyzer using Uniform Symbol Table, Lexical Errors.

LEX : LEX Specification, Generation of Lexical Analyzer by LEX.

The Role of Lexical Analyzer (LA)

6 Tasks/Roles of the Lexical Analyzer(LA)

1) As the **first phase of compiler**, main task is to read the input characters of the **source program** and group them into lexemes
   - **Output** produced is a sequence of tokens for each lexeme in the source program

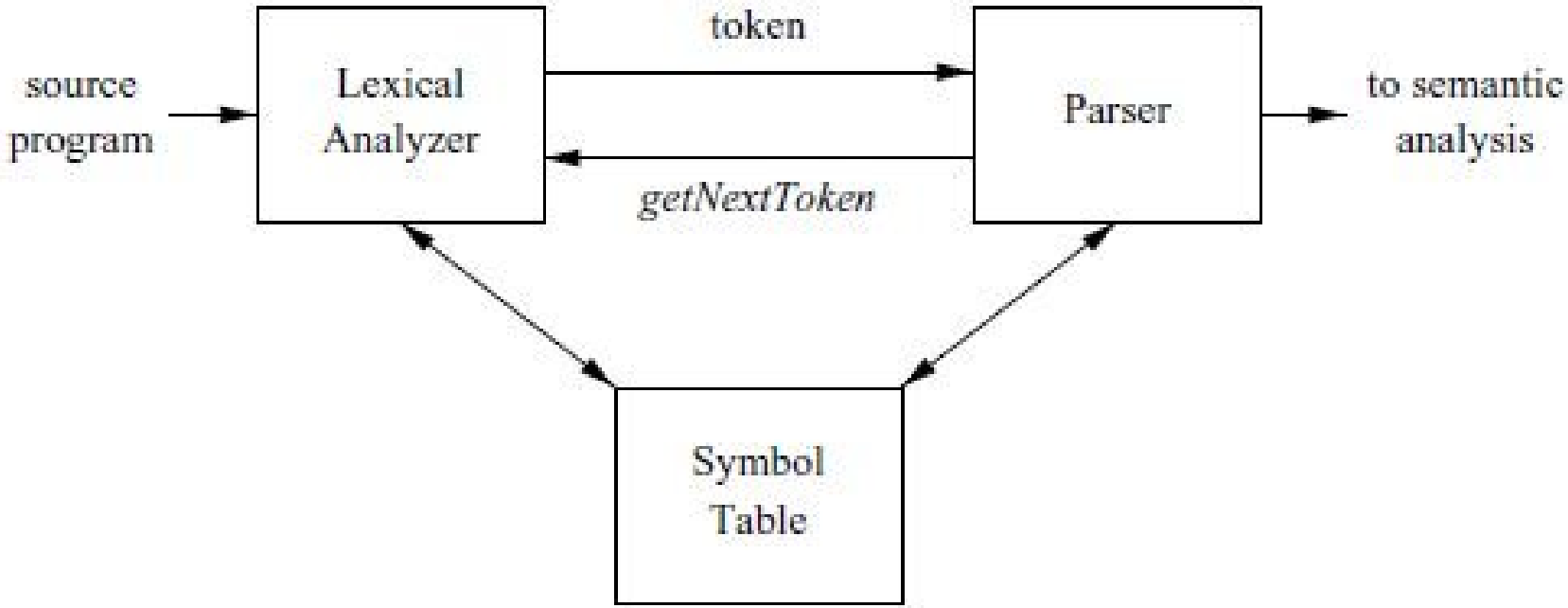
….contd…6 Tasks of the Lexical Analyzer (LA)

2) LA needs to **enter identifiers** into the symbol table

3) LA **preprocesses** the source code text like removing comments, white spaces, newline characters, tabs, (*also other characters which delimit the tokens*), etc.

4) LA communicates with Parser (next phase of compilation) by passing TOKENS

Interactions between LA and Parser



….contd…Tasks of the Lexical Analyzer (LA)

5) **Correlates error messages** generated with the source program
   - Associate a source code line n umber with each error message
   - Or, LA makes a copy of the source program with the error messages inserted at the appropriate positions

6) If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the LA

Q)

List the 6 L A Roles

Lexical Analysis Versus Parsing

- Phase 1: Lexical Analysis
- Phase 2: Syntax Analysis

Q) Why is the analysis portion of a compiler is separated into **lexical analysis** and **parsing** (syntax analysis) phases?

Why better to separate the two tasks into separate phases?

1) Simplicity of design :
   - Dividing task into 2 tasks, often allows us to **simplify at least one** of these tasks
   - E.g. if parser that had to deal with comments and whitespace as syntactic units (grammar rules) would be considerably more complex than if they were detected and removed in LA phase
   - When designing a new language, separating lexical and syntactic issues leads to a cleaner language design
2) Improving compiler efficiency :
   - Specialized improvement technique meant for lexical tasks only can be applied
     - Specialized buffering techniques for reading input characters can speed up the compiler significantly
3) Compiler portability is enhanced : Input-device-specific peculiarities can be restricted to the lexical analyzer

Unit III :
Introduction To Compilers (TOPICS)

Phase structure of Compiler and entire compilation process

Lexical Analyzer: The Role of the Lexical Analyzer, **Input Buffering**

Specification of Tokens, Recognition Tokens,

Design of Lexical Analyzer using Uniform Symbol Table,

Lexical Errors.

LEX : LEX Specification, Generation of Lexical Analyzer by LEX.

Input Buffering:

- Amount **of time taken to process characters** and the **large number of characters** that must be processed during the **compilation of a large** source program, **is large**
- **Specialized buffering techniques** have been developed to **reduce the amount of overhead** required to process a single input character

Two pointers to the input are maintained:

- Pointer ***Lexeme Begin***, marks the beginning of the current lexeme, whose extent we are attempting to determine
- Pointer ***Forward,*** scans ahead until a pattern match is found.
- Once the next lexeme is determined, *forward* is set to character at its right end.
- Then, after the lexeme is recorded as an attribute value of a token returned to the parser, *Lexeme Begin* is set to the character immediately after the lexeme just found.

