System Programming and Compiler Construction

MODULE 5

COMPILERS : ANALYSIS PHASE

Prof. Sonal Shroff
Computer Engineering Department
TSEC

Syllabus topics

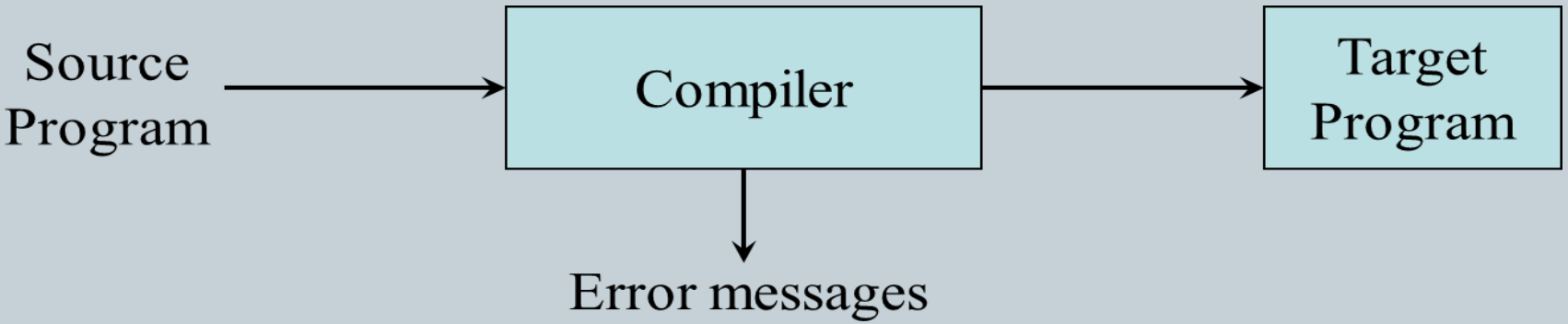
- Introduction to Compilers
- Phases of compilers:
- Lexical Analysis
  - Role of Finite State Automata in Lexical Analysis
  - Design of Lexical analyzer, data structures used .
- Syntax Analysis-
  - Role of Context Free Grammar in Syntax Analysis
  - Types of Parsers:
    - Top down parser- LL(1)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

Introduction

- 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

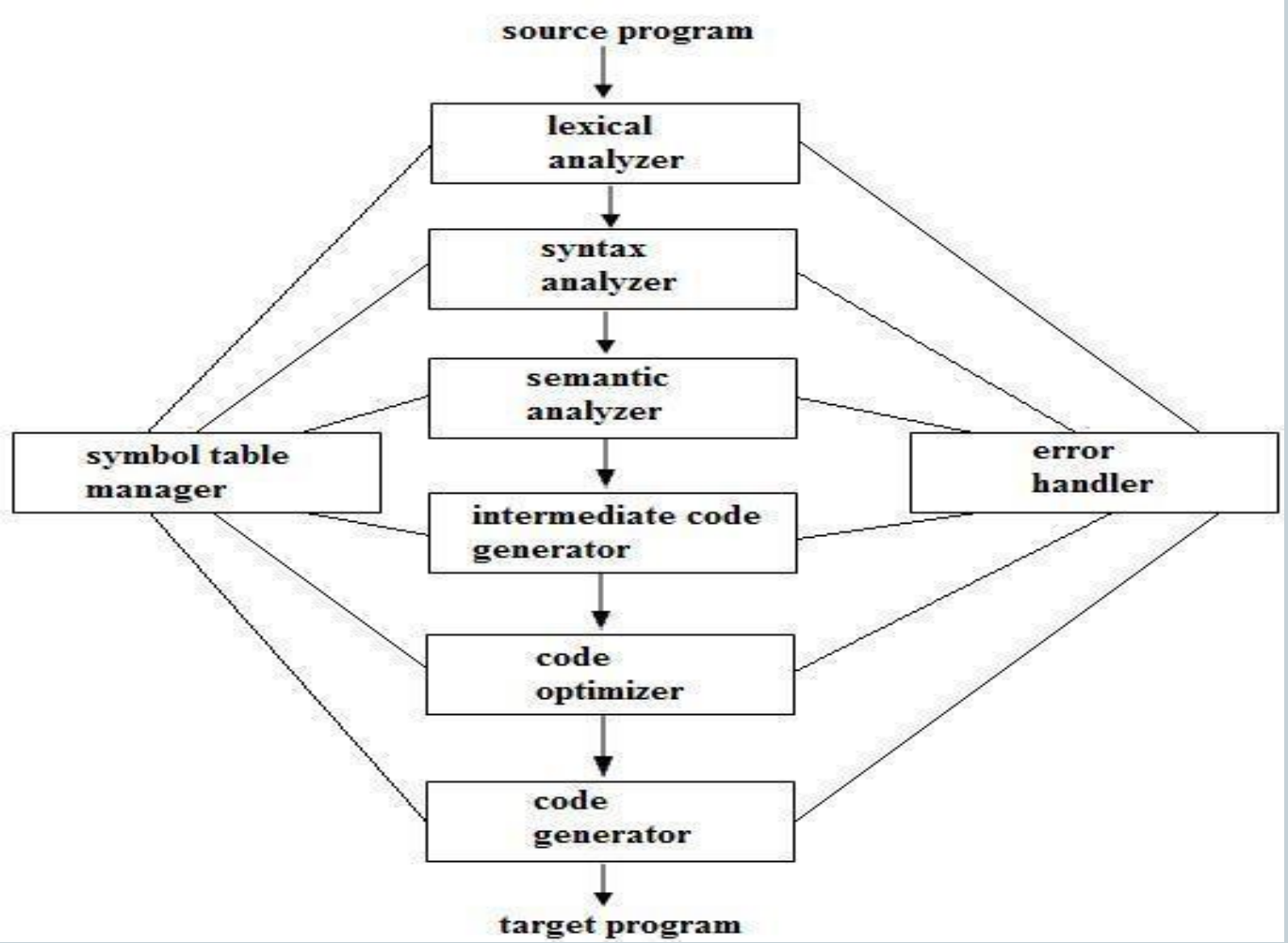
- Compiler
  - Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



Phases of a Compiler

- 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



Phases of a Compiler

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

  <token-name, attribute-value>
- Token types – keywords, identifiers, operators, punctuations, constants

