MIT-WPU

Final Year(B.Tech)

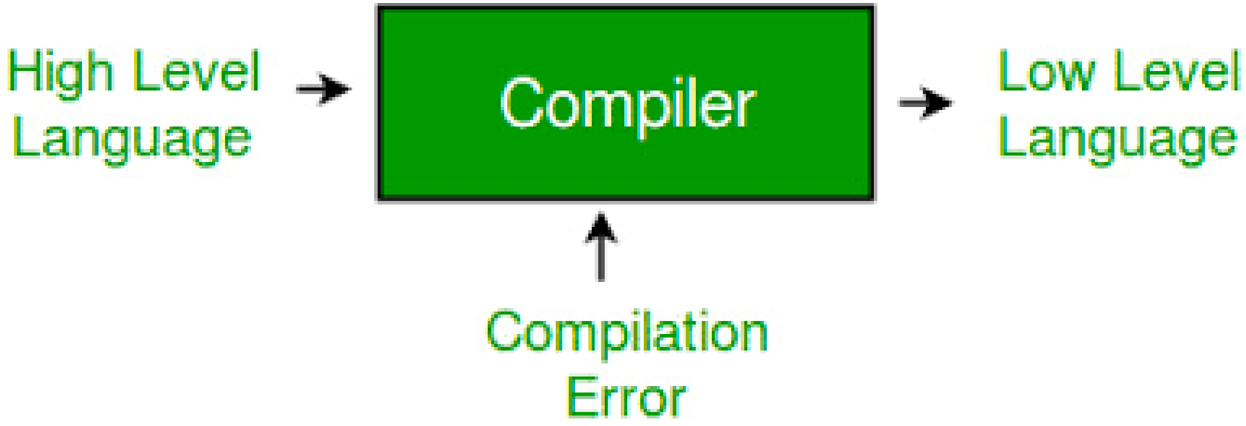
System Software and Compiler Design

Unit III

- **Introduction to compilers:** passes, phases, symbol table.
- **Lexical Analyzer:** Role of LEX Analyzer, Specification of tokens, Recognition of tokens, input buffering.
- **Syntax Analysis:** RDP, Predictive parser, SLR, LR (1), LALR parsers, using ambiguous grammar, Error detection and recovery.
- **LEX and YACC:** Specification and generation using LEX tool, Lexical errors. Automatic construction of parsers using YACC

Introduction to Compilers

- A compiler is a program that can read a program in one language – the *source* language – and translate it into an equivalent program in another language – the *target* language.



High Level Language

Compiler

Low Level Language

Compilation Error

Passes

- A pass is a complete traversal of the source program, or a complete traversal of some internal representation of the source program.
- Sometimes a single “pass” corresponds to several phases that are interleaved in time.
- What and how many passes a compiler does over the source program is an important design decision.

Passes

- In an implementation of compiler ,portions of one or more phases are combined into a module called a **pass.**
- A **pass** in compiler design is the group of several phases of compiler to perform analysis or synthesis of source program.
- Two types of pass:-
  1:-one pass
  2:-two pass
- In one pass structure:

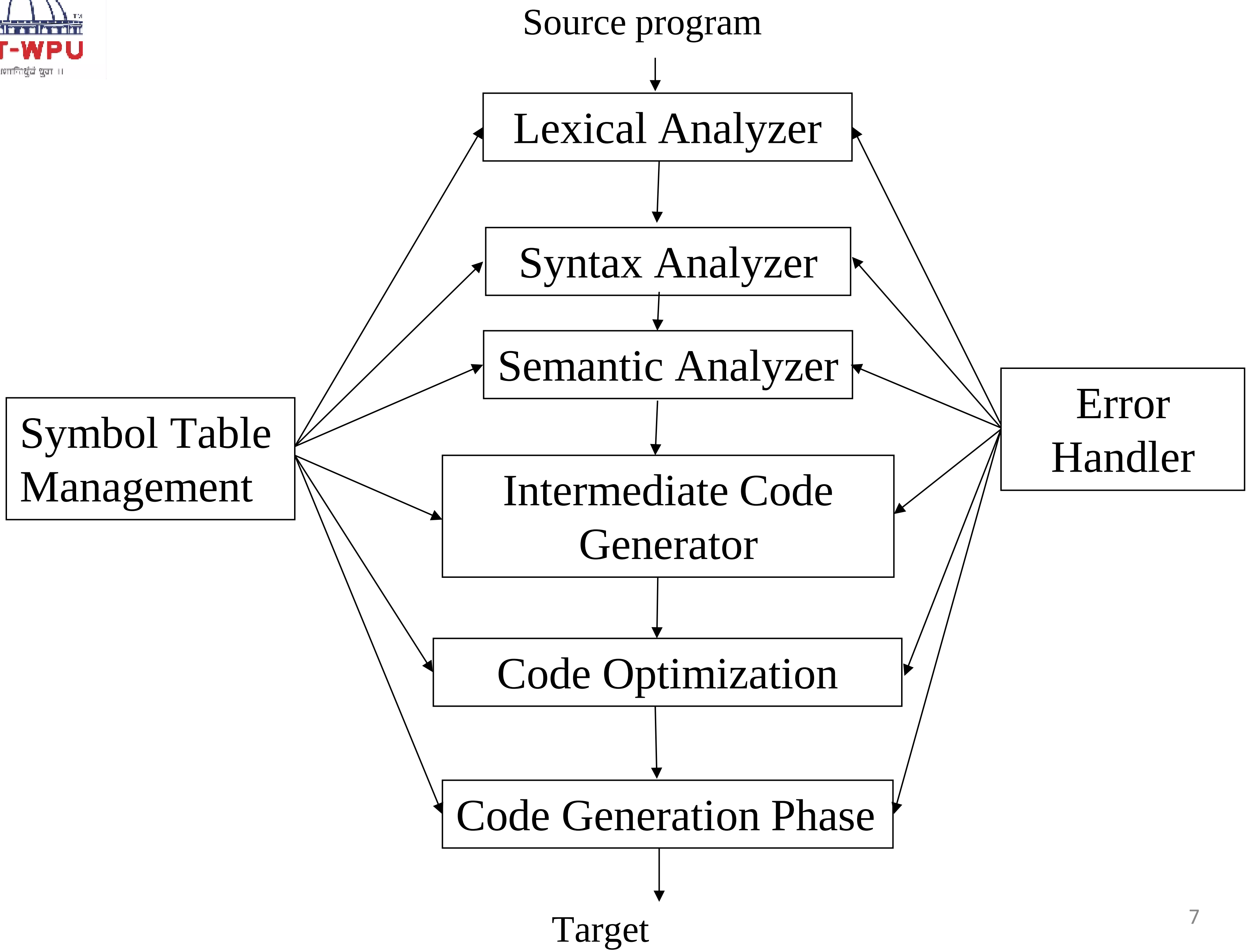
  both analysis and synthesis of source program is done in the flow from beginning to end of program.
- In two pass structure:

analysis of source program is done in first pass

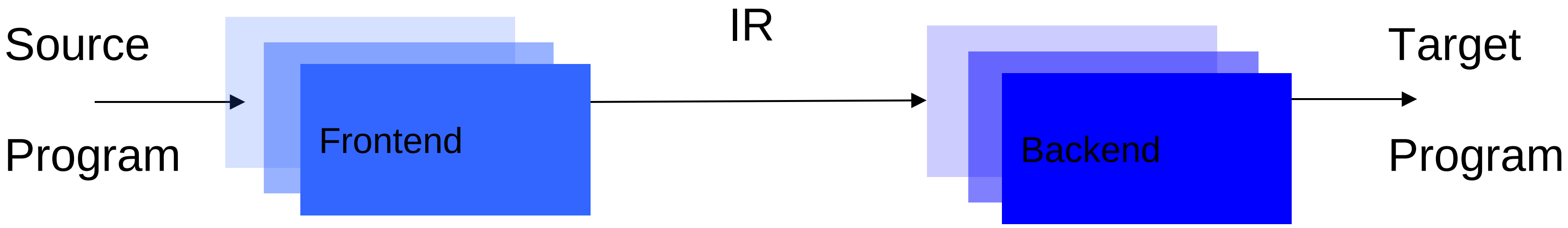
synthesis of source program is done in second pass

Phases of a Compiler

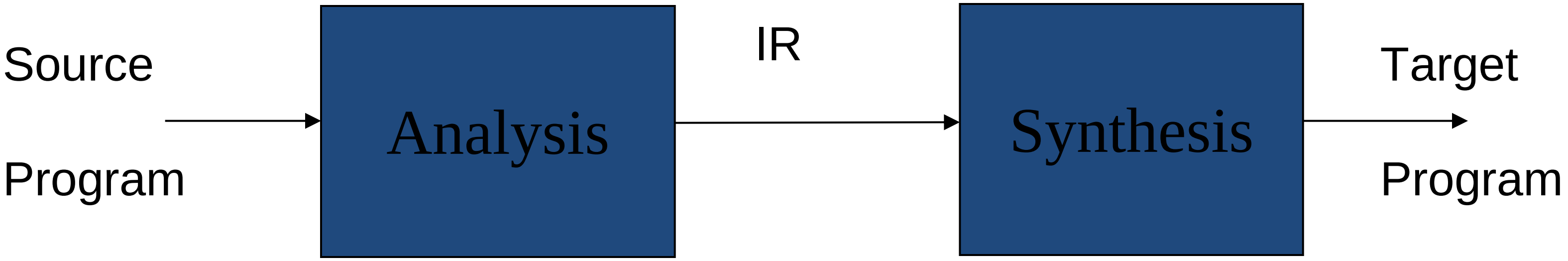
- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate (machine-independent) code generation
- code optimization
- Target (machine-dependent) code generation


Source program
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Code Optimization
Code Generation Phase
Target
Symbol Table Management
Error Handler

Front End and Back End Model of Compiler



- Analysis and Synthesis Phase of Compiler



Symbol table

- **Symbol table**
- It is an important data structure created and maintained by compilers.
- It is used by compiler to keep track of scope/binding information about names.
- These names are used in the source program to identify various program elements like variables names, function names, objects, classes, interfaces, etc.
- Symbol table is used by both the analysis and the synthesis parts of a compiler.

Symbol table

- A symbol table purposes are :

1. To store the names of all entities in a structured form at one place.
2. To verify if a variable has been declared.
3. To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
4. To determine the scope of a name (scope resolution).
5. A symbol table is simply a table which can be either linear or a hash table.

- It maintains an entry for each name in the following format:

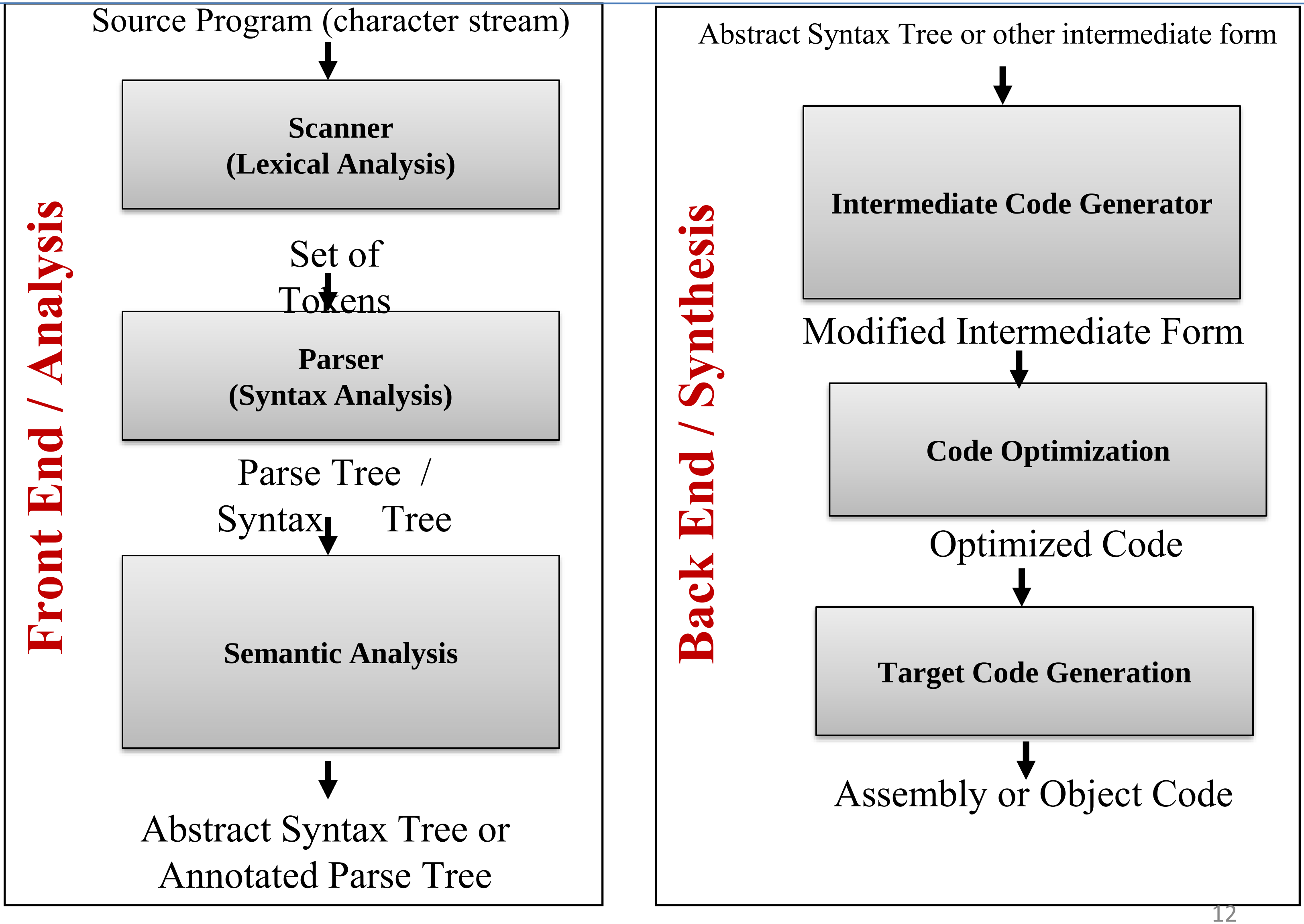
  <symbol name,type,attribute>

Symbol tables

- Data structures used for symbol table:

1. List
2. Linked list
3. Binary trees
4. Hash tables

Compiler Front End – Back End / Analysis –Synthesis Phase



Front End / Analysis

Source Program (character stream)

Scanner
(Lexical Analysis)

Set of Tokens

Parser
(Syntax Analysis)

Parse Tree / Syntax Tree

Semantic Analysis

Abstract Syntax Tree or Annotated Parse Tree

Back End / Synthesis

Abstract Syntax Tree or other intermediate form

Intermediate Code Generator

Modified Intermediate Form

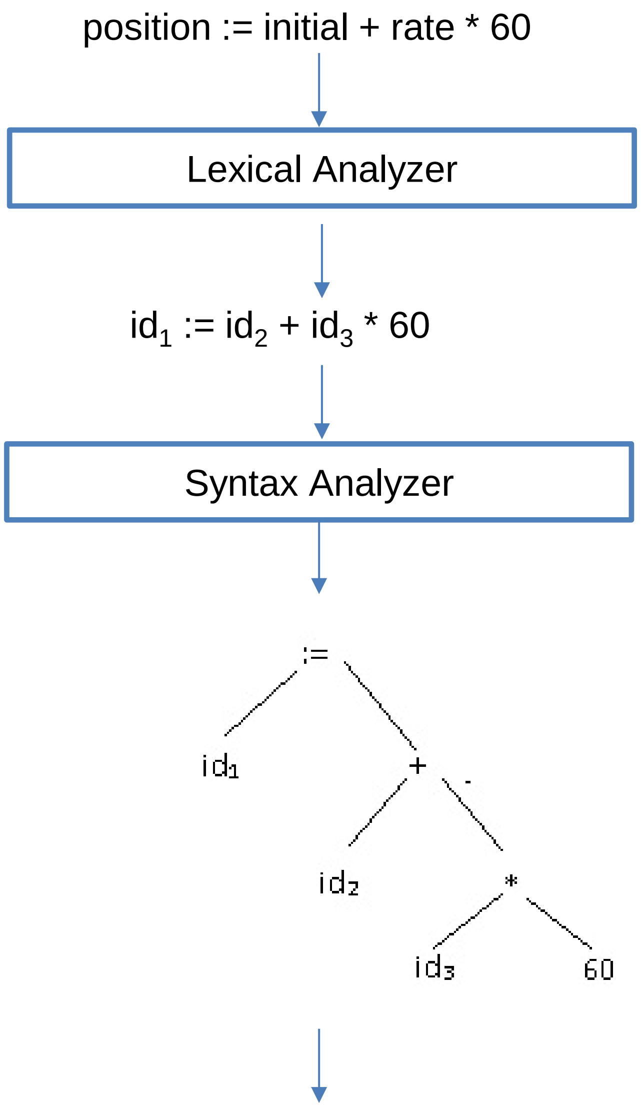
Code Optimization

Optimized Code

Target Code Generation

Assembly or Object Code

Assignment Statement Translation



position := initial + rate * 60

Lexical Analyzer

$id_1 := id_2 + id_3 * 60$

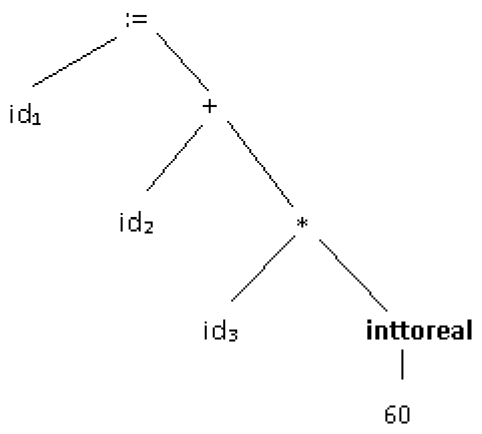
Syntax Analyzer

Symbol Table

1	position	…
2	initial	…
3	rate	…
4		

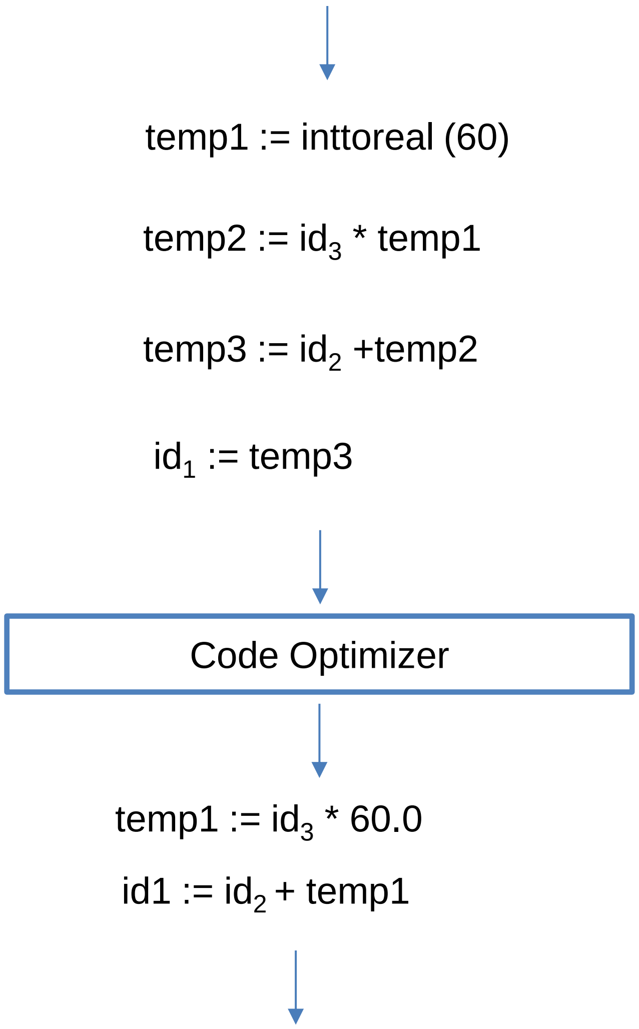
Assignment Statement Translation

Semantic Analyzer



Intermediate Code Generator

Assignment Statement Translation



temp1 := inttoreal (60)

temp2 := id_3 * temp1

temp3 := id_2 +temp2

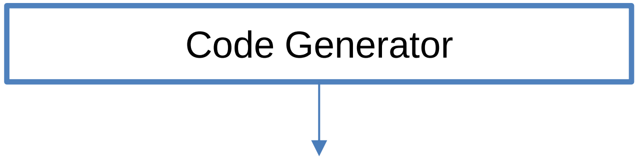
id_1 := temp3

Code Optimizer

temp1 := id_3 * 60.0

id1 := id_2 + temp1

Assignment Statement Translation



Code Generator

MOVF id_3, R2

MULF #60.0, R2

MOVF id_2, R1

ADDF R2, R1

MOVF R1, id_1

Regular Expression

- Rules

1. Є is a RE that denotes {Є }.

Regular Expression

- Rules

1. Є is a RE that denotes {Є }.
2. If ‘a’ is a symbol in ∑ RE is {a}

Regular Expression

- Rules

1. Є is a RE that denotes {Є }.
2. If ‘a’ is a symbol in ∑ RE is {a}
3. Suppose r & s are RE s denoting the languages L(r) & L(s) then

   a.(r) | (s) is a RE denoting L(r) U L(s)

   b. (r)(s) is a RE denoting L(r) . L(s)

   c. (r) * is a RE denoting (L(r)) *

   d. . (r) is a RE denoting L(r)

Contd…

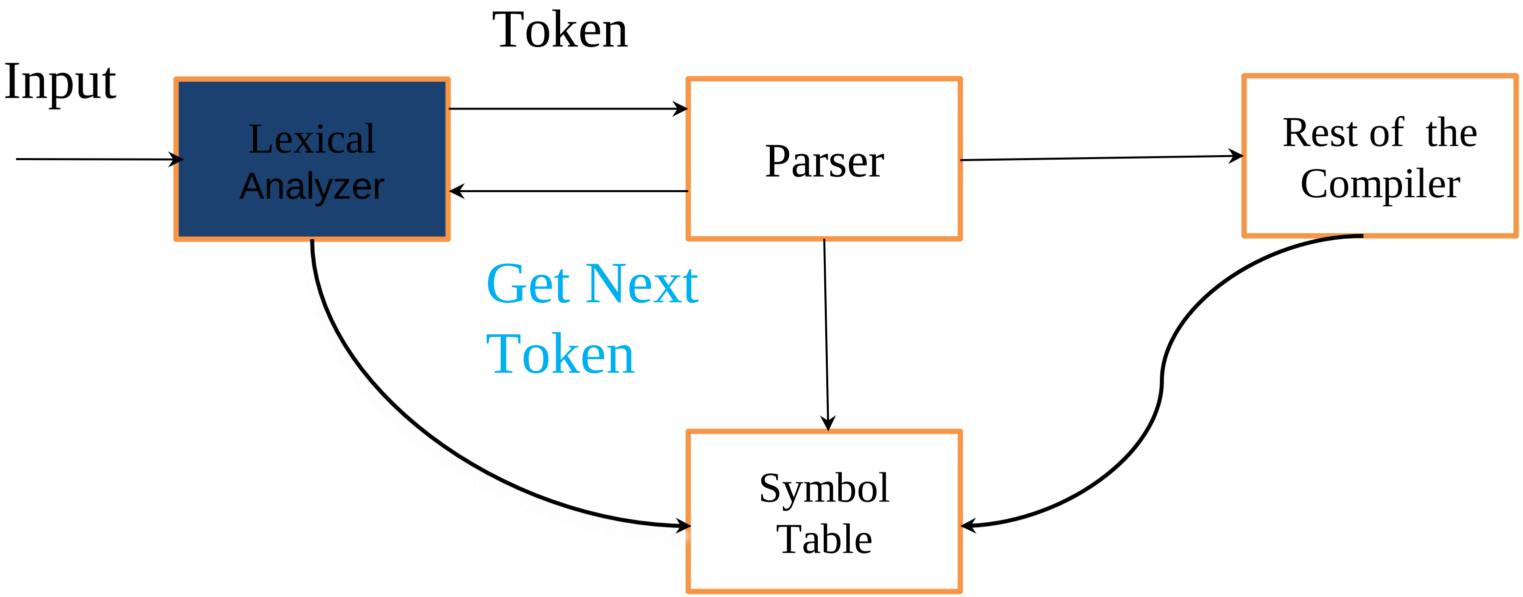
Axioms

- r|s = s|r
- r| (s|t) = (r|s)|t
- (rs)t = r(st)
- r|s|t = rs|rt
- (s|t)r = sr|tr
- Є r = r
- r Є = r
- $r^{*} = (r|\text{Є})^{*}$
- $r^{**} = r^{*}$

Description

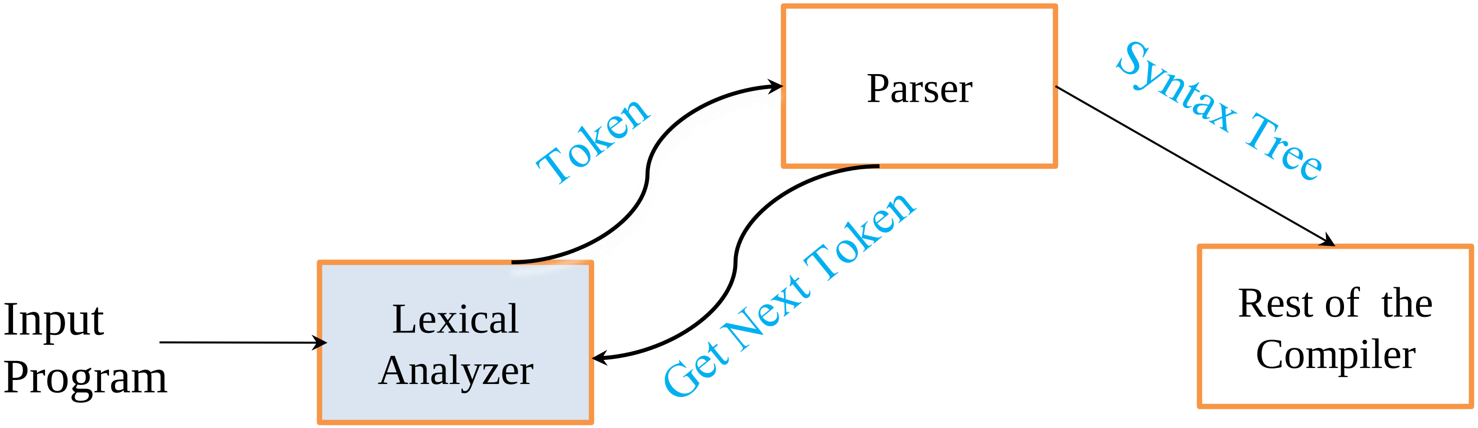
- | is commutative
- | is associaltive
- Concat is associative
- Concat is distributes over
- same as above
- Є is the identity element For concatenation
- Relation between * and Є
- * is idempotent

Lexical Analyzer



Lexical Analyzer (cont…)

- Where does the Lexical Analyzer fits into the rest of Compiler?
  - The front end of most compilers is Parser Driven.
  - When the parser needs the next Token, it involves the Lexical Analyzer.
  - Instead of analyzing the entire input string, the lexical analyzer sees enough of the input string to return a single Token.



Lexical Analyzer acts as a **Sub-routine.**

Contd…

- Issues in lexical analysis

1. Simple design
2. Compiler efficiency is improved
3. Compiler portability is enhanced

Lexical Analyzer (cont…)

- **Terms used in Lexical Analyzer:**

  - LEXEME-Smallest Logical Unit (Word) of Program.

    e.g.{ I, sum, buffer, for,10,+ …}

  - TOKEN –Set of Similar Lexemes.

    e.g.

    Identifier - { I, sum, buffer …}

    Keyword – { for, ….}

    Number – { 0, 23, ….}

  - PATTERN- as good as Regular Expression

    e.g. DIGIT [0-9]

Example

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	3.14,0,6.02	Any numerical constant
literal	“core dumped”	Any characters bet “ and “ except “

Lexical Analyzer (cont…)

- **Lexemes not passed to the parser:**

  - White Spaces (WS) – Tab, Blanks, New Lines
  - Comments

These too have to be detected and Ignored.

Lexical Analyzer (cont…)

- **Tasks of a Lexical Analyzer:**

1. Scans the input program,identifies valid words of the language.
2. Removes extra white spaces,blanks,tabs,new lines ,comments etc
3. Expands user defined macros

   (done at the compile time by lexical analyser)

   e.g. #define Max 5

   #include<stdio.h>

4. Report presence of foreign words
5. May perform case conversion
6. It generates tokens and pass to syntax analysis phase.
7. Lexical Analyzer is implemented as Finite automata.

Lexical Analyzer (cont…)

- **Basic Tasks of a Lexical Analyzer:**
  - Recognizing Basic Elements.
  - Removal of White Spaces and Comments.
  - Recognizing Constants and Literals.
  - Recognizing Keywords and Identifiers.

< token, token value>

Pointer to Symbol Table Entry

Ex: < id, .>

< no, 9>

Lexical Analyzer (cont…)

- **Token:**
- Token stream: Each significant lexical chunk of the program is represented by a token
  - Operators & Punctuation: {}[]!+-=*;: …
  - Keywords: if while return goto
  - Identifiers: id & actual name
  - Constants: kind & value; int, floating-point character, string, …

Lexical Analyzer (cont…)

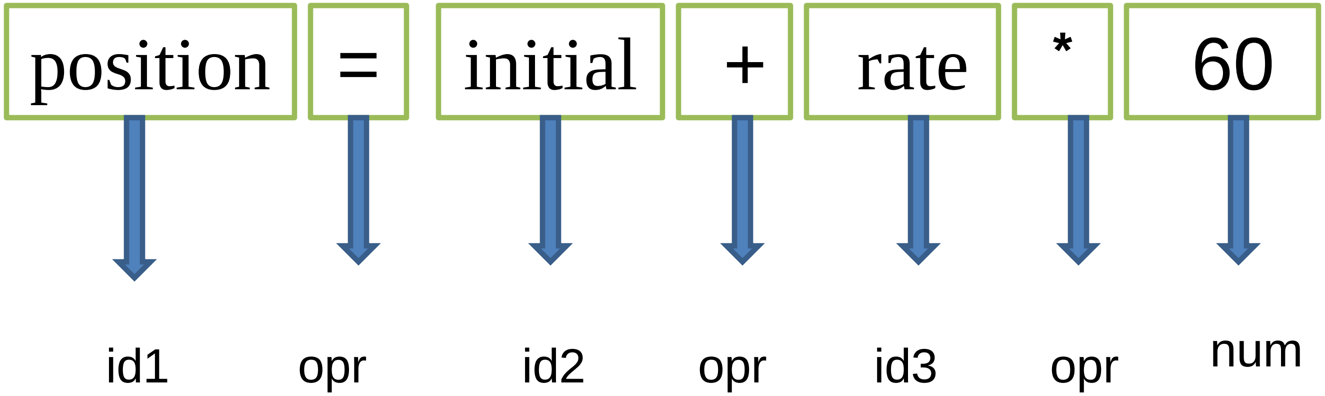
Example:

position = initial + rate * 60

Tokenized to:

position	:	The identifier
=	:	The Assignment Operator
initial	:	The identifier
+	:	The Plus Operator
rate	:	The identifier
*	:	The Multiplication Operator
60	:	The Number/ Constant

Lexical Analyzer (cont…)



position	=	initial	+	rate	*	60
id1	opr	id2	opr	id3	opr	num

id1 = id2 + id3 * 60

Design of Lexical Analyzer

- Every action is implemented by Transition Diagram

- TG for Identifiers, Keywords, Operators…
- Regular Expression.
- Finite Automata.

Two Approaches

1.Hand Code : This is only of historical interest now.
(possibly more efficient)

2.Use Generator : To generate the lexical analyzer from a format description.

- The generation process is faster.

- Less prone to Errors.

Contd…

- Lexical analyzer generator consists of two parts:

1. Specification of tokens – done through RE
2. Specification of actions

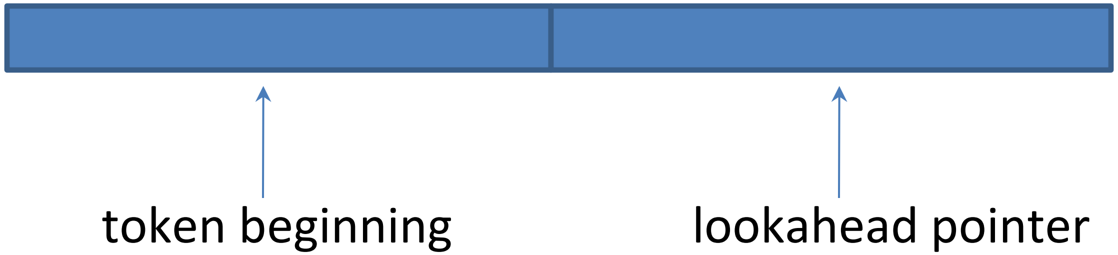
The lexical analyzer generator

- Processes RE s & forms a graph DFA
- Copies the action routines without any change
- Adds a driver routine

these 3 things put together constitutes the lexical analyzer.

Input buffering

- The lexical analyzer scans the characters of the source program one at a time to discover tokens.
- many characters beyond the next token may have to be examined before the next token itself can be determined.
- For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer
- Figure shows a buffer divided into two halves of, say 100 characters each.



Contd…

- E.g. DECLARE(arg1,arg2,…,argn)

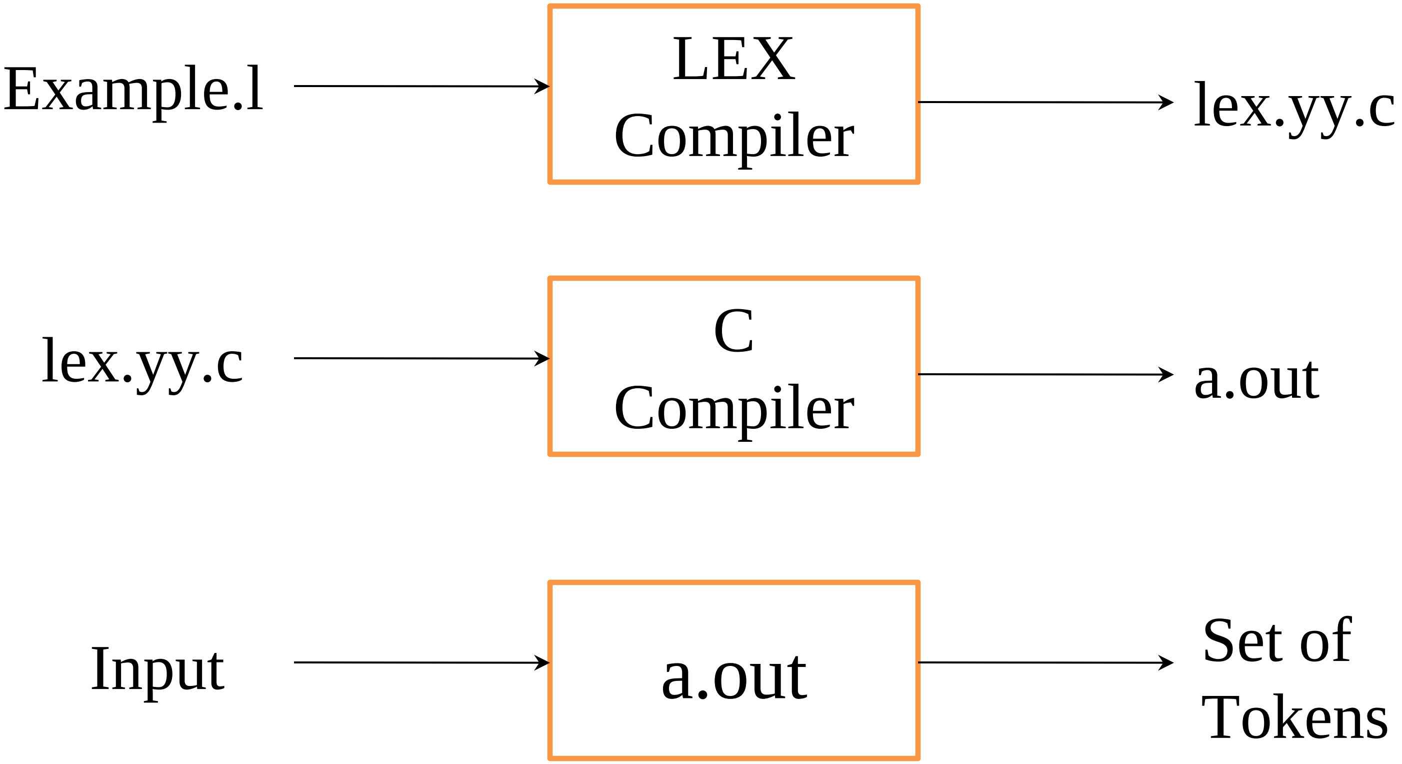
  Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis.

  If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file.

Lexical Errors

- Matched but ambiguous:
  - left to the other phases(e.g., parser)
  - e.g., fi (a == f(x)) … : fi => identifier ?? misspelling of “if”
- d=2r ,no symbol can start with 2(digit)
- Unmatched:
  - Panic mode recovery: delete successive characters from the remaining input until a well-formed token is found
  - Repair input (single error):
    - deleting an extraneous character
    - inserting a missing character
    - replacing with a correct character
    - transposing two adjacent character

LEX



Example.l → LEX Compiler → lex.yy.c

lex.yy.c → C Compiler → a.out

Input → a.out → Set of Tokens

LEX Specification

- Declaration Section : Variable, Manifest Constant, Regular Definition.

%{

%}

- Translation Rules Section

P1 ………. { action1}

P2 ………. { action2}

P3 ………. { action3}

P4 ………. { action4}

%%

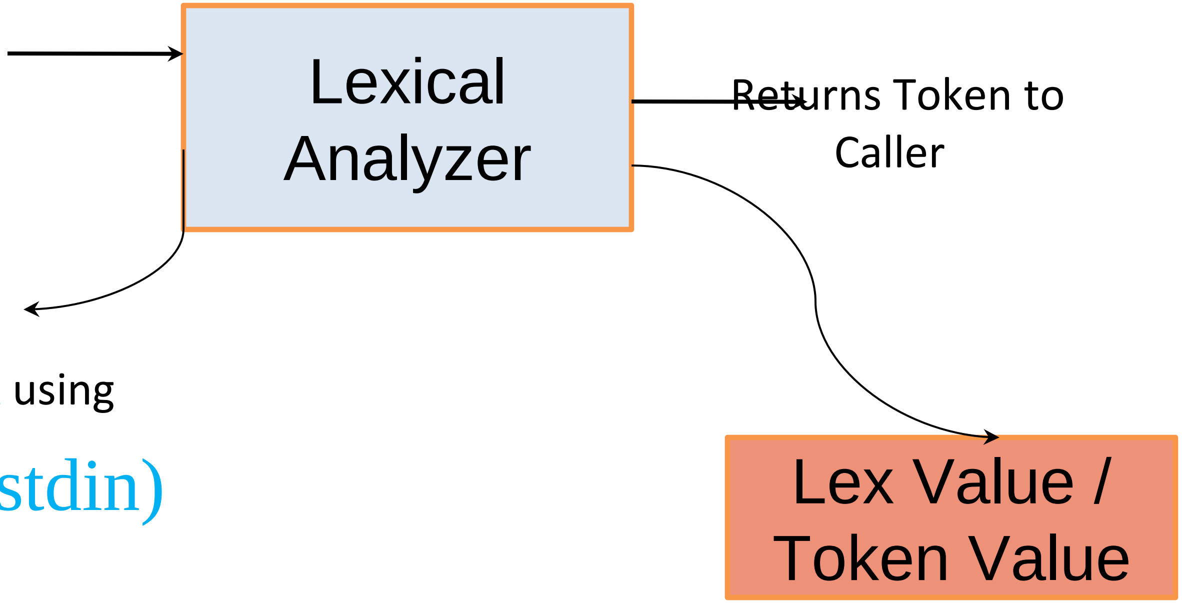
%%

COMPULSARY SECTION

- Subroutine Section/Auxiliary Procedures

LEX (cont_...)

Implementing the Interactions



Uses getchar() In

C to read a Character

Lexical Analyzer

Returns Token to Caller

Pushes back