Single Buffer

- Single Buffer

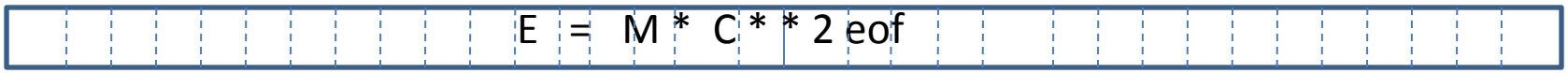
Optimizing Reads from the Buffer

- A buffer at its end may contain an initial portion of a lexeme

```
E = M*C**2
```

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
- **Two buffer scheme** to handle large look-aheads safely



..Contd..Input Buffering

- Examining ways of speeding reading the source program
  - Two-buffer scheme handling large look ahead safely

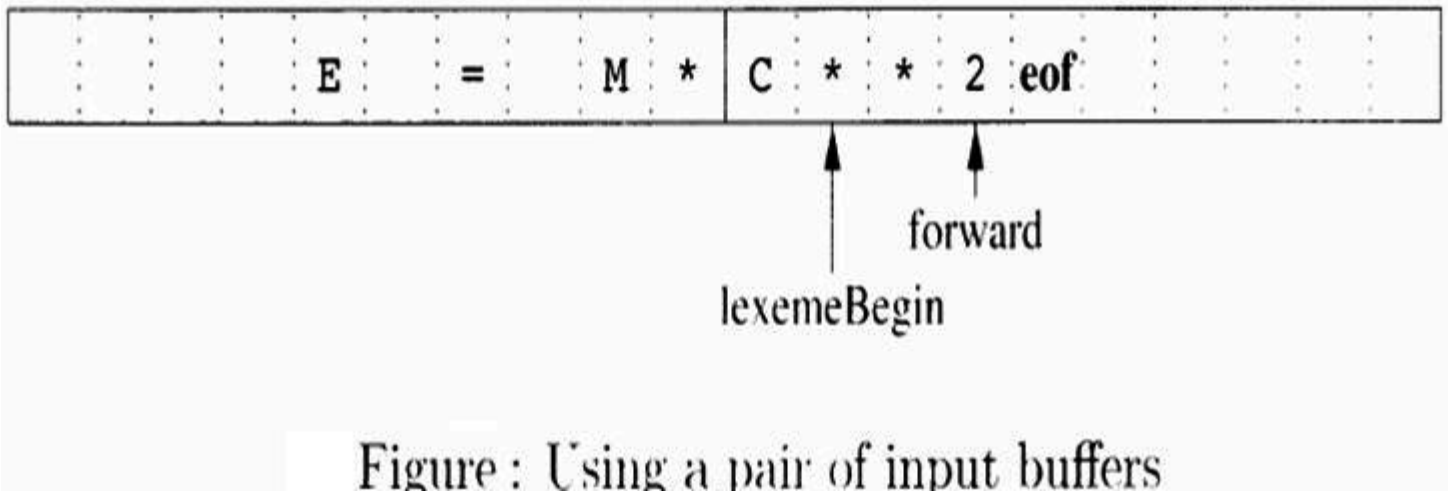


Figure : Using a pair of input buffers

Input buffering

Buffer Pairs

- Two buffers of the same size, say 4096, are alternately reloaded.
- Two pointers to the input are maintained:
  - Pointer **`lexeme_Begin`** marks the beginning of the current lexeme.
  - Pointer **`forward`** scans ahead until a pattern match is found.

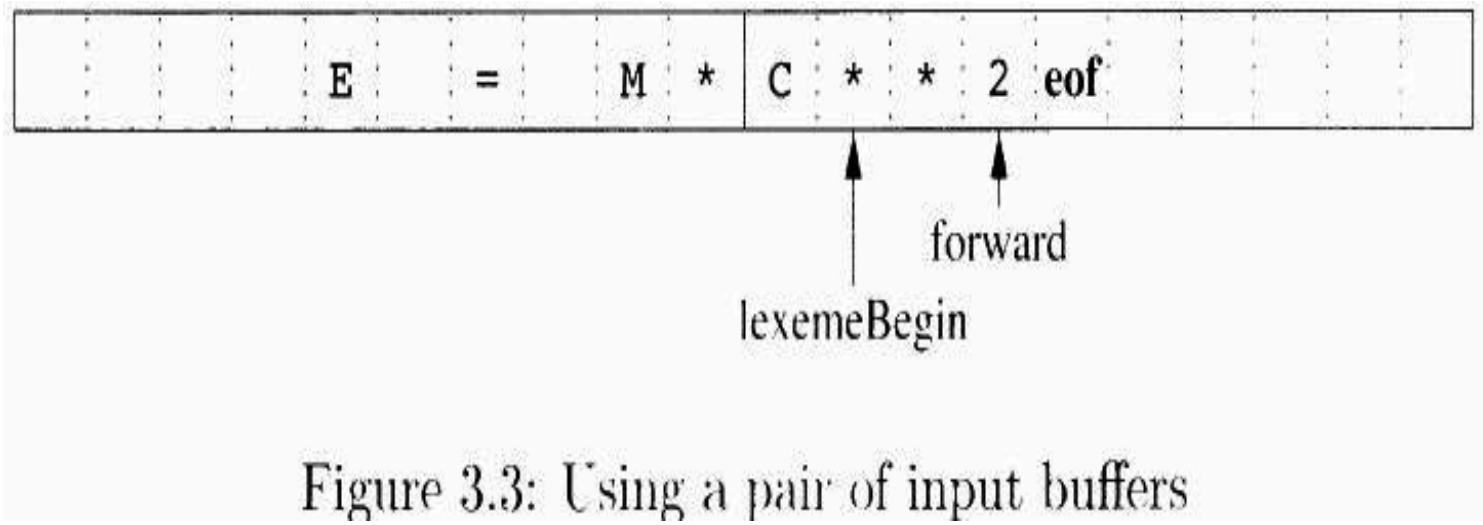
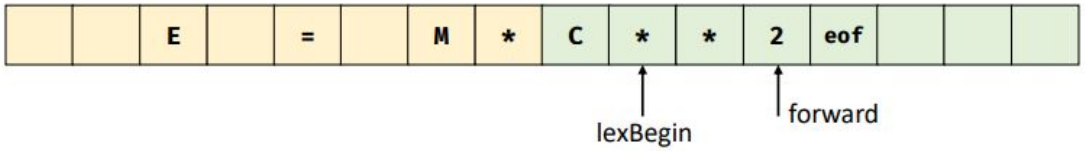


Figure 3.3: Using a pair of input buffers

Two – buffer scheme

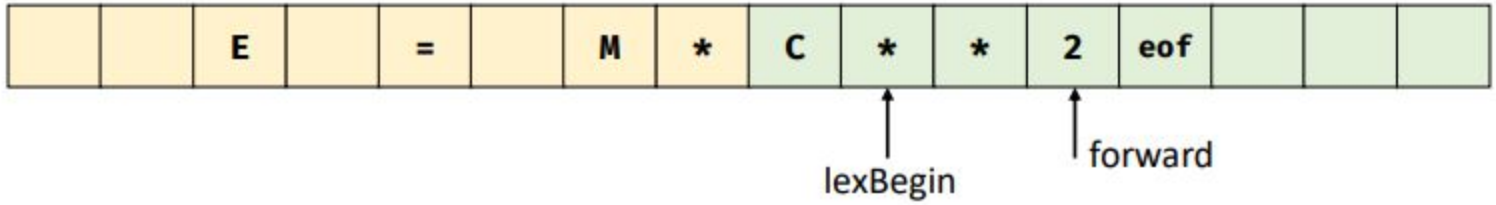
Optimizing Reads from the Buffer

- A buffer at its end may contain an initial portion of a lexeme
  - It creates problem in refilling the buffer, so a two-buffer scheme is used
  - The two buffers are filled alternatively



Read from buffer

- (1) Check for end of buffer, and (2) test the type of the input character
- If end of buffer, then reload the other buffer

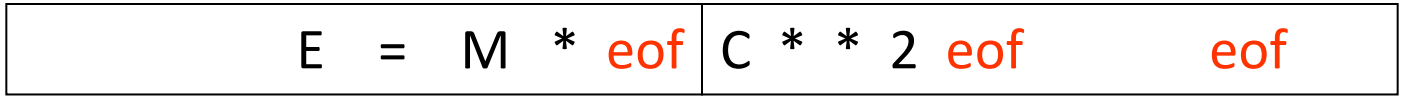


Input buffering 4

```
If forward at end of first half then begin
        reload second half;
        forward:=forward + 1;
End
Else
if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
End
Else forward:=forward + 1;
```

Input buffering 5

Sentinels



Sentinel is a special character that cannot be part of the source program and a natural choice is the character ***eof***

Q)

- Explain the need and technique of Input Buffering.

Unit III :
Introduction To Compilers (TOPICS)

Phase structure of Compiler and entire compilation process

Lexical Analyzer: The Role of the Lexical Analyzer, Input Buffering.

Specification of Tokens, Recognition Tokens,

Design of Lexical Analyzer using Uniform Symbol Table,

Lexical Errors.

LEX : LEX Specification, Generation of Lexical Analyzer by LEX.

Specification of Tokens

REs

Lexical grammar

- Formal grammar defining the syntax of tokens
- A program is written using characters that are defined by the **lexical structure** of the language used
- The **character set** is equivalent to the **alphabet** used by any written language
- The lexical grammar lays down the rules governing how a character sequence is divided up into subsequences of characters, each part of which represents an **individual token**
  - This is frequently defined in terms of **regular expressions**

Regular expressions

- Have the capability to express finite languages by defining a pattern for finite strings of symbols.
  - The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.
  - Regular expression is an important notation for specifying patterns.
  - Each pattern matches a set of strings, so regular expressions serve as names for a set of strings.
  - Using the algebraic laws that are obeyed by regular expressions, one c manipulate regular expressions into equivalent forms

Specification of Tokens

- **Finite number of token types:** The lexical analyzer needs to scan and identify only a **finite set of valid string/token/lexeme** that belong to the **language** in hand.
- It searches for the **pattern defined** by the **language rules**.

- **Programming language tokens :** can be described by regular languages
  - The specification of regular expressions is an example of a recursive definition
  - Regular languages are easy to understand and have efficient implementation

Sample Lexical grammar for many programming languages

- Specifies the following
  - **String Literal** starts with a " character and continues until a matching " is found (escaping makes this more complicated)
    - “bbb”
  - An **identifier** is an alphanumeric sequence (letters and digits, usually also allowing underscores, and disallowing initial digits)
    - A_dgt50_r
  - **Integer literal** is a sequence of digits - 1, 101 , 9980646

Sample Lexical grammar for many programming languages

- Sample
  - INPUT: "abc" xyz1 23
  - TOKENS : *string*, *identifier* and *number* (plus whitespace tokens?)
- Certain **sequences** are categorized as **keywords** – these generally have the same form as identifiers (usually alphabetical words), but are **categorized separately**; formally they have a **different token type**

Building a Regular Expression

Built recursively out of smaller regular expressions, using the rules described below.

Each regular expression, r, denotes a language L(r), which is also defined recursively from the languages denoted by r's subexpressions.

Rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. ϵ is a regular expression, and L(ϵ) is { ϵ }, that is, the language whose sole member is the empty string.
2. If a is a symbol in Σ, then **a** is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with a in its one position.

Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.

Alphabet Σ

- An alphabet is any finite set of symbols
- Typical examples of symbols are letters, digits, and punctuation **– for Programming Languages**
- The set {0,1} is the binary alphabet
- **ASCII is an important example of an alphabet**; it is used in many software systems
- **Unicode**, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones.

Suppose r and s are regular expressions denoting languages L(r) and L(s), respectively.

1. (r)|(s) is a regular expression denoting the language L(r)IL(s).
2. (r)(s) is a regular expression denoting the language L(r)L(s)
3. (r)* is a regular expression denoting(L(r))*
4. (r) is a regular expression denoting L(r). (i.e. We can add additional pairs of parentheses around expressions without changing the language they denote)

Regular expressions rule summary

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) $\cup$ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

Regular definitions

d1 -> r1
d2 -> r2
…
dn -> rn

Examples:

Labc -> a|b|c

letter_ -> A | B | … | Z | a | b | … | z | _
digit -> 0 | 1 | … | 9
id -> letter_ (letter_ | digit)*
Id -> [a-z_A-Z][a-z_A-Z0-9]*

Extensions

- One or more instances of r: $(r)^+$
- Zero or more instances of r : r*
- Zero or one instances of r: r?
- Character classes: [abc]

- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter_|digit)*

Recap: String and Language

- A **string** over an alphabet is a **finite sequence** of symbols drawn from that alphabet. In language theory, the terms “**sentence**" and “**word**" are often used as synonyms for “**string**”
- A **language** is any countable set of strings over some fixed alphabet (broad definition)
  - Sample languages:
    - Abstract languages containing empty set
    - Set of all syntactically well-formed C programs
    - Set of all grammatically correct English sentences

Term for Parts of String

1. A prefix of string s is any string obtained by removing zero or more symbols from the end of s. For example, ban, banana, and epsilon are prefixes of banana

2**. A suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana, banana, and epsilon are suffixes of banana

3. A **substring** of s is obtained by deleting any prefix and any suffix from s. For instance, banana, nan, and epsilon are substrings of banana.

4. The **proper** prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not epsilon or not equal to s itself

5. A **subsequence** of s is any string formed by **deleting zero or more not necessarily consecutive positions** of s. For example, baan is a subsequence of banana

Sample REs for Programming Languages

More Regular Expressions (Lex)

A ,IF , or, run - matches A, IF, or, run respectively

abc - matches abc

[abc] - matches a or b or c

[a-z] - matches a or b or c

[a-zA-Z]+ - combinations of small and capital alphabets

[0-9] - matches any digit

[0-9]+ - matches any positive integer

Unit III :
Introduction To Compilers (TOPICS)

Phase structure of Compiler and entire compilation process

Lexical Analyzer: The Role of the Lexical Analyzer, Input Buffering.

Specification of Tokens, **Recognition Tokens,**

Design of Lexical Analyzer using Uniform Symbol Table,

Lexical Errors.

LEX : LEX Specification, Generation of Lexical Analyzer by LEX.

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

Recap: Deterministic Finite Automata(DFA)

- A DFA is a special kind of finite automaton that has exactly one transition out of each state for each input symbol
- Transitions on empty input are disallowed
- It is Easily simulated and makes a good implementation of a **lexical analyzer**, similar to a transition diagram

Recap: Nondeterministic Finite Automata

- Automata that are not DFA's are called nondeterministic.
- NFA's often are easier to design than are DFA's.
- Another possible architecture for a lexical analyzer is to tabulate all the states that NFA's for each of the possible patterns can be in, as we scan the input characters

USE of tokens for **Syntax**

- Starting point is the language grammar to **understand the tokens**:

stmt -> **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| ε

expr -> term **relop** term
| term

term -> **id**
| **number**

Recognition of tokens (cont.)

- The next step is to **formalize the patterns (REs)**:

  digit -> [0-9]
  Digits -> digit+
  number -> digit(.Digits) ----
  letter -> [A-Za-z_]
  id -> letter(letter|digit)*
  IF -> if
  THEN -> then
  Else -> else
  Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  ws -> (blank | tab | newline)+

Recognition of tokens (cont.)

- Usually based on a finite-state machine (FSM)
- It has encoded within it, information on the possible sequences of characters that can be contained within any of the tokens it handles
  - Recap: Individual instances of these character sequences are termed lexemes
    - E.g. **integer** lexeme may contain any sequence of numerical digit characters

Recognition of tokens : FSM

- Longest match rule : In many cases, the first non-whitespace character can be used to deduce the kind of token that follows and subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token
- In some languages, the **lexeme creation rules are more complex** and may involve backtracking over previously read characters

  – Example : p does not indicate printf(keyword) /p1(identifier)

Q)

- What is the shortest string that is not a member of the language represented by the regular expression a*(ab)*b* (shortest in length)?
- (A) aab
- (B) bba
- (C) ba
- (D) bb

Q)

- Which of the following strings is a member of the set represented by the regular expression
- a(a|b)*a?
- **(A) aabab**
- **(B) aababa**
- **(C) aaab**
- **(D) bababa**

Q)

- Which of the following is the regular expression to represent all binary strings having at least 3 characters, and the third character is 0?
- (A) 0(0|1)0(0|1)*
- (B) 1(0|1)0(0|1)*
- (C) (0|1)(0|1)0(0|1)*
- (D) (0|1)(0|1)0(0|1)

Q)

- How many bit strings of length exactly five are matched by the regular expression 0(0|1)*1?
-
- (A) 8
- (B) 10
- (C) 12
- (D) 14

Q)

- How many bit strings of length at most four are matched by the regular expression 0(0|1)*1?

- (A) 5
- (B) 6
- (C) 7
- (D) 8

Q)

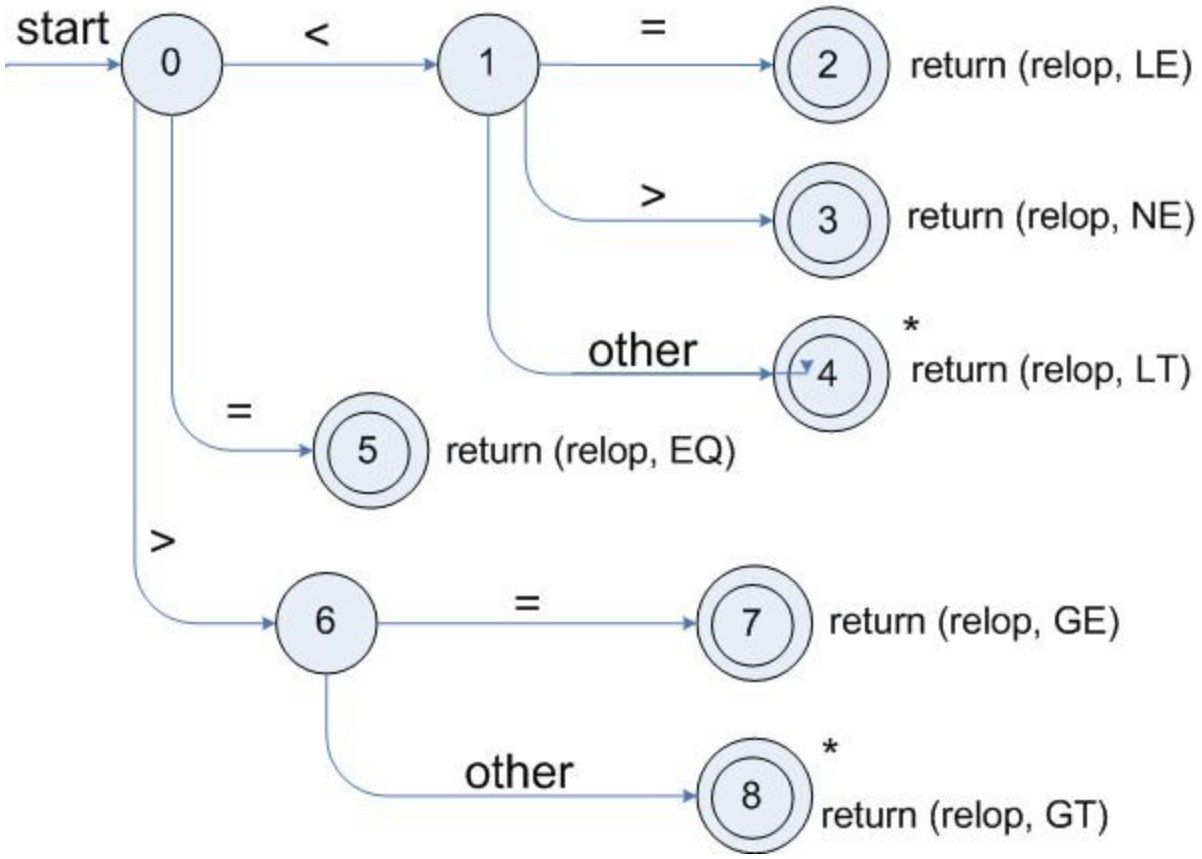
Properties followed by the strings conforming to the regular expression (0|1)*0(0|1)(0|1)(0|1) is/

are:

- (A) Length at least 4
- (B) Fourth character from end is a 0
- (C) Ends with 0 or 1
- (D) All of the other options

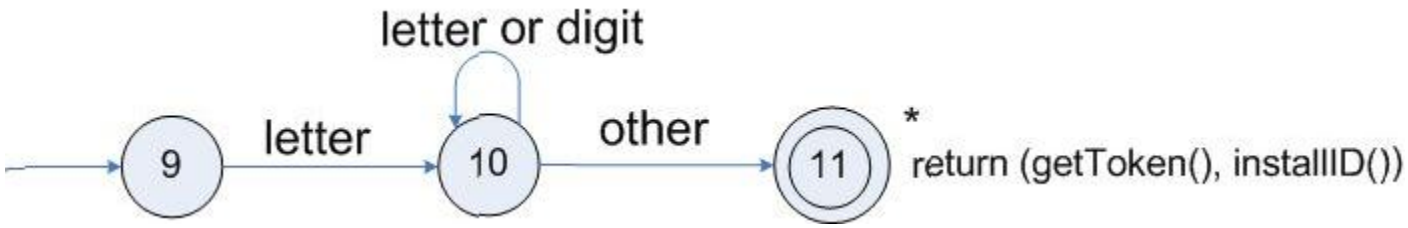
Transition diagrams

- Transition diagram for relop



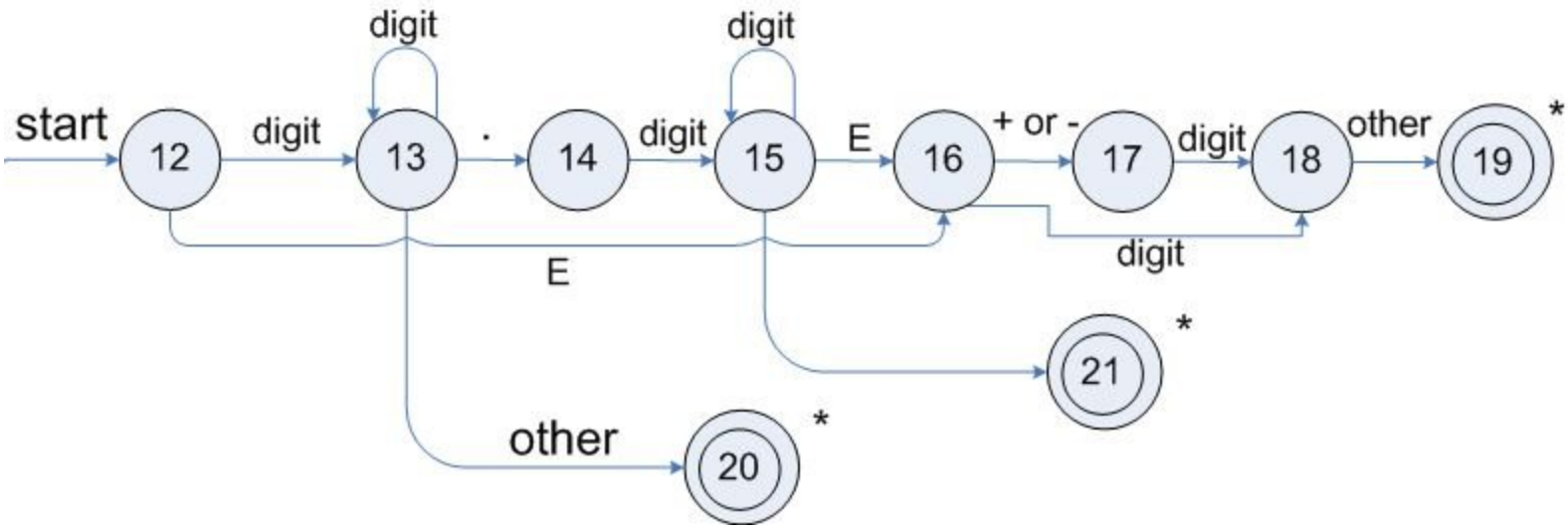
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



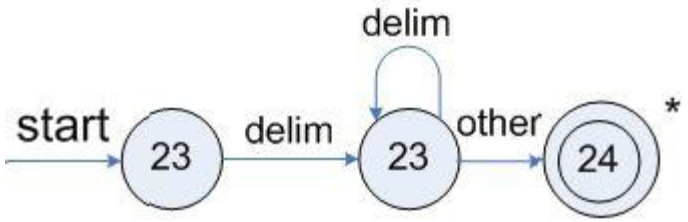
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

- Transition diagram for whitespace



Unit III :
Introduction To Compilers (TOPICS)

Phase structure of Compiler and entire compilation process

Lexical Analyzer: The Role of the Lexical Analyzer, Input Buffering.

Specification of Tokens, Recognition Tokens

Design of Lexical Analyzer using Uniform Symbol Table,

Lexical Errors.

LEX : LEX Specification, Generation of Lexical Analyzer by LEX.

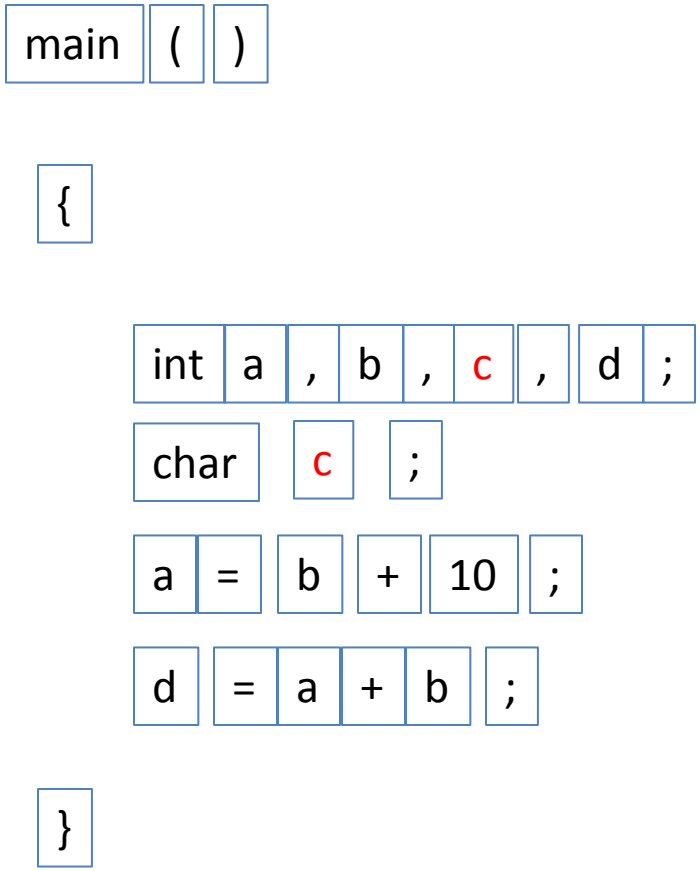
Design of Lexical Analyzer

With Uniform Symbol Table

Tables for Lexical Analysis

- Symbol Table (Name, Attributes)
- Literal Table (literal, Type, Address)
- Terminal Table
- Uniform Symbol Table (Class, Index)

Sample code (with semantic errors) – and Lexical Analyzer output



…contd…Sample code – and Lexical Analyzer output

	Terminal Table
1	(
2	)
3	{
4	}
5	;
6	=
7	+
8	,
9	int
10	char
	……

	Symbol Table	
	Name	**Attribute**
1	main	Filled by later phases
2	a	
3	b	
4	c	
5	d	

Literal Tab	
Literal	**Type**
10	int

main
(
)
{
int
a
,
b
,
ar
c
,

main ( )

{

int a , b , c , d ;

char c ;

a = b + 10 ;

d = a + b ;

}

..

Uniform Symbol Table	
Class	Index
IDN	1
TRM	1
TRM	2
TRM	3
TRM	9
IDN	2
TRM	8
IDN	3
TRM	8
IDN	4
TRM	8
IDN	5
TRM	5
TRM	10
IDN	4
TRM	5
IDN	2
TRM	6
IDN	3
TRM	7
LIT	1
TRM	5
……	……

Unit III :
Introduction To Compilers (TOPICS)

Phase structure of Compiler and entire compilation process

Lexical Analyzer: The Role of the Lexical Analyzer, Input Buffering.
Specification of Tokens, Recognition Tokens,
Design of Lexical Analyzer using Uniform Symbol Table,
Lexical Errors.
LEX : LEX Specification, Generation of Lexical Analyzer by LEX.

Lexical Errors

Lexical Errors

- During the lexical analysis phase **some type** of errors can be detected
- Lexical error is a **sequence of characters** that does not match the pattern of any token
- A character sequence which is not possible to scan into any valid token is a **lexical error**
- Lexical errors are not very common, but it should be managed by a scanner

Sample Lexical errors

- Missing quotes around text intended as a string,
  - Unmatched string, etc

  -printf(“hello); “[^”]”
- Unmatched /* */ ---- if handled by REs

  -/*lexical error
- Unmatched { } ?? NO //Syntax Error not lexical error

…contd… Sample Lexical errors

- **Appearance of some illegal character**, *mostly at the beginning of a token* (e.g. 6val, m@in)

- **Exceeding length** of identifier or numeric constants

…contd…Sample Lexical errors

- Can Recognize errors like:
  - **d = 2r**
  - Such errors are recognized when no pattern for tokens matches a character sequence

- Some errors are **out of power of lexical** analyzer to recognize:
  - **fi (a == f(x)) …**

Error Handling in Lexical Analysis

A Lexical Analyzer (LA) performs lexical Analysis

- *LA can catch simple errors like – e.g. illegal symbols*

- LA skips characters in the input until a well-formed token is found
  - This is called "**panic mode**" recovery

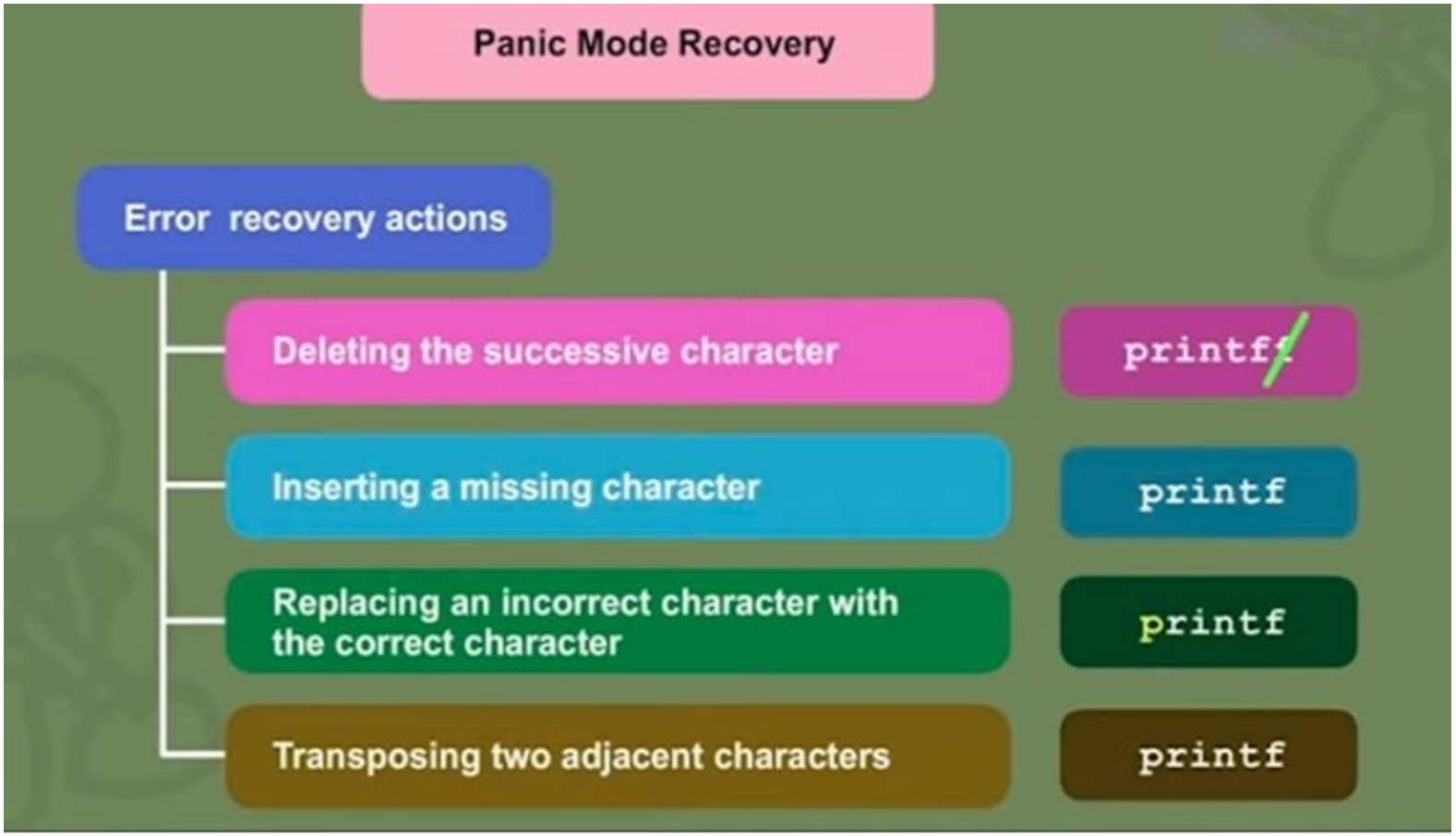
Recovery from Lexical Errors

- **Panic mode**: Successive characters are ignored until we reach to a well formed token

Other possible recovery strategies

- Delete one character from the remaining input, or insert a missing character
- Replace a character, or transpose two adjacent characters
- Idea is to see if a single (or few) transformation(s) can repair the error

Error recovery : Error handling in Lex



Error recovery : Error handling in Lex

- Panic mode: Successive characters are ignored until we reach a well formed token

-Delete one character from the remaining input

e.g. ;scv Valid token scv

-Insert a missing character into the remaining input

e.g prntf - printf

-Replace a character by another character

e.g. whele - while

- Transpose 2 adjacent characters

e.g. fi-if

LEX

BOOK


UNIX Programming Tools
lex & yacc
O'REILLY®
John R. Levine,
Tony Mason & Doug Brown

The Simplest Lex Program

This lex program copies its standard input to its standard output:

```
%%
.|\n        ECHO;
%%
```

Example: Word recognizer

```
%{
/*
 * this sample demonstrates (very) simple recognition:
 * a verb/not a verb.
 */

%}
%%

[\t ]+              /* ignore whitespace */ ;

is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go          { printf("%s: is a verb\n", yytext); }
```

```
[a-zA-Z]+   { printf("%s: is not a verb\n", yytext); }

.|\n        { ECHO; /* normal default anyway */ }
%%

main()
{
      yylex();
}
```

```
% example1
did I have fun?
did: is a verb
I: is not a verb
have: is a verb
fun: is not a verb
?
^D
%
```

Sample – Patterns Actions

```
is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go          { printf("%s: is a verb\n", yytext); }

very |
simply |
gently |
quietly |
calmly |
angrily     { printf("%s: is an adverb\n", yytext); }

to |
from |
behind |
above |
below |
between |
below       { printf("%s: is a preposition\n", yytext); }

if |
then |
and |
but |
or          { printf("%s: is a conjunction\n", yytext); }

their |
my |
your |
his |
her |
its         { printf("%s: is an adjective\n", yytext); }
```

Three parts to Lex

1. **Declarations**

It has global C and Lex declaration

- Regular expression definitions of tokens

```
digit   -->   [0-9]
number  -- > {digit} +
```

2. **Transition Rules**

pattern action

- Regular Expression and Action when matched

```
{number}        { printf("The number is %s\n", yytext); }
junk            { printf("Junk is not a valid input!\n"); }
quit            { return 0; }
```

3. **Auxilliary Procedures**

- Written into the C program…..
- int main() is required

%% separates the three parts

- **Available variables**
  - **`yylval –value associated with token`**
  - **`yytext`** (null terminated string)
  - **`yyleng`** (length of the matching string)

- **Available functions**
  - **`yylex()`** (the primary function generated)-starts the analysis
  - **`input()`** - Returns the next character from the input
  - **`int main(int argc, char *argv[])`**
    - Calls yylex to perform the lexical analysis

LEX program

```
%%
[\t  ]+          {  ;}
isI amIareI wereIwas I be                    {printf("%s: is a verb\nn, yytext); }
[a-zA-Z]+                                      {printf(“%s:is not a verb\n”, yytext);}
.                              {ECHO;}
%%
main()
{
yylex();
}
yywrap()
{}
```

Regular Expression in Lex

- A - matches A
- abc – matches abc
- [abc] - matches a, b or c
- [a-z] - matches a, b or c
- [a-zA-Z]+ - combinations of small and capital alphabets
- [0-9] - matches any digit
- [0-9]+ - matches any integer

How to Use Lex (flex)

- Run Unix “man flex” for full information
- Write regular expressions and actions
- Compile using Lex (flex) tool
  - `lex <prog_name>.l`
  - Results in C code
- Compile using C compiler
  - Link to the lex library
  - `gcc lex.yy.c`
- Run the a.out file and recognize tokens
  - `./a.out < input.text>` /// or on console

Using Regular Expressions in LEX patterns

The characters that form regular expressions are:

- Matches any single character except the newline character ("\n”)
- Matches zero or more copies of the preceding expression

- …contd….

…..The characters that form regular expressions are:

[] A *character class* which matches any character within the brackets. If the first character is a circumflex ("^") it changes the meaning to match any character *except* the ones within the brackets. A dash inside the square brackets indicates a character range, e.g., "[0–9]" means the same thing as "[0123456789]". A "–" or "]" as the first character after the "[" is interpreted literally, to let you include dashes and square brackets in character classes. POSIX introduces other special square bracket constructs useful when handling non-English alphabets. See Appendix H, *POSIX Lex and Yacc*, for details. Other metacharacters have no special meaning within square brackets except that C escape sequences starting with "\" are recognized.

….The characters that form regular expressions are:

^ Matches the beginning of a line as the first character of a regular expression. Also used for negation within square brackets.

$ Matches the end of a line as the last character of a regular expression.

{} Indicates how many times the previous pattern is allowed to match when containing one or two numbers. For example:

A{1,3}

matches one to three occurrences of the letter A. If they contain a name, they refer to a substitution by that name.

\ Used to escape metacharacters, and as part of the usual C escape sequences, e.g., "\n" is a newline character, while "*" is a literal asterisk.

+ Matches one or more occurrence of the preceding regular expression. For example:

```
[0-9]+
```

matches "1", "111", or "123456" but not an empty string. (If the plus sign were an asterisk, it would also match the empty string.)

? Matches zero or one occurrence of the preceding regular expression. For example:

```
-?[0-9]+
```

matches a signed number including an optional leading minus.

| Matches either the preceding regular expression or the following regular expression. For example:

```
cow|pig|sheep
```

matches any of the three words.

"..." Interprets everything within the quotation marks literally—meta-characters other than C escape sequences lose their meaning.

/ Matches the preceding regular expression but only if followed by the following regular expression. For example:

```
0/1
```

matches "0" in the string "01" but would not match anything in the strings "0" or "02". The material matched by the pattern following the slash is not "consumed" and remains to be turned into subsequent tokens. Only one slash is permitted per pattern.

() Groups a series of regular expressions together into a new regular expression. For example:

```
(01)
```

represents the character sequence 01. Parentheses are useful when building up complex patterns with *, +, and ?.

….The characters that form regular expressions are:

Examples of Regular Expressions

We are ready for some examples. First, we've already shown you a regular expression for a "digit":

```
[0-9]
```

We can use this to build a regular expression for an integer:

```
[0-9]+
```

We require at least one digit. This would have allowed no digits at all:

```
[0-9]*
```

Let's add an optional unary minus:

```
-?[0-9]+
```

We can then expand this to allow decimal numbers. First we will specify a decimal number (for the moment we insist that the last character always be a digit):

```
[0-9]*\.[0-9]+
```

Notice the "\" before the period to make it a literal period rather than a wild card character. This pattern matches "0.0", "4.5", or ".31415". But it won't match "0" or "2". We'd like to combine our definitions to match them as well. Leaving out our unary minus, we could use:

```
([0-9]+)|([0-9]*\.[0-9]+)
```

Examples of Regular Expressions

We use the grouping symbols "()" to specify what the regular expressions are for the "|" operation. Now let's add the unary minus:

```
-?(([0-9]+)|([0-9]*\.[0-9]+))
```

We can expand this further by allowing a float-style exponent to be specified as well. First, let's write a regular expression for an exponent:

```
[eE][-+]?[0-9]+
```

This matches an upper- or lowercase letter E, then an optional plus or minus sign, then a string of digits. For instance, this will match "e12" or "E-3". We can then use this expression to build our final expression, one that specifies a real number:

```
-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
```

Our expression makes the exponent part optional. Let's write a real lexer that uses this expression. Nothing fancy, but it examines the input and tells us each time it matches a number according to our regular expression.

Thank You