Phases of a Compiler

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

- 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

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

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

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

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

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

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

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

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

- Example

Total = number1 + number2 * 5

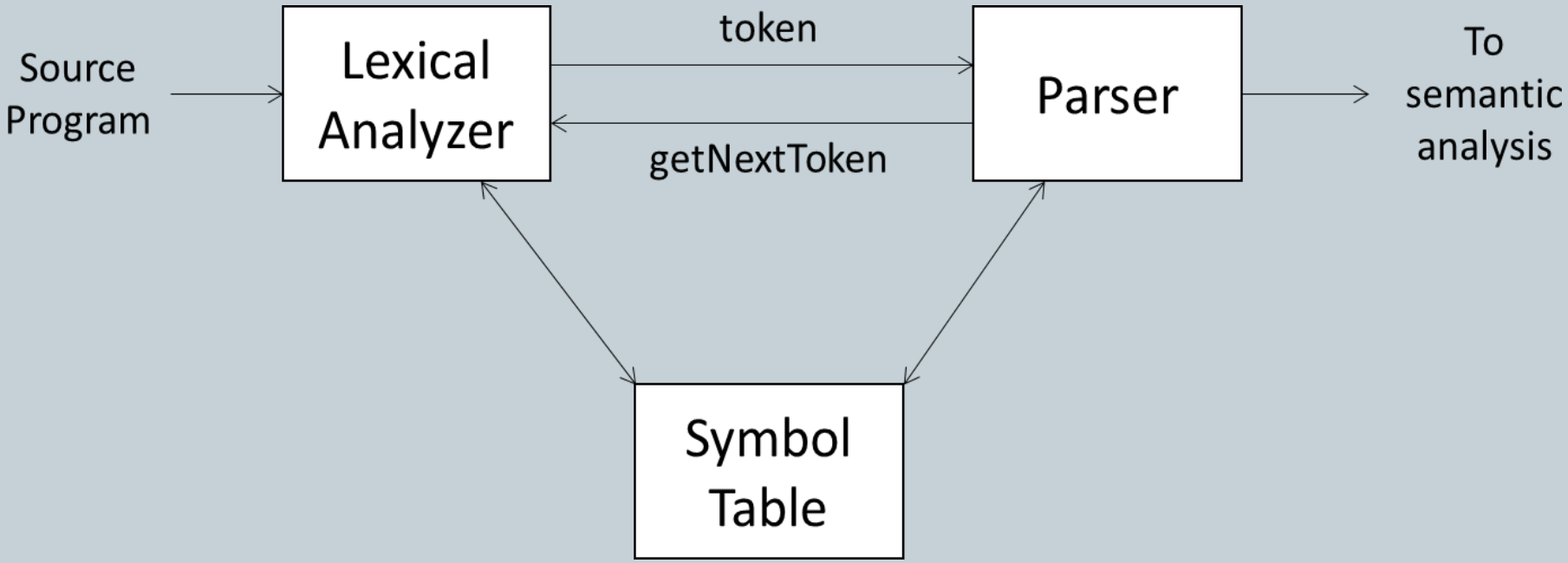
Lexical Analysis

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

- The role of the Lexical Analyzer



Interaction between the lexical analyzer and the parser

Lexical Analysis

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

- 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

- 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

- 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

- 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

<id,1> <=> <id,2> <+> <id,3> <*> <5>

Lexical Analysis

- 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

- 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

- 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

- Example

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

S.No.	Lexeme	Token
1.	int	Keyword
2.	add	Identifier
3.	(	Punctuation
4.	int	Keyword
5.	a	Identifier
6.	,	Punctuation
7.	int	Keyword
8.	b	Identifier
9.	)	Punctuation
10.	{	Punctuation
11.	int	Keyword
12.	c	Identifier

S.No.	Lexeme	Token
13.	;	Punctuation
14.	c	Identifier
15.	=	Operator
16.	a	Identifier
17.	+	Operator
18.	b	Identifier
19.	;	Punctuation
20.	return	Keyword
21.	c	Identifier
22.	;	Punctuation
23.	}	Punctuation

Lexical Analysis

- 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

- 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

- 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

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

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

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

- 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

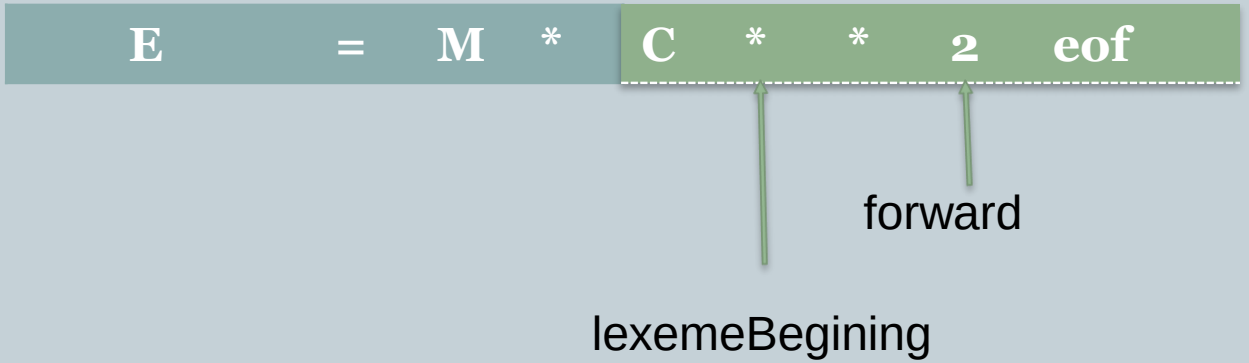
Lexical Analysis

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

- Buffer Pairs

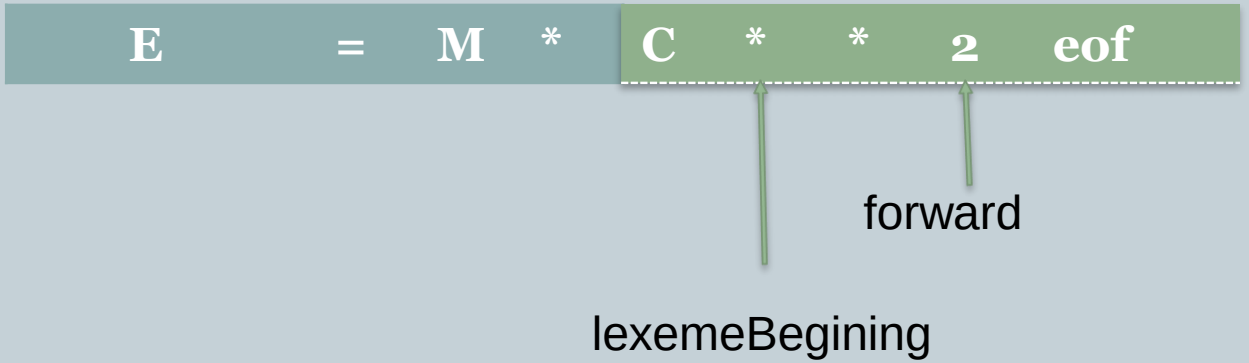


Each buffer is of the same size N.

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

Lexical Analysis

- 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.

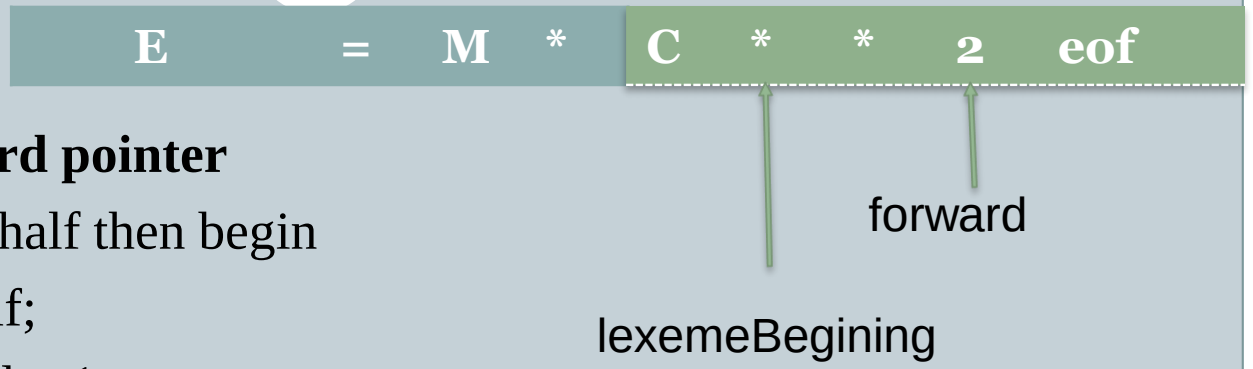
Lexical Analysis

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

- Buffer Pairs



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;
```

Lexical Analysis

Sentinels

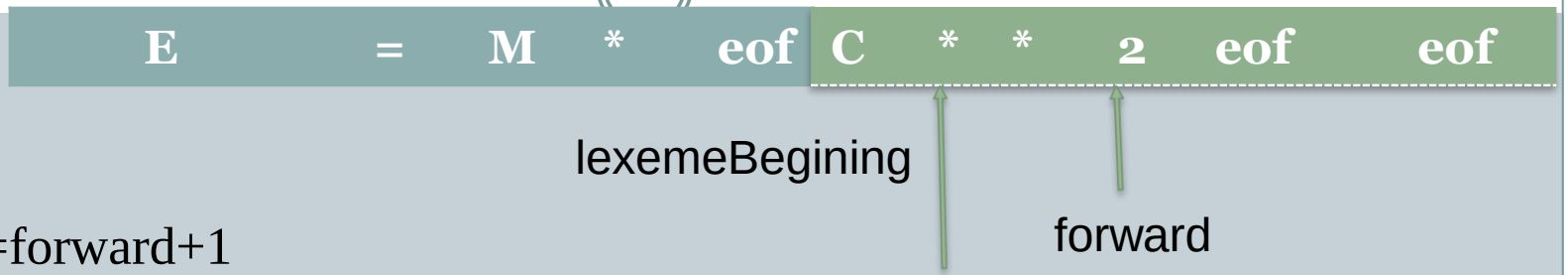
- In Two Buffer Scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

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




```
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

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

- 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 sufixes 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

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

- Specification of tokens

Language

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

Abstract languages like ø, the empty set, or {ε}, the set containing only the empty string.

The meaning to the string is not the requirement here.

Specification of tokens

Operations on Languages

In lexical analysis, the most important operations on languages are

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

Lexical Analysis

Specification of tokens

Operations on Languages

Let L be the set of letters {A, B,.....Z,a,b,.....z}and D be the set of digits{0,1,....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 U 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 U 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

Regular Expressions

Let ∑ = { 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 = {ε, a, aa, aaa, aaaa, ... }
- If two regular expressions represents same language then we can say that they are equivalent

Lexical Analysis

Regular Expressions

Regular expression are used to specify lexeme patterns.

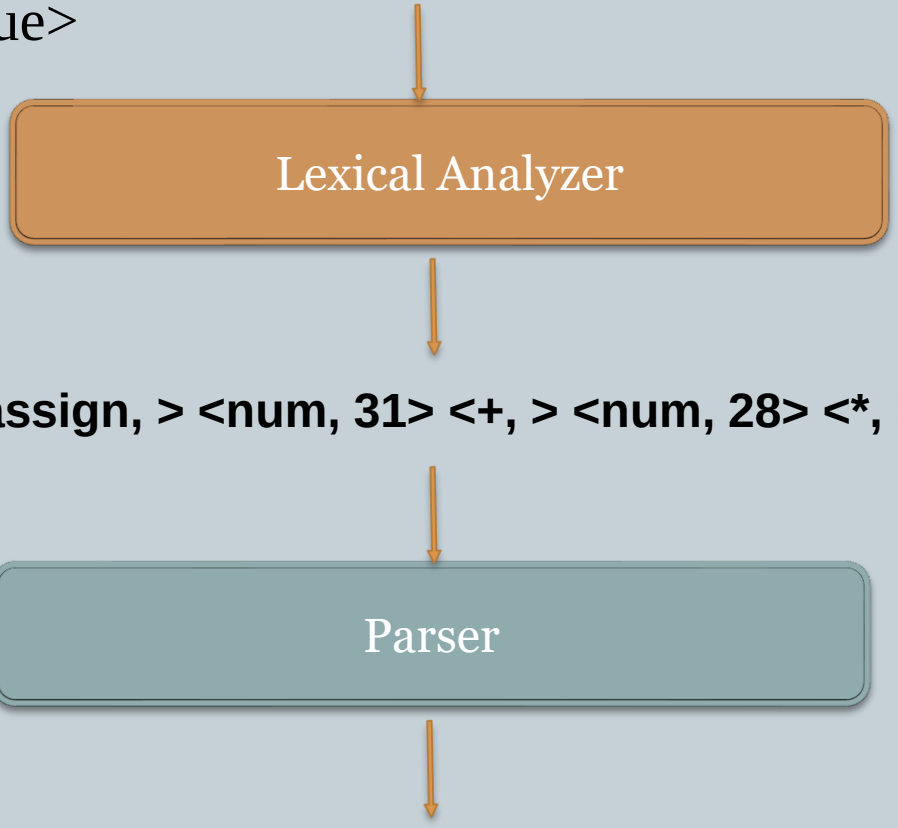
- letter → A|B|……|Z|a|b|……|z
- letter_ → [A-Za-z_]
- digit → 0|1|2|……|9
- digit → [0-9]
- id → letter_ (letter_ | digit)*
- digits → digit* (.digits)? (E[+-]? digits)?
- op → < | > | <= | >= | = | <>
- ws → (blank|tab|newline)+

Lexical Analysis

- Recognition of Tokens

y = 31 + 28 * x

Token is a pair <type, value>



Lexical Analysis

- Recognition of Tokens

A Grammar for branching statement

stmt -> **if** expr **then** stmt

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

| ε

expr -> term **relop** term

| term

term -> **id**

| **number**

Lexical Analysis

- 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

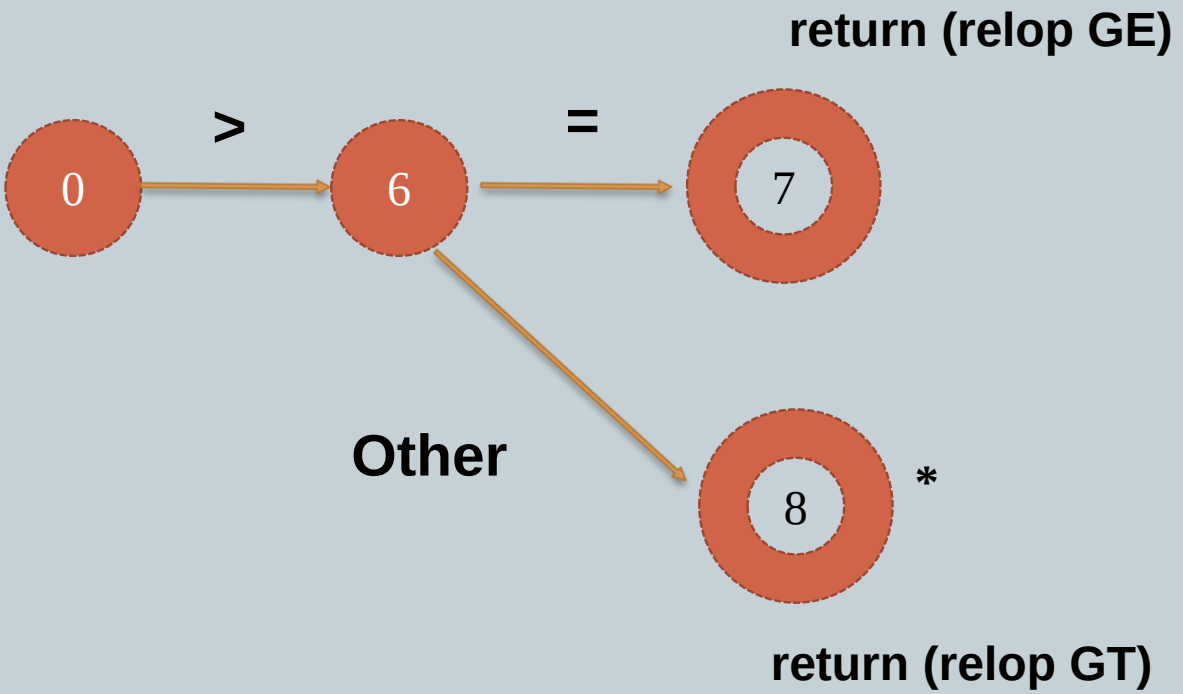
Lexemes	Token Name	Attribute Value
Any ws	--	--
if	if	--
else	else	--
then	then	--
id	id	Pointer to table entry
num	num	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Lexical Analysis

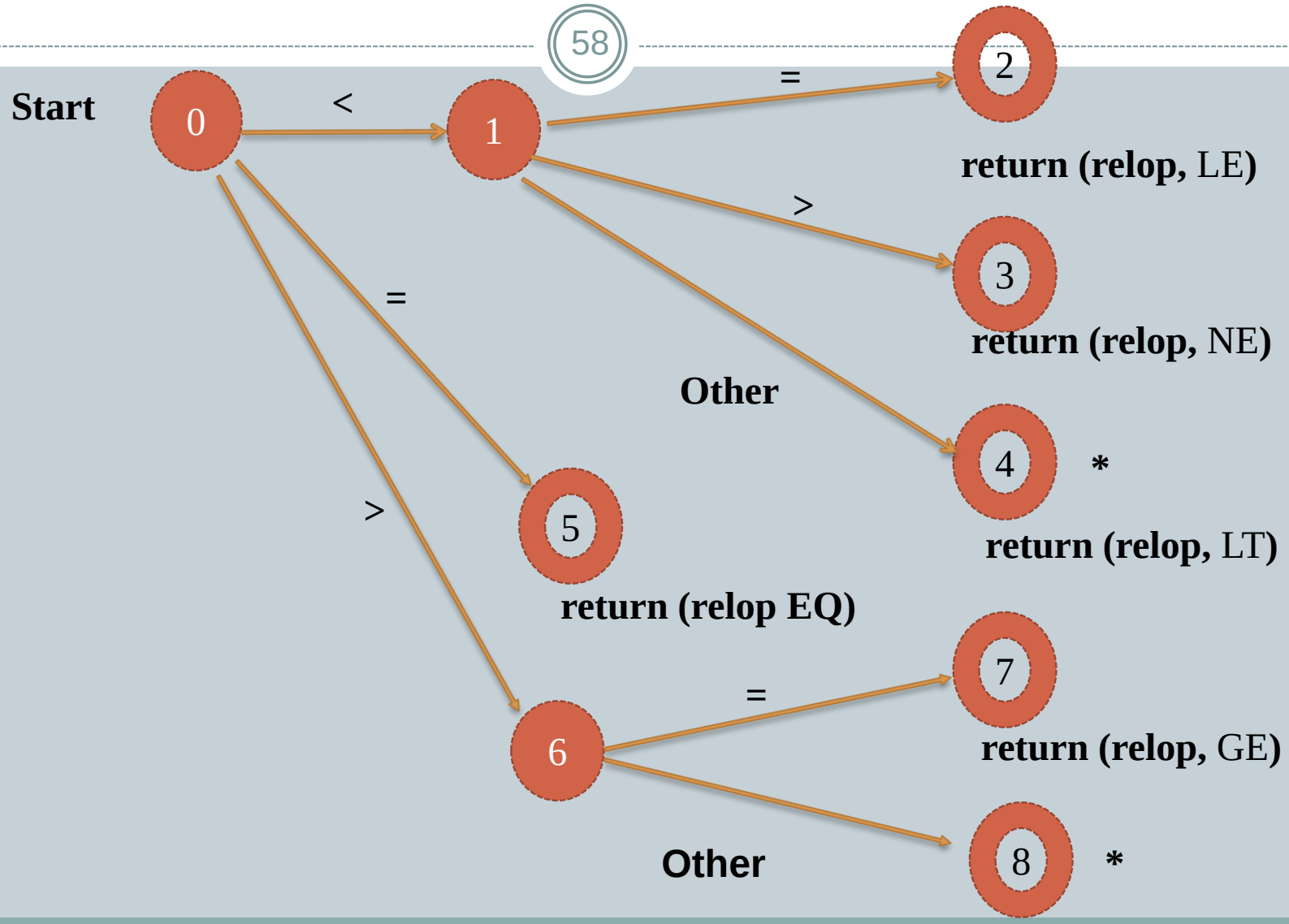
Transition Diagrams

- 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

Transition Diagram for >=

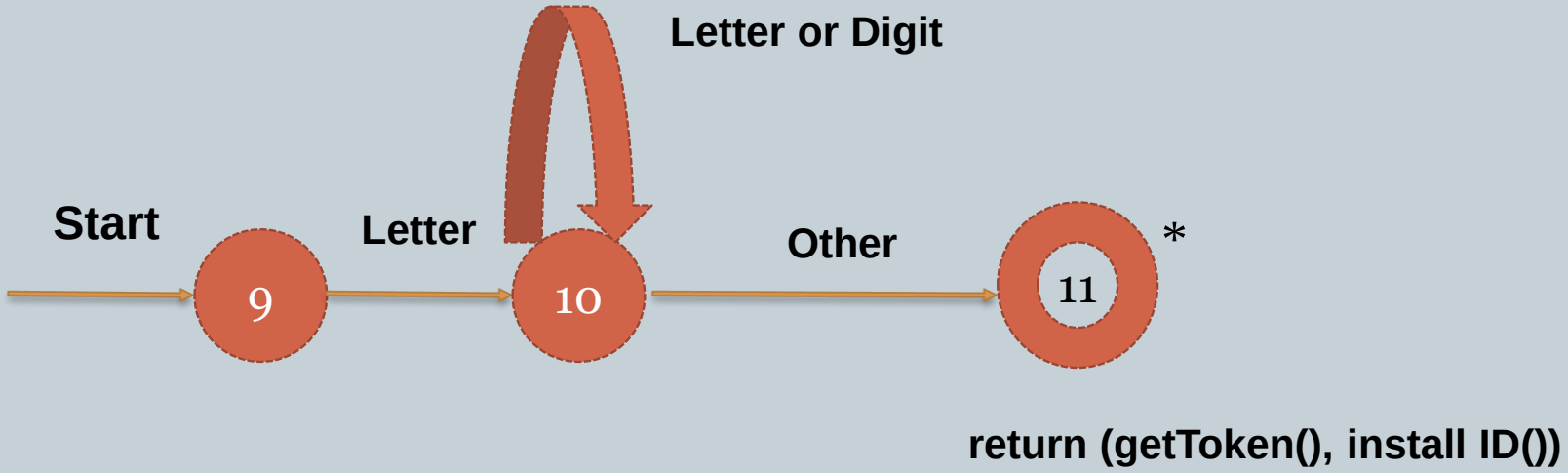


Transition Diagram for **relop**



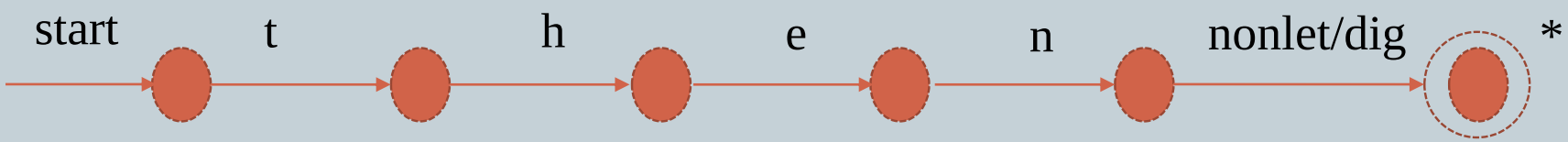
Lexical Analysis

- **Transition Diagram for identifiers and keywords**



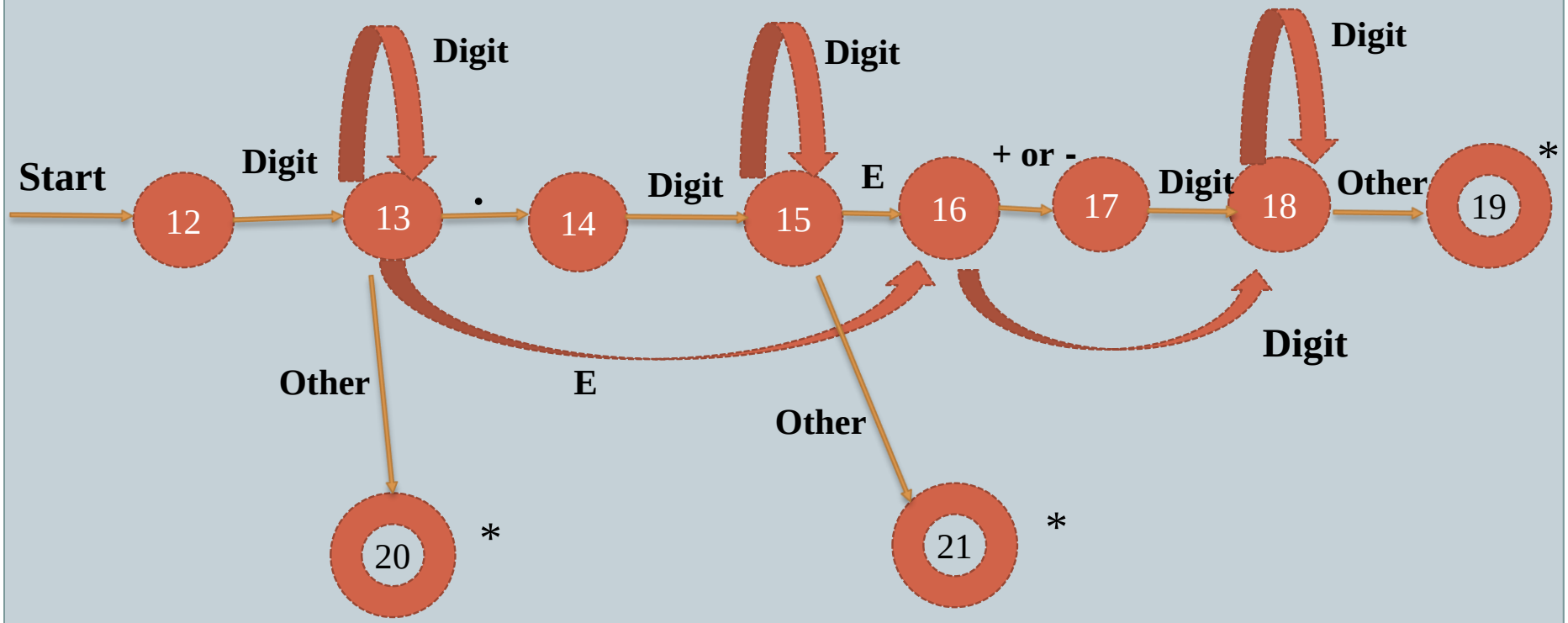
Lexical Analysis

- **Hypothetical Transition Diagram for keyword ‘then’**

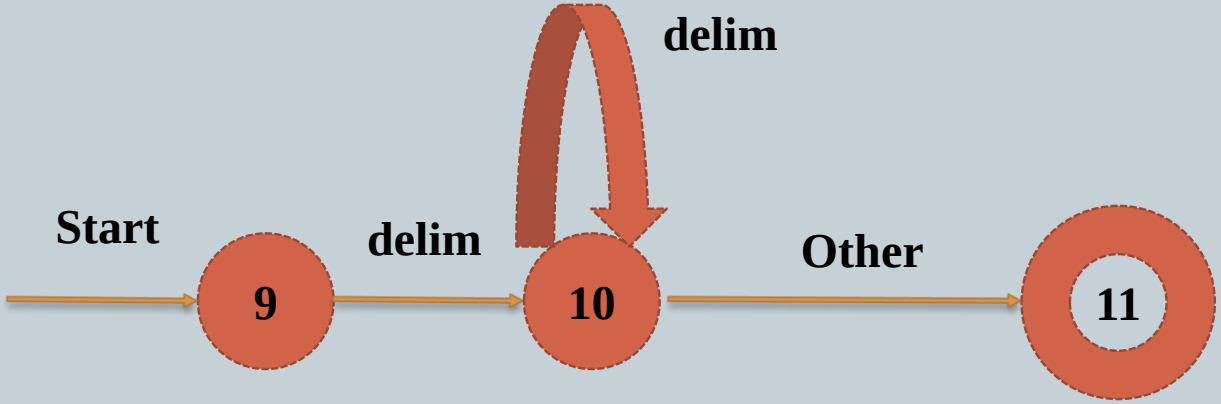


Lexical Analysis

- **Transition Diagram for unsigned number**



Transition Diagram for White Spaces



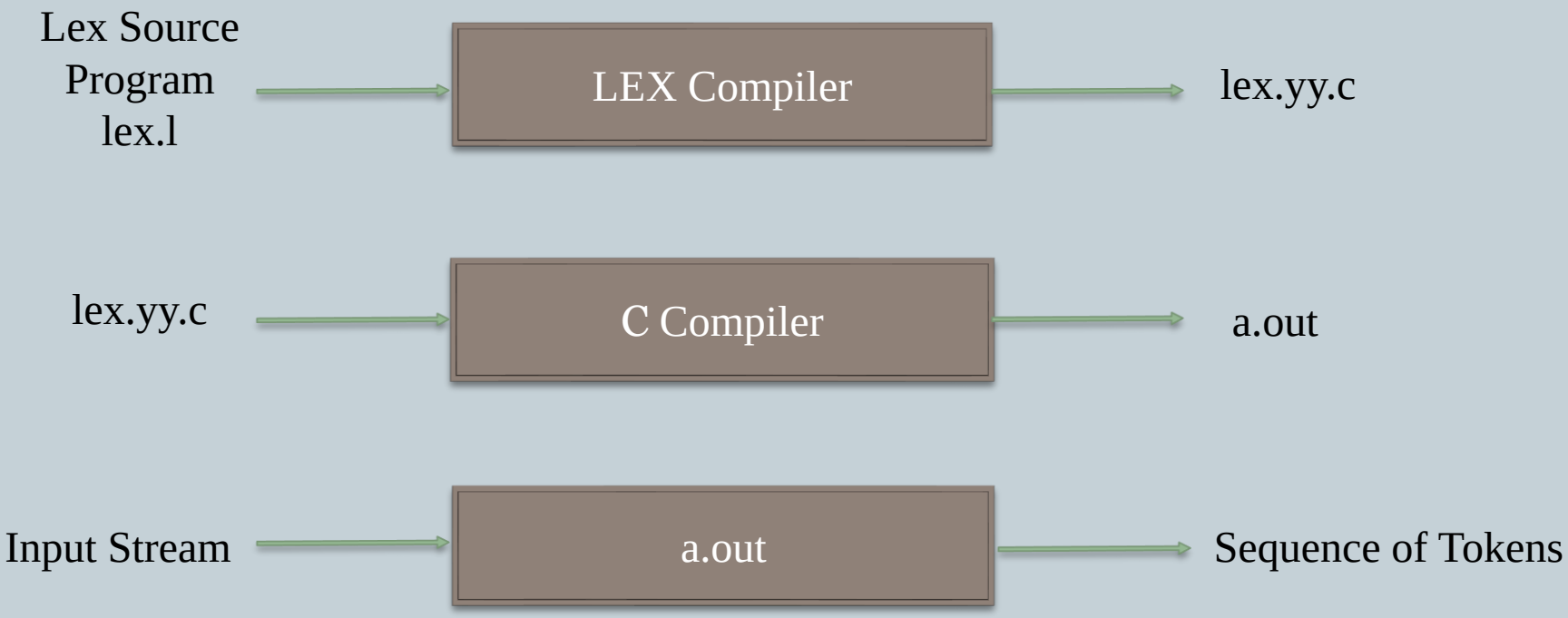
Lexical Analysis

The Lexical-Analyzer Generator Lex

- Allows to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.
- The tool – Lex compiler
- The input notation – Lex language
- The Lex compiler transforms the input patterns into a transition diagram and generates code – lex.yy.c

Lexical Analysis

- The Lexical-Analyzer Generator Lex



Creating a lexical analyzer with Lex

Lexical Analysis

- The structure of lex program

Lex program consists of three parts:-

%{

Declaration

%}

%%

Translation rules

%%

Auxiliary functions

Lexical Analysis

- The structure of lex program

%{

Declaration

%}

The declarations section includes

- declarations of variables,
- manifest constants (identifiers declared to stand for a constant)
- regular definitions

Lexical Analysis

- The structure of lex program

%%

Translation rules

%%

Translation Rules are of the form:-

P1 {action 1}

P2 {action 2}

....

Pi {action i}

Lexical Analysis

- The structure of lex program

Pattern { Action }

- where each Pi is a regular expression and
- each action i is a program fragment describing what action the lexical analyzer should take when pattern Pi matches a lexeme

- The structure of lex program

Auxiliary functions

- This section holds whatever auxiliary functions are used in the actions
- These functions can be compiled separately and loaded with the lexical analyzer

Lexical Analysis

The structure of lex program

- When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time
- It reads until it finds the longest prefix of the input that matches one of the patterns Pi.
- It then executes action Ai which returns the control to the parser
- But if it does not then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.

Lexical Analysis

The structure of lex program

- The lexical analyzer returns a single value i.e. the token name to the parser.
- To pass an attribute value (additional information about the lexeme), a shared integer variable is set known as **yylval**

Lexical Analysis

- The lex program for the tokens

```
%{
 /* Definitions of manifest constants
LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
```

Lexical Analysis

- The lex program for the tokens

```
/* Regular definitions */

delim [\t \n]

 ws {delim}+

 letter [A-Za-z]

 digit [0-9]

 id {letter}( {letter} | {digit} )*

 number {digit}+(\ . {digit} +) ? (E [+-] ? {digit}+) ?
```

Lexical Analysis

- The lex program for the tokens

```
%%
{ws}          { /* No action and No Return */ }
if            { return (IF); }
else          { return (ELSE); }
then          { return (THEN); }
{id}          { yylval = (int) installID(); return (ID); }
{number}      { yylval = (int) installNum(); return (NUMBER); }
```

Lexical Analysis

- The lex program for the tokens

```
"<"      { yylval = LT;  return(RELOP); }
"<="     { yylval = LE;  return(RELOP); }
 "="     {  yylval = EQ;  return(RELOP); }
"<>"     {  yylval = NE;  return(RELOP); }
">"      { yylval = GT;  return(RELOP); }
">="     { yylval = GE;   return(RELOP); }
%%
```

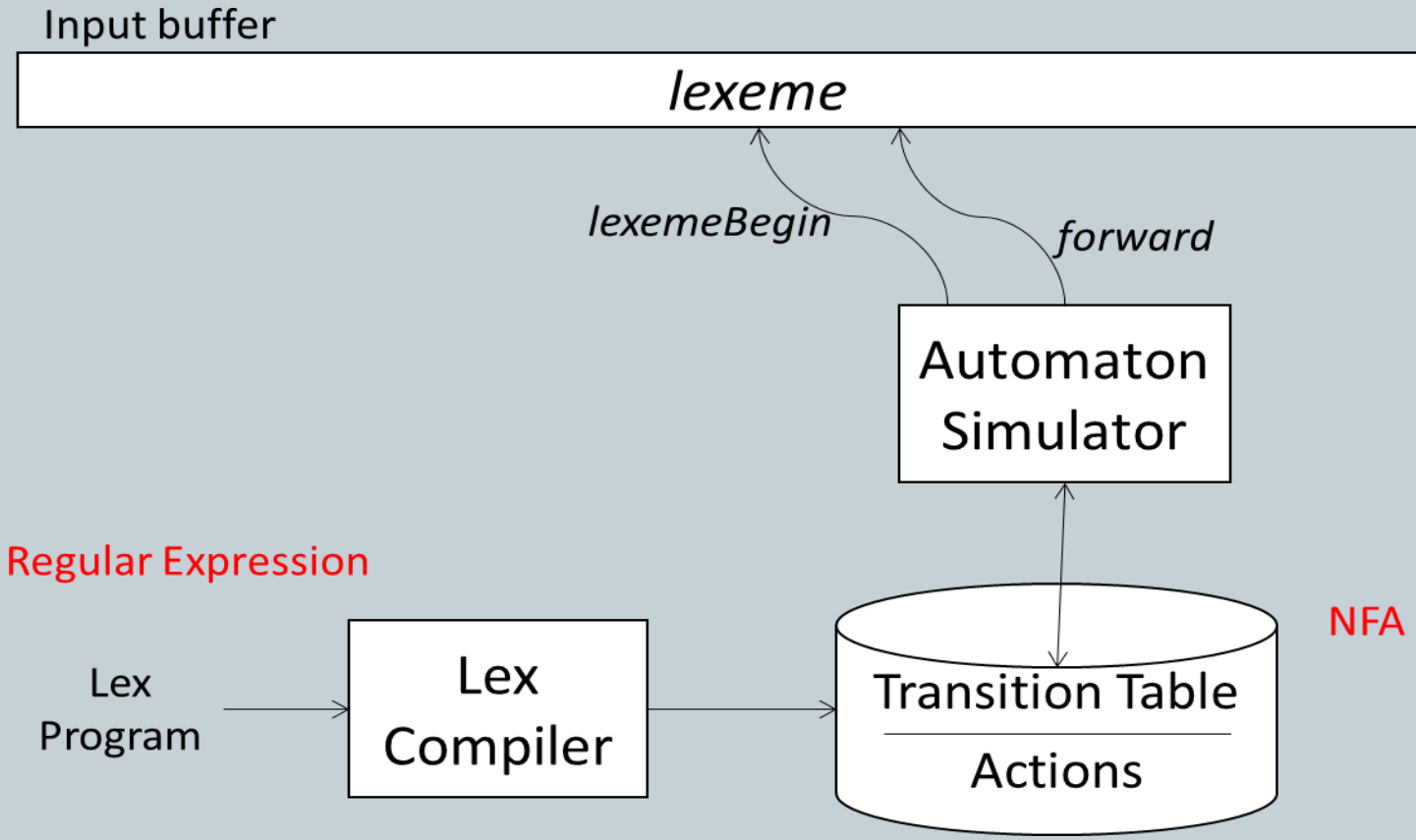
Lexical Analysis

- The lex program for the tokens

```
int installID() {
/* Function to install the lexeme whose first character is pointed to by yytext,
   and whose length is yyleng, into the symbol table and return a pointer to
   Symbol Table
*/ }
int installNum() {
/* Similar to installID but puts numerical constants into a separate table */ }
```

Lexical Analysis

- Design of a Lexical-Analyzer Generator



Lexical Analysis

Design of a Lexical-Analyzer Generator

A Lex program is turned into a transition table and actions which are used by a finite automaton simulator

The program that serves as the lexical analyzer includes a fixed program that simulates an automaton

The rest of the lexical analyzer consists of following component

1. A transition table of the automaton
2. Those functions that are passed directly through Lex to the output
3. The action from the input program which is to be invoked at the appropriate time by Automaton Simulator

Lexical Analysis

- Design of a Lexical-Analyzer Generator

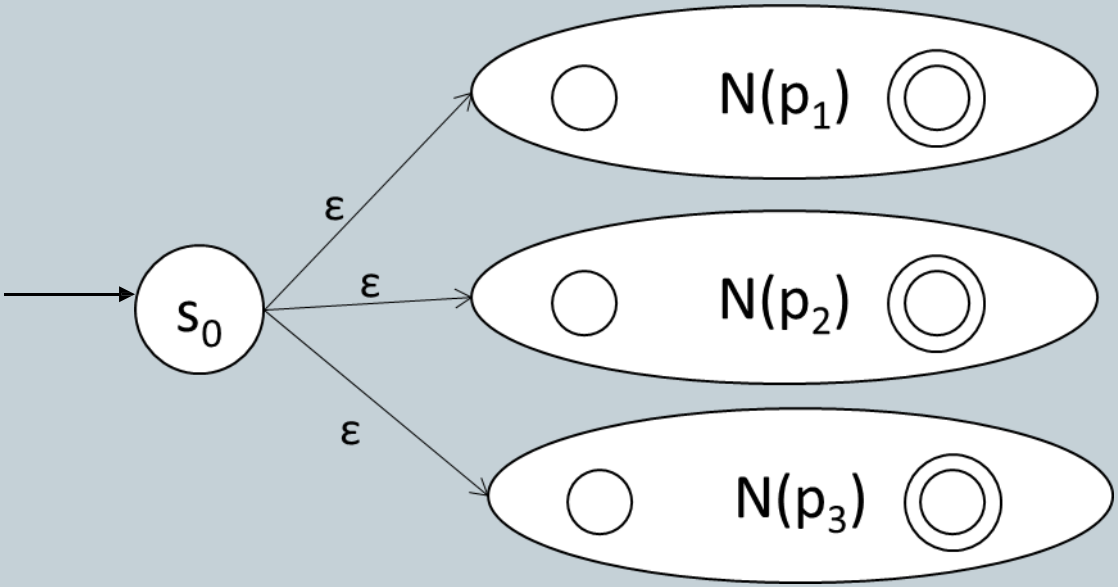
To construct the automaton, take each regular expression pattern in the Lex program and convert it using Algorithm to an NFA

We need a single automaton that will recognize lexemes matching any of the patterns in the program

Hence combine all the NFA s into one by introducing a new start state with ε -transitions to each of the start states of the NFA s Ni for pattern Pi

Lexical Analysis

- Design of a Lexical-Analyzer Generator



Lexical Analysis

- Design of a Lexical-Analyzer Generator

Example :

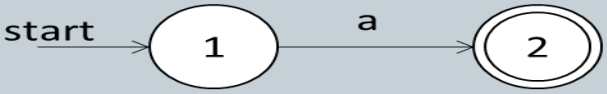
- To identify the following patterns
  - a {action A_1 for pattern p_1}
  - abb {action A_2 for pattern p_2}
  - a^*b^+ {action A_3 for pattern p_3}

Lexical Analysis

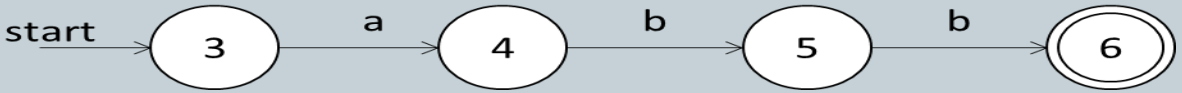
- Design of a Lexical-Analyzer Generator

Example :

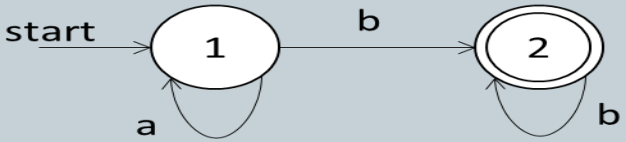
NFA for a



NFA for abb



NFA for a^*b^+

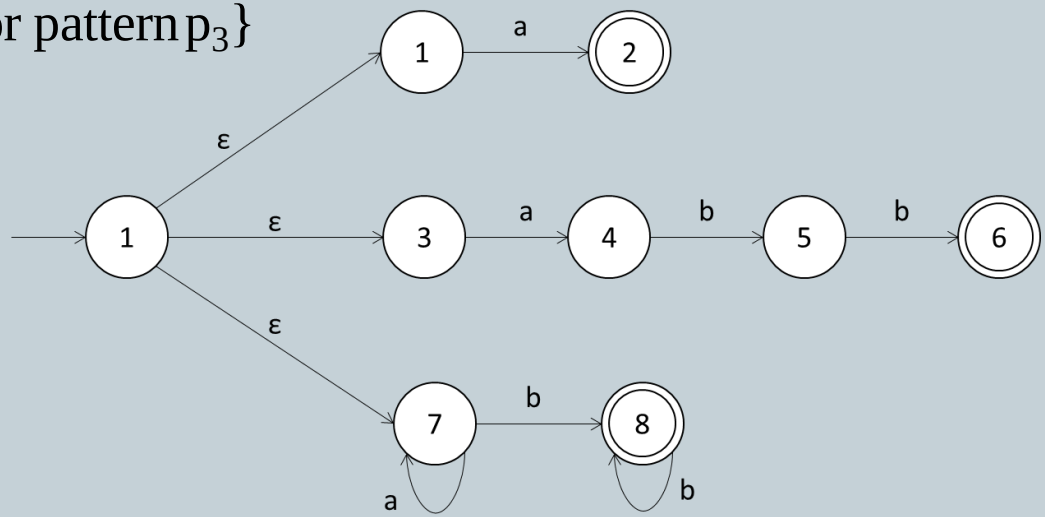


Design of a Lexical-Analyzer Generator

Example :

- To identify the following patterns
  - a {action A_1 for pattern p_1}
  - abb {action A_2 for pattern p_2}
  - a^*b^+ {action A_3 for pattern p_3}

Combined NFA :



Lexical Analysis

Data Structures used in Lexical Analyzer

- Symbol: Identifier used in the Source Program
- Examples: Names of variables, functions and Procedures
- Symbol Table: To maintain information about attributes of symbol
- Operations on Symbol Table:
  1. Add a symbol and its attributes
  2. Locate a symbol's entry
  3. Delete a symbol's entry
  4. Access a symbol's entry

Lexical Analysis

Data Structures used in Lexical Analyzer

Design Goal of Symbol Table

1. The Table's organization should facilitate efficient search
2. Table should be compact (Less Memory)

Time Space Trade off

To improve search efficiency, allocate more memory to symbol table

Organization of Entries:

1. Linear Data Structure
2. Non-Linear Data Structure

Lexical Analysis

- Data Structures used in Lexical Analyzer

Symbol Table Entry Format

- Number of Fields to accommodate attributes of one symbol
- Symbol Field: The symbol to be stored
- Key Field: Basis for the search in table
- Entry of Symbol is called record
- Each entry can be of type fixed length, variable length or hybrid

Lexical Analysis

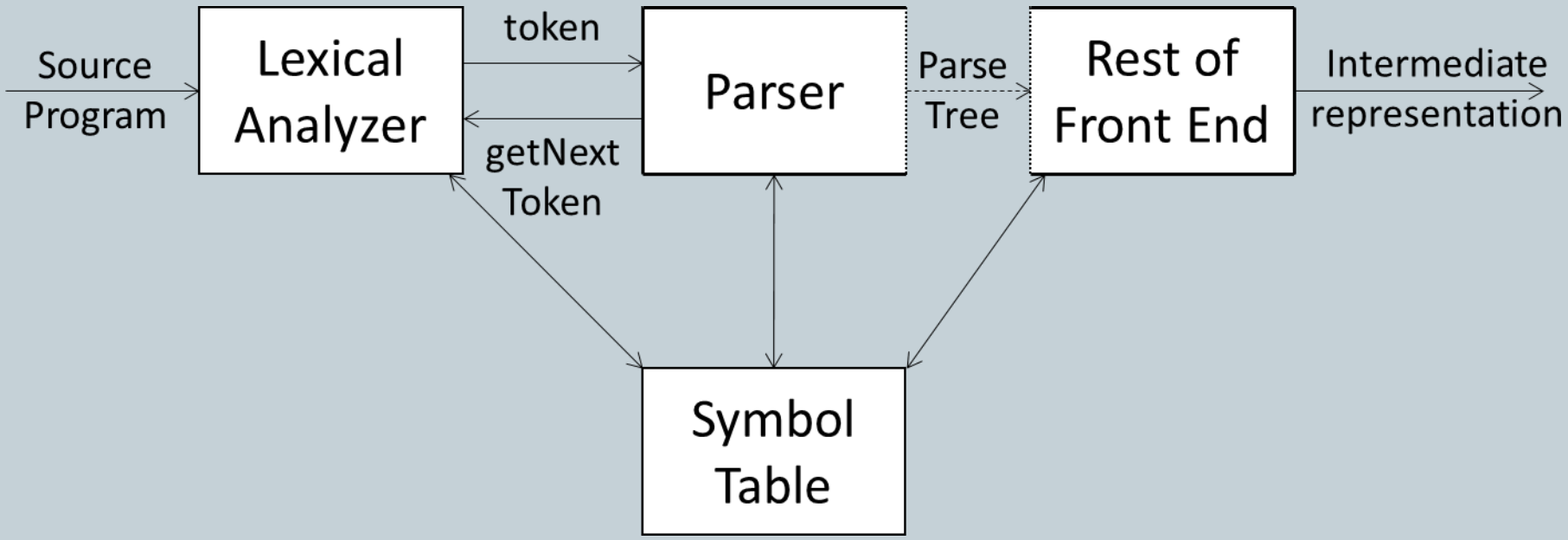
- Data Structures used in Lexical Analyzer

Symbol Class	Attributes
Variable	Type, Length, number and bounds of dimensions
Procedure	Number of parameters, address of parameter list
Function	Number of parameters, address of parameter list, type and length of returned value
Label	Statement Number

Syntax Analysis

- Syntax analysis is the second phase of the compiler
- It is also called as parsing and it generates parse tree
- Parser: It is the program that takes tokens and grammar (context-free grammar -CFG) as input and validates the input token against the grammar

Syntax Analysis



Position of Parser in compiler model

Syntax Analysis

- Syntax error handling

- **Lexical:** such as misspelling a keyword.
- **Syntactic:** such as an arithmetic expression with unbalanced parentheses.
- **Semantic**, such as an operator applied to an incompatible operand.
- **Logical:** such as an infinitely recursive call

Syntax Analysis

- Syntax error handling

Goals of Error handler in a parser (simple to state but challenging to realize) :

- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to detect subsequent errors
- Add minimal overhead to the processing of correct programs

Syntax Analysis

- Error handling strategies

- Panic Mode recovery
- Phrase Level recovery
- Error Productions
- Global Correction

- Error handling strategies

- Panic Mode recovery

- Panic mode error recovery is based on the idea of discarding input symbols one at a time until one of the designated set of synchronized tokens is found
- The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous.
- Advantage of simplicity and does not go into an infinite loop.

Error handling strategies

Phrase Level recovery

- On discovering error, a parser may perform a local correction on remaining input that allows the parser to continue.
- A typical local correction –
  - Replace comma by semicolon
  - Delete extraneous semicolon
  - Insert a missing semicolon
- Is major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection

Error handling strategies

Phrase Level recovery

- In phrase level recovery mode each empty entry in the parsing table is filled with a pointer to a specific error routine to take care of that error
- These error routine may be:

1. Change , insert, or delete input symbol
2. Issue an appropriate error message
3. Pop items from the stack

Error handling strategies

Error productions

- Error production adds rules to the grammar that describes the erroneous syntax .
- This strategy can resolve many , but not all potential errors
- It includes production for common errors and we can augment the grammar for the production rules that generates the erroneous constructs
- A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing
- Since it is almost impossible to know all the errors that can be made by the programmers , this method is not practical

Syntax Analysis

- Error handling strategies

- Global correction

- Replace incorrect input with correct input with as few changes as possible.
- This requires expensive techniques that are costly in terms of time and space.

The algorithm states that:

- For a given grammar G and an incorrect input string X
- Now find a parse tree for a related string Y with the help of an algorithm such that the number of insertions , deletion and changes of token required to transform X into Y are as small as possible
-

Syntax Analysis

➢ Context Free Grammars

- Syntax of a language is specified using a notation called – Context Free Grammar (CFG)
- A grammar naturally describes the hierarchical structure of most programming language constructs.
- Example –
  - statement → ***if*** (expression) statement ***else*** statement

specifies the structure of this form of conditional statement.

Syntax Analysis

Context Free Grammars

- **Terminals** – basic symbols from which strings are formed. Also called as “token name” or “token”.
- **Nonterminal** – syntactic variables that denote sets of strings. They help define the language generated by the grammar. They impose a hierarchical structure on the language.
- **Start symbol** – set of strings denoted by start symbol is the language generated by the grammar.
  productions for the start symbol are listed first

Syntax Analysis

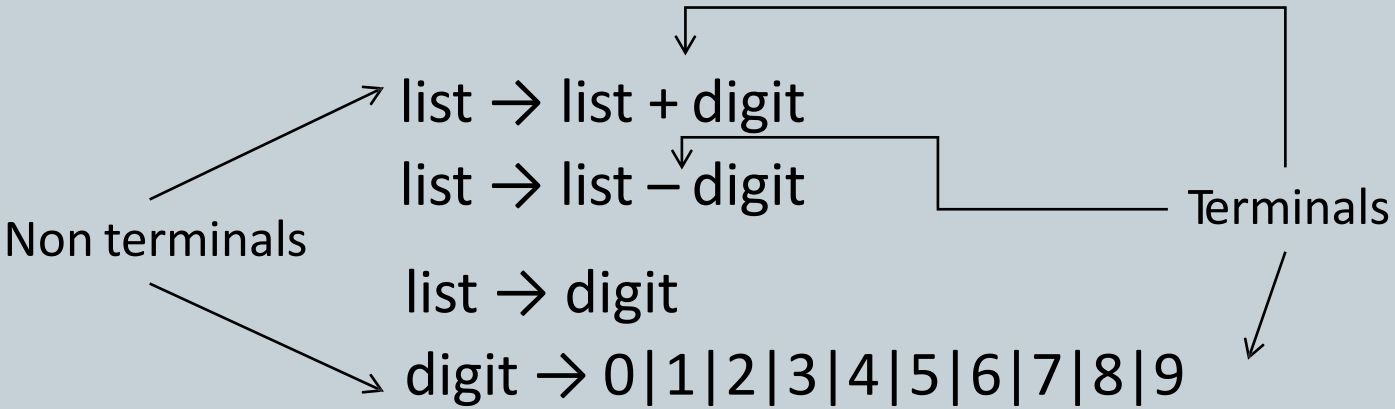
Context Free Grammars

- **Productions** – specify manner in which terminals and non-terminals can be combined to form strings.

- Each production consists of –
  - A Nonterminal : head of the production. Defines some strings denoted by the head
  - The symbol "→"
  - A Body or right side : consisting of 0 or more terminals and non-terminals. Describes one way in which strings of the nonterminal at the head can be constructed.

Syntax Analysis

- Context Free Grammars



- Above grammar can also be written as

list → list + digit | list – digit | digit

digit → 0|1|2|3|4|5|6|7|8|9

Syntax Analysis

Context Free Grammars

- Grammar for simple arithmetic expression

expr → expr + term

expr → expr – term

expr → term

term → term * factor

term → term / factor

term → factor

factor → (expr)

factor → id

Syntax Analysis

- Context Free Grammars

- Using notational conventions

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid id$$

- Notational conventions tell us that E, T and F are nonterminals, with the start symbol E. The remaining symbols are terminals.

Syntax Analysis

- Notational conventions

Syntax Analysis

- Derivations