

System Programming and Compiler Construction



MODULE 5

COMPILERS : ANALYSIS PHASE

Prof. Sonal Shroff
Computer Engineering Department
TSEC

Syllabus topics

2

- Introduction to Compilers
- Phases of compilers:
- Lexical Analysis
 - Role of Finite State Automata in Lexical Analysis
 - Design of Lexical analyzer, data structures used .
- Syntax Analysis-
 - Role of Context Free Grammar in Syntax Analysis
 - Types of Parsers:
 - Top down parser- LL(1)
 - Bottom up parser- SR Parser
 - Operator precedence parser
 - SLR
 - Semantic Analysis
 - Syntax directed definitions.

Introduction

3

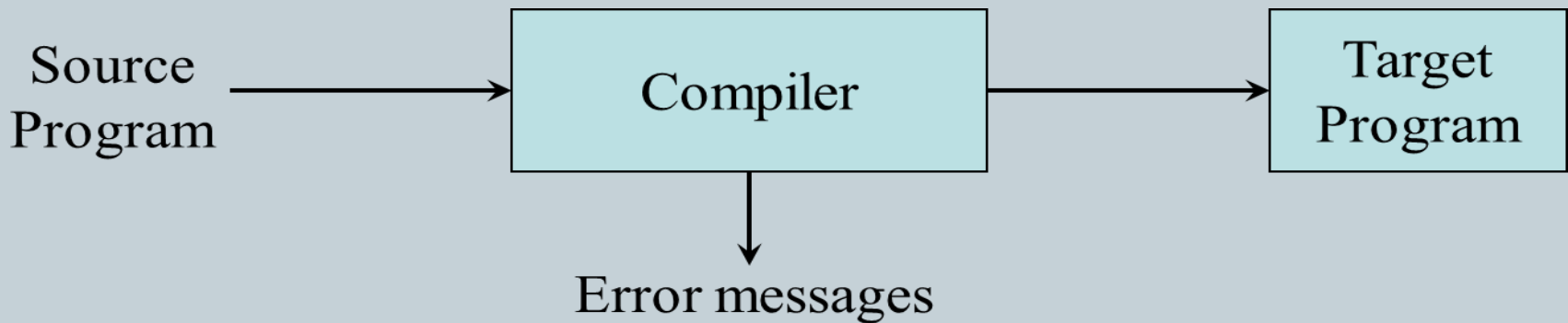
- **Translator**
 - Program that takes input a program written in one programming language and produces output an equivalent program in another programming language.
- **Compiler**
 - If source language is a high level language and output is a low level language, then such a translator is called a compiler.

Introduction

4

- **Compiler**

- Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



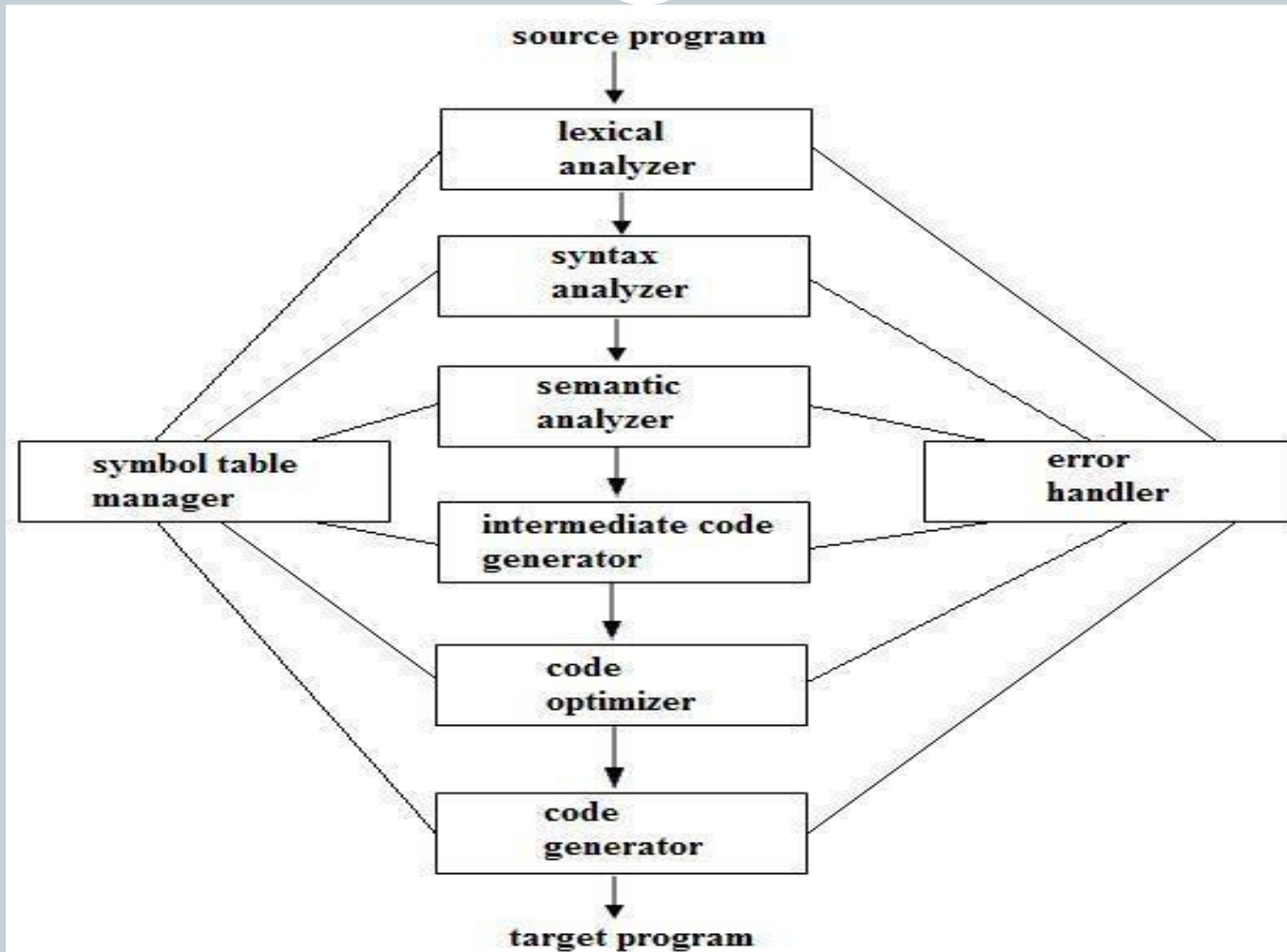
Phases of a Compiler

5

- As compilation is a complex process, it is divided into several phases.
- A phase is a reasonably interrelated process that takes input in one representation and produces the output in another representation.

Phases of a Compiler

6



Phases of a Compiler

7

➤ Lexical analysis phase (Scanning)

- First phase of compiler.
- Reads stream of characters making up source program .
- It groups logically related characters together that are called as **lexemes**.
- For each lexeme, token is generated in the form
 $\langle \text{token-name, attribute-value} \rangle$
- Token types – keywords, identifiers, operators, punctuations, constants

Phases of a Compiler

8

➤ Lexical analysis phase (Scanning)

- For each lexeme, token is generated in the form

`<token-name, attribute-value>`

token-name is the name of the symbol

attribute-value is the location of that token in the symbol table

- It also does the additional job of removing extra whitespace, comments added by the user, etc. from the program
- Passes tokens to next phase (syntax analysis)

Phases of a Compiler

9

➤ Syntax analysis phase (Parsing)

- Syntax analysis
- Semantic analysis

Syntax analysis

- The syntax analysis phase takes tokens from the lexical analyzer and checks the syntactic correctness of the input program

Phases of a Compiler

10

Syntax analysis

- Syntax analysis identifies and notifies the syntactic errors in the program once the program has been broken into the tokens
- It often referred as hierarchical analysis or simply parsing
- The hierarchical structure of the source program can be represented by a parse tree

Phases of a Compiler

11

Semantic analysis

- Once the program is syntactically correct means grammatically correct the next task is to check semantic correctness
- It uses parse tree and symbol table for checking semantic consistency of the language definition of the source program.
- Ex. Checks whether operator has operands of matching type
- It gathers the type information about the program element and saves it in the symbol table.

Phases of a Compiler

12

➤ Intermediate code generation phase

- Here, parse tree representation of the source code is converted into low level intermediate representation
- The intermediate representation should have two important properties:-
 - It should be easy to produce
 - It should be easy to translate into the target program
- There are several forms for representing intermediate code.

Phases of a Compiler

13

➤ Code optimization phase

- An optional phase that performs optimization of the intermediate code.
- The code optimization phase attempts to improve the intermediate code so that a faster running machine code could be generated
- Code optimization is performed to:
 - Minimize the time taken to execute a program
 - Minimize the amount of memory occupied

Phases of a Compiler

14

➤ Code generation phase

- This phase is responsible for the generation of target code (machine language code)
- Memory locations are selected for each variable
- Instructions are translated into a sequence of assembly instructions
- Variables and intermediate results are assigned to memory registers
- The output machine code can be executed directly on the machine

Phases of a Compiler

15

➤ Symbol table management

- **Symbol table** is a data structure used by compiler to record and collect information about all the symbols defined in the source program
- It is used as a reference table by all the phases of a compiler
- The typical information stored in the symbol table includes the name of the variables, their types, sizes relative offset within the program and so on.

Phases of a Compiler

16

➤ Symbol table management

- The generation of this table is normally carried out by the lexical analyzer and syntax analyzer phases.
- It should be designed in an efficient way so that it permits the compiler to locate the record for each token quickly and to allow rapid transfer of data from the records.

Phases of a Compiler

17

➤ Error handler

- It is invoked whenever any fault occurs in the compilation process of source program.
- Error handling is responsible for handling the error which can occur in any of the compilation phase
- After detecting the error, a phase must deal with that error so that compilation can proceed.

Phases of a Compiler

18

➤ Example

$\text{Total} = \text{number1} + \text{number2} * 5$

Lexical Analysis

19

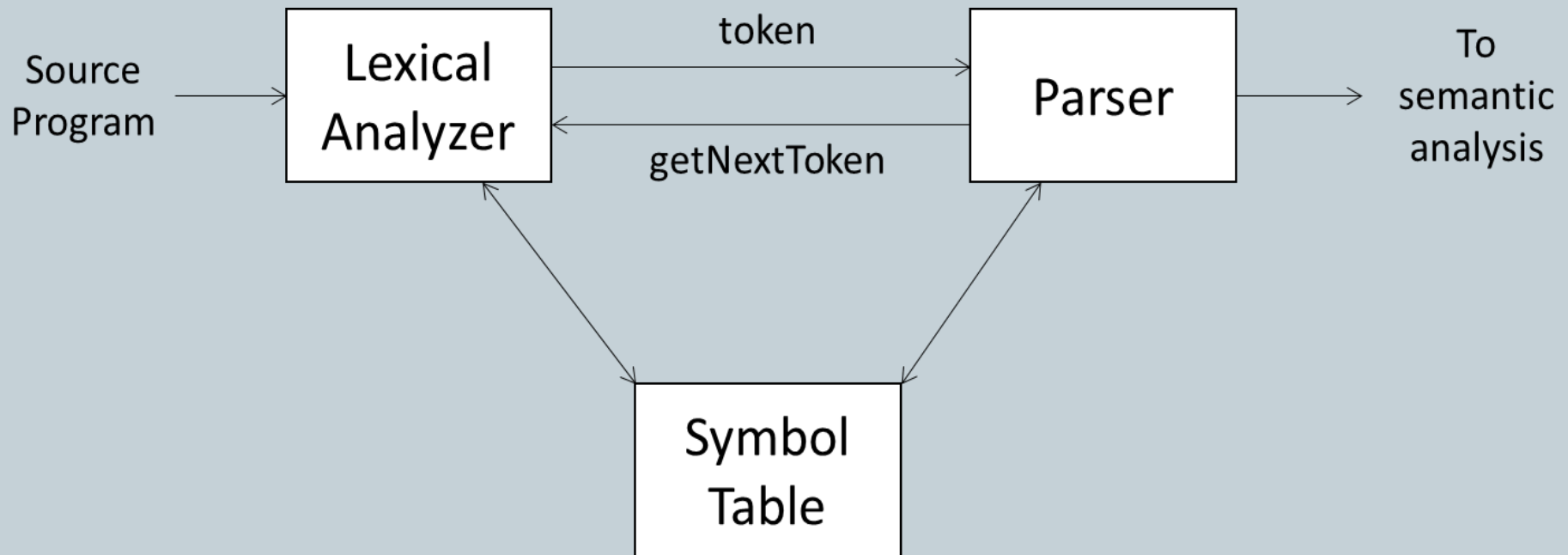
➤ The role of the Lexical Analyzer

- Lexical Analyzer is the first phase of a compiler
- Main Task: to read the input character from source program, group them into lexemes and produce output as sequence of tokens for each lexemes in the source program.
- The stream of tokens is sent to the parser for syntax analysis.
- Lexical analyzer interacts with the symbol table as well.
- Lexical analyzer enters lexemes into the symbol table.

Lexical Analysis

20

➤ The role of the Lexical Analyzer



Interaction between the lexical analyzer and the parser

Lexical Analysis

21

➤ The role of the Lexical Analyzer

- Identification of lexemes
- Stripping out comments and white spaces
- Correlating error messages from the compiler with the source program
- If source language uses some macro-preprocessor, then these macro expansion may also be performed

Lexical Analysis

22

➤ Lexical analyzer is divided into cascade of two processes

1. Scanning :

- Deletion of comments
- Compaction of consecutive whitespace characters into one, etc

2. Lexical analysis

- To produce tokens from the output of the scanner

Lexical Analysis

23

➤ Lexical analysis Vs Parsing

Why analysis portion of a compiler is separated into lexical analysis and parsing (syntax analysis)

1. Simplicity of design is the most important consideration
2. Compiler efficiency is improved
3. Compiler portability is enhanced

Lexical Analysis

24

➤ Tokens Patterns and Lexemes

Token – a pair consisting of token name and an optional attribute value.

Token names are abstract symbol representing a kind of lexical unit.

Attribute value is commonly referred to as token value.

Each token represents a sequence of characters that can be treated as a single entity.

Ex. Identifiers, keywords, constants, operators, punctuation symbols.

Lexical Analysis

25

➤ Tokens Patterns and Lexemes

Token – broadly classified into 2 types

1. Specific strings such as if, else, comma, semicolon etc.
2. Classes of strings such as identifiers, constants, labels etc.

Ex. $A = B + C * 5$

After lexical analysis

$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 5 \rangle$

Lexical Analysis

26

➤ Tokens Patterns and Lexemes

Pattern – a rule that defines a set of input strings for which the same token is produced as output.

Regular expressions play an important role in specifying patterns

For keyword, pattern is – sequence of characters.

For identifier, pattern is – letter (letter/ digit)*

Lexeme – a group of logically related characters in the source program that matches the pattern for a token.

It is identified as an instance of that token.

Lexical Analysis

27

➤ Examples of tokens, patterns and lexemes

Token	Informal Description	Sample Lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or <= or = or <> or >= or >	<, <=, =, <>, >, >=
id	Letter followed by letters and digits	Pi, count, D2
number	Any Numeric Constant	3.1416, 0, 6.02E23
literal	Any character between " and "	"core dumped"

Lexical Analysis

28

➤ Classes that cover most of the Tokens

- Keyword – The pattern for a keyword is the same as the keyword itself.
- Operators – either individually or in classes.
- Identifier – one token for all identifiers
- Constants – one or more tokens representing constants such as numbers and literal strings
- Punctuation symbols – tokens for parenthesis, comma, semicolon, etc.

Lexical Analysis

29

➤ Example

```
int add(int a, int b)
//addition of two
numbers
{
    int c;
    c=a+b;
    return c;
}
```

S.No.	Lexeme	Token	S.No.	Lexeme	Token
1.	int	Keyword	13.	;	Punctuation
2.	add	Identifier	14.	c	Identifier
3.	(Punctuation	15.	=	Operator
4.	int	Keyword	16.	a	Identifier
5.	a	Identifier	17.	+	Operator
6.	,	Punctuation	18.	b	Identifier
7.	int	Keyword	19.	;	Punctuation
8.	b	Identifier	20.	return	Keyword
9.)	Punctuation	21.	c	Identifier
10.	{	Punctuation	22.	;	Punctuation
11.	int	Keyword	23.	}	Punctuation
12.	c	Identifier			

Lexical Analysis

30

➤ Attributes for tokens

When more than one lexeme can match a pattern, then lexical analyzer must provide additional information about the lexeme.

Lexical analyzer provides token name and its attribute value.

The attribute value describes the lexeme represented by the token.

Token name influences parsing decisions.

Attribute value influences the translation of token after the parse.

Lexical Analysis

31

➤ Attributes for tokens

A token has a single attribute – A pointer to the symbol table entry in which the information about the token is kept

The pointer becomes attribute for the token

The attribute may have a structure that combines the several pieces of information.

Example – token **identifier**

Lexical Analysis

32

➤ Attributes for tokens

Example – token **identifier**

Information about an identifier – its lexeme, its type, its location etc is stored in symbol table.

Thus, the appropriate attribute value for an identifier is a pointer to the symbol table entry for that identifier.

Lexical Analysis

33

$$E = M * C ** 2$$

Token names and attributes for above statements

1. <id, pointer to symbol-table entry for E>
2. <assign_op,>
3. <id, pointer to symbol-table entry for M>
4. <mult_op,>
5. <id, pointer to symbol-table entry for C>
6. <exp_op,>
7. <num, integer value 2>

Lexical Analysis

34

➤ Lexical errors

Error Recovery Actions

1. Panic Mode Recovery- Delete successive characters from the remaining input until the lexical analyzer finds a well-formed token
2. Deleting one character from remaining input
3. Inserting a missing character into the remaining input
4. Replacing an incorrect character by a correct character
5. Transposing two adjacent characters

Lexical Analysis

35

➤ Input Buffering

- **Scanner** is the only part of the compiler which reads a complete program.
- Lexical Analyzer is the only phase of the compiler that reads the source program character by character
- It takes approx. 25-30% of the compiler's time.
- So, Lexical Analysis phase consumes considerable amount of time, due to which, compilation time goes low.
- Hence, speed of Lexical Analysis is a major concern!

Lexical Analysis

36

➤ Input Buffering

As large amount of time consumption takes place in moving characters, Specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character by lexical Analyzer.

A block of data is first read into a BUFFER and then it is read by Lexical Analyzer

Lexical Analysis

37

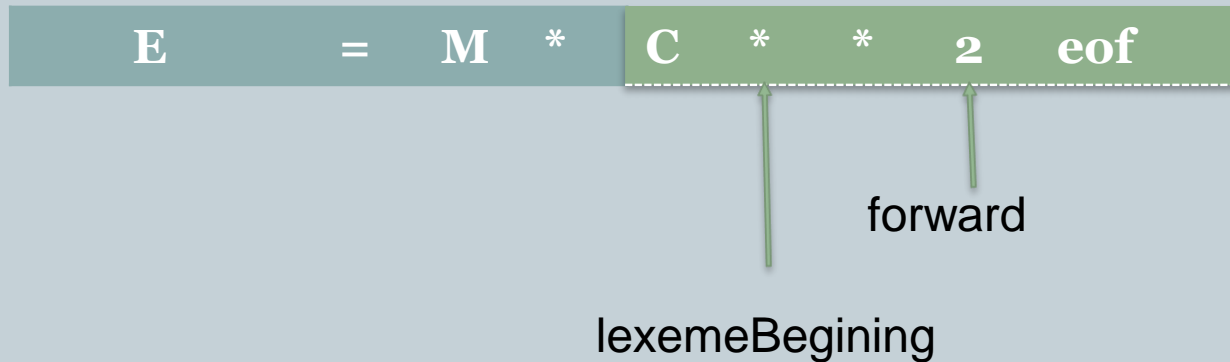
➤ Buffer Pairs

- Using one system read command we can read N characters into a buffer, rather than using one system call per character.
- If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file.
- “eof” is different from any possible character of the source program.

Lexical Analysis

38

➤ Buffer Pairs



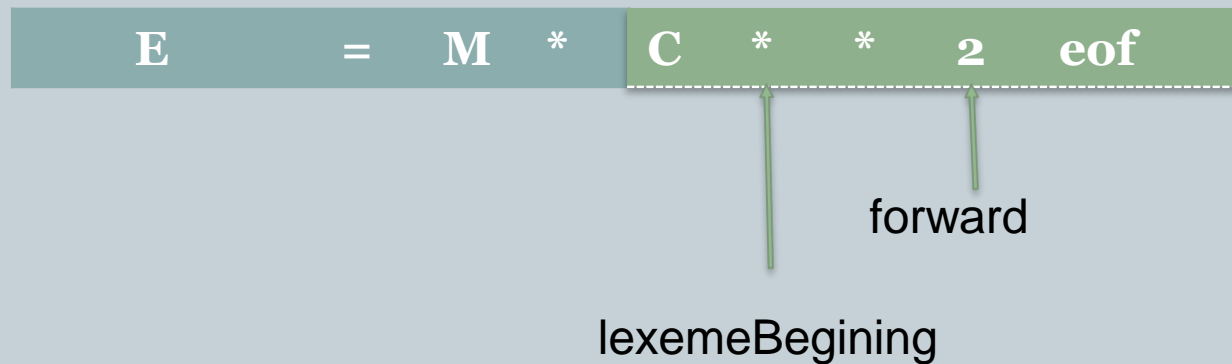
Each buffer is of the same size N.

N is usually the size of the disk block. Eg. 4096 bytes

Lexical Analysis

39

➤ Buffer Pairs



Two pointers are maintained

- Pointer **lexemeBegin** marks the beginning of the current lexeme, whose extent we are attempting to determine.
- Pointer **forward** scans ahead until a pattern match is found.

Lexical Analysis

40

➤ Buffer Pairs

- Once the next lexeme is determined, forward is set to the character at its right end.
- The lexeme is recorded as an attribute value of a token returned to the parser, lexemeBegin is set to the character immediately after the lexeme just found

Lexical Analysis

41

➤ Buffer Pairs

E	=	M	*	C	*	*	2	eof
---	---	---	---	---	---	---	---	-----

Code to advance forward pointer

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;

lexemeBeginning

forward

Lexical Analysis

42

➤ Sentinels

- In Two Buffer Scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.
 1. For end of buffer.
 2. To determine what character is read.

The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.

The sentinel is a special character that cannot be part of the source program. (***eof* character is used as sentinel**)

Lexical Analysis

43

E

=

M

*

eof

C

*

*

2

eof

eof

lexemeBeginning

forward

```
forward:=forward+1
if forward= eof then begin
    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 /* eof within a buffer signifying end of input */
        terminate
end
```

Lexical Analysis

44

➤ Specification of tokens

Alphabet

- Any finite set of symbols – Letters, digits and punctuation
- $\{0,1\}$ – binary alphabet

String

String over an alphabet is a finite sequence of symbols drawn from that alphabet.

- “Compiler” is a string of length eight. ($|s| = 8$)
- The empty string, denoted ϵ , is the string of length zero

Lexical Analysis

45

➤ Specification of tokens

Terms for parts of String

- A prefix of string s – any string obtained by removing zero or more symbols from the end of s . ex. ban, banana, ϵ are prefixes of banana.
- A suffix of string s – any string obtained by removing zero or more symbols from the beginning of s . ex. ana, banana, ϵ are suffixes of banana
- A substring of s – any string obtained by deleting any prefix and any suffix from s . ex. nan, banana, ϵ are substrings of banana

Lexical Analysis

46

➤ Specification of tokens

Terms for parts of String

- The proper prefixes, suffixes and substrings of s are those, prefixes, suffixes and substrings, respectively, of s that are not ϵ or not equal to s itself.
- A subsequence of s – any string formed by deleting zero or more not necessarily consecutive positions of s . ex. baan is a subsequence of banana.

Lexical Analysis

47

➤ Specification of tokens

Language

It is any countable set of strings over some fixed alphabet.

Abstract languages like \emptyset , the empty set, or $\{\epsilon\}$, the set containing only the empty string.

The meaning to the string is not the requirement here.

Lexical Analysis

48

➤ Specification of tokens

Operations on Languages

In lexical analysis, the most important operations on languages are

Operation	Definition
Union	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation	$L . M = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = L^0 \cup L^1 \cup L^2 \dots$
Positive Closure of L	$L^+ = L^1 \cup L^2 \cup L^3 \dots$

Lexical Analysis

49

➤ Specification of tokens

Operations on Languages

Let L be the set of letters $\{A, B, \dots, Z, a, b, \dots, z\}$ and D be the set of digits $\{0, 1, \dots, 9\}$

L and D are the alphabets upper and lower case letters and of digits.

OR L and D are languages, all of whose strings are of length one.

Possible Operations:

1. $L \cup D$ is the set of letters and digits
2. LD is the set of string consisting of one letter followed by one digit
3. L^4 is the set of all four letter string
4. L^* is the set of all strings of letters including empty string ϵ
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with letter
6. D^+ is the set of all strings of one or more digits

Lexical Analysis

50

➤ Regular Expressions

Let $\Sigma = \{ a, b \}$

- The regular expression $a|b$ denotes the set $\{a, b\}$
- The regular expression $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$. The set of all strings of a's and b's of length two
- The regular expression a^* denotes set of all strings of zero or more a's.

$$r = \{\epsilon, a, aa, aaa, aaaa, \dots\}$$

- If two regular expressions represents same language then we can say that they are equivalent

Lexical Analysis

51

➤ Regular Expressions

Regular expression are used to specify lexeme patterns.

- letter $\rightarrow A|B|.....|Z|a|b|.....|z$
- letter_ $\rightarrow [A-Za-z_]$
- digit $\rightarrow 0|1|2|.....|9$
- digit $\rightarrow [0-9]$
- id $\rightarrow \text{letter_}(\text{letter_} | \text{digit})^*$
- digits $\rightarrow \text{digit}^*(.\text{digits})?(E[+-]? \text{digits})?$
- op $\rightarrow < | > | <= | >= | = | <>$
- ws $\rightarrow (\text{blank}|\text{tab}|\text{newline})^+$

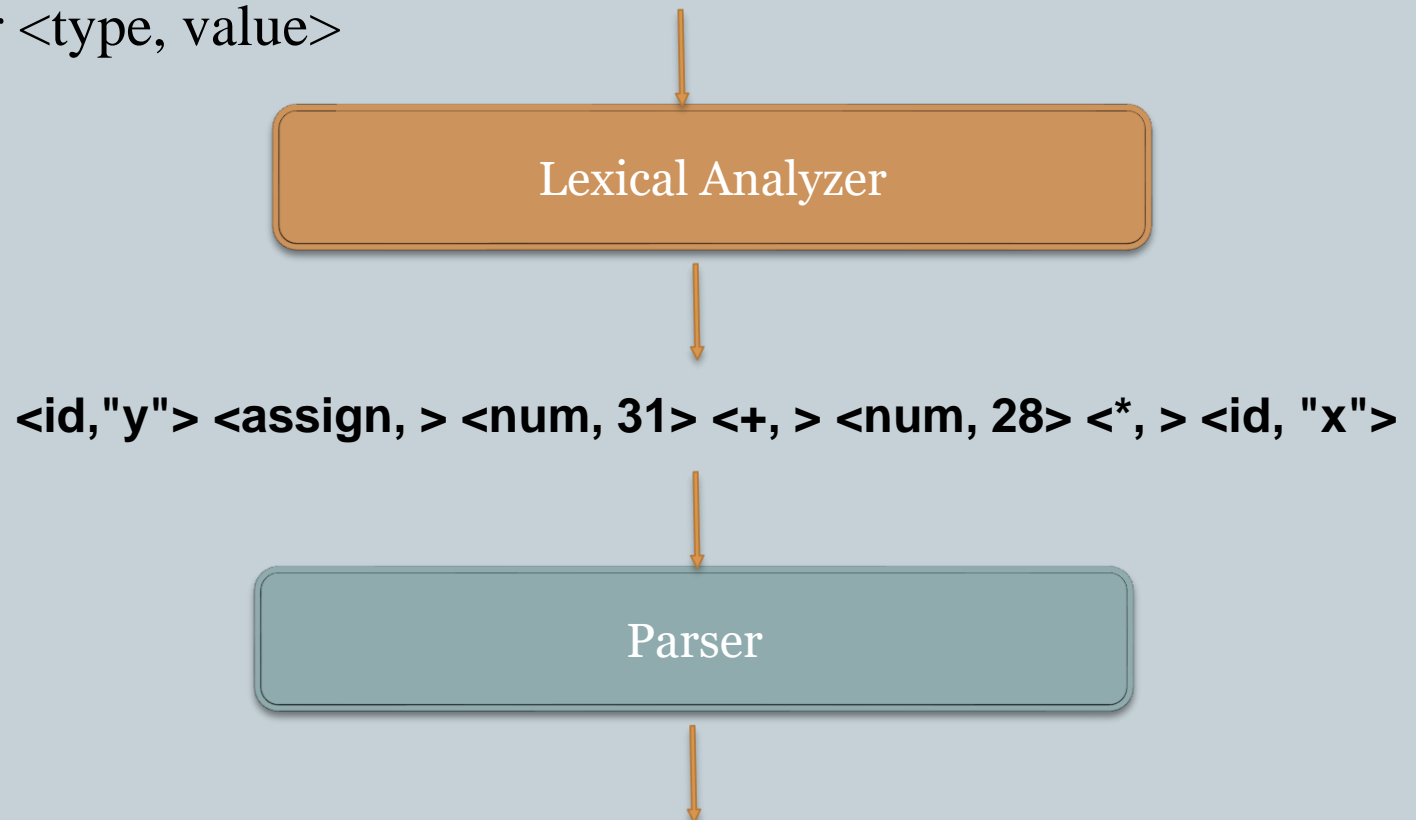
Lexical Analysis

52

➤ Recognition of Tokens

$y = 31 + 28 * x$

Token is a pair <type, value>



Lexical Analysis

53

➤ Recognition of Tokens

A Grammar for branching statement

stmt \rightarrow **if** expr **then** stmt

| if expr **then** stmt **else** stmt

| ϵ

expr \rightarrow term **relop** term

| term

term \rightarrow **id**

| **number**

Lexical Analysis

54

➤ Pattern of Tokens

Token	Pattern
digit	[0–9]
digits	digit+
number	digits (.digits) ? (E [+–] ? digits)?
letter	[A-Za-z]
id	letter (letter digit)*
if	if
then	then
else	else
relop	< <= > >= = <>

Lexical Analysis

Lexemes	Token Name	Attribute Value
Any ws	--	--
if	if	--
else	else	--
then	then	--
id	id	Pointer to table entry
num	num	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Lexical Analysis

56

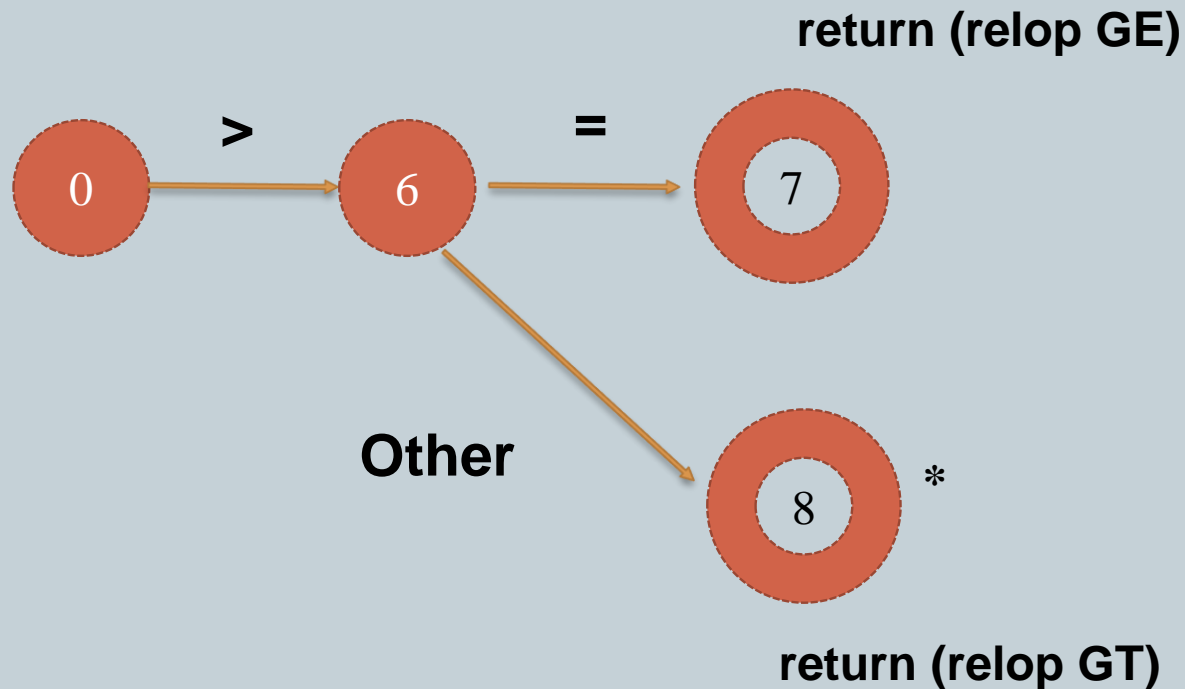
➤ Transition Diagrams

- We convert patterns into stylized flowcharts, called "transition diagrams"
- Transition diagrams have a collection of nodes or circles, called states
- Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns
- Edges are directed from one state of the transition diagram to another
- Each edge is labeled by a symbol or set of symbols

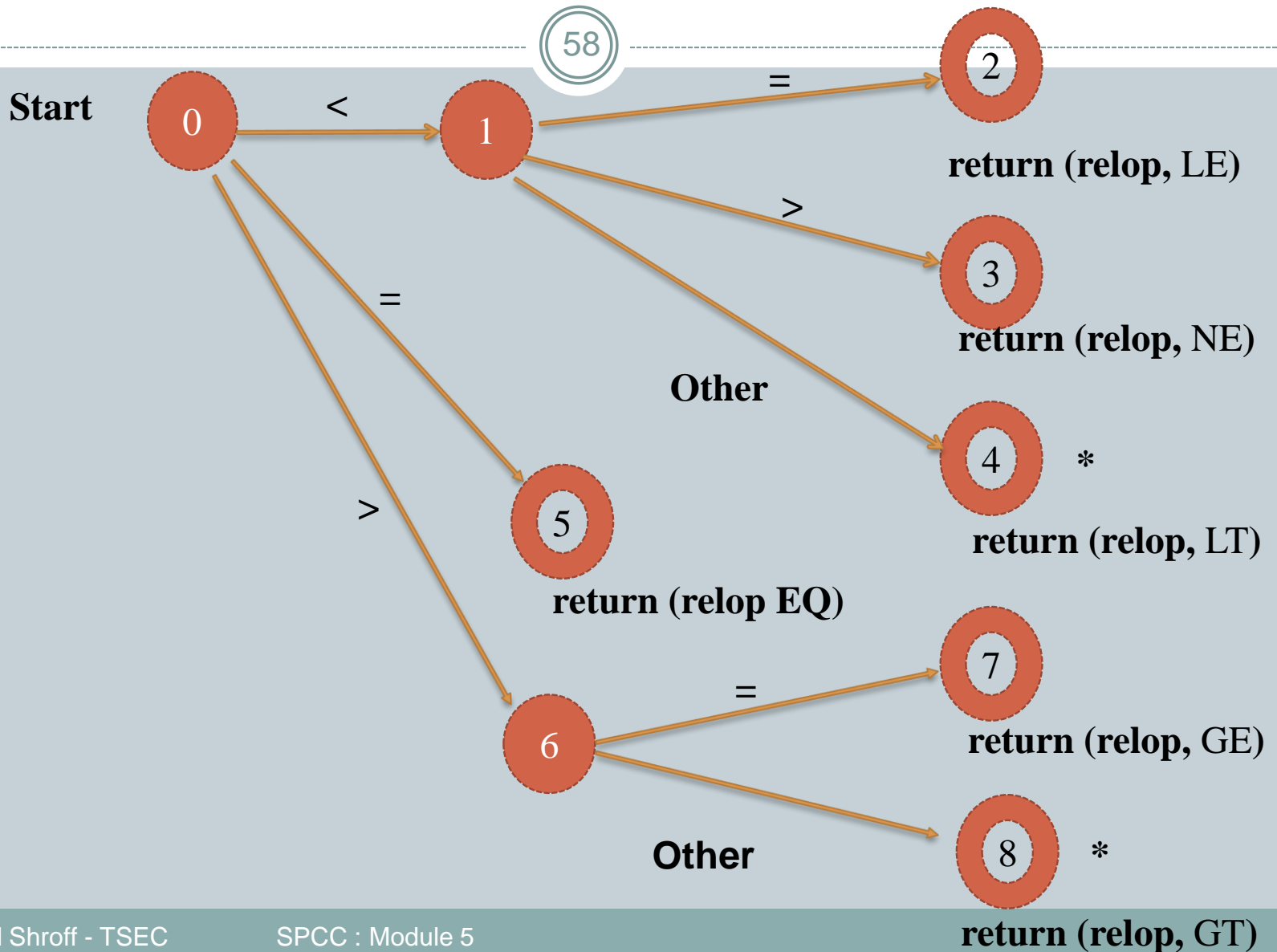
Lexical Analysis

57

Transition Diagram for \geq



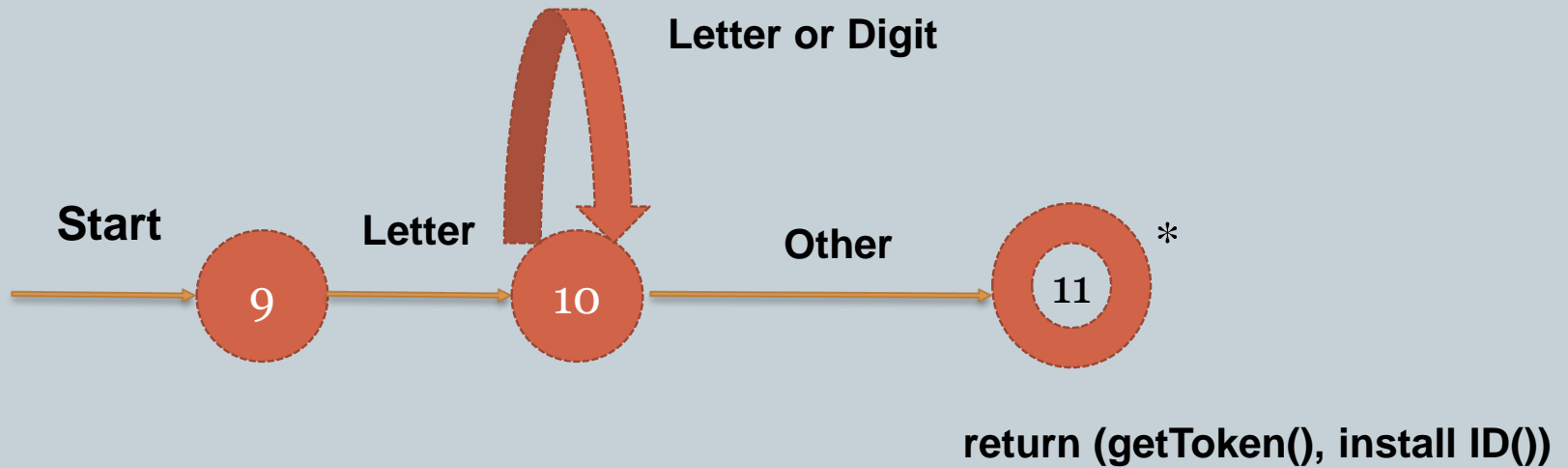
Transition Diagram for **relop**



Lexical Analysis

59

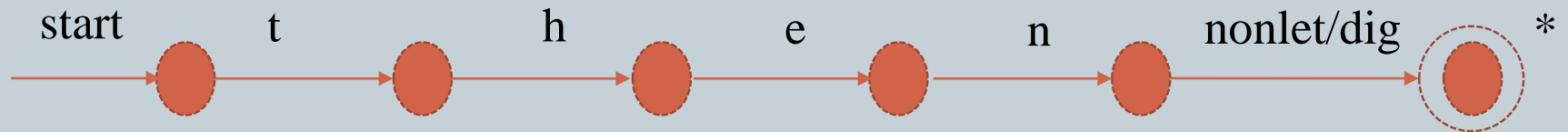
➤ Transition Diagram for identifiers and keywords



Lexical Analysis

60

➤ Hypothetical Transition Diagram for keyword 'then'



Lexical Analysis

61

➤ **Transition Diagram for unsigned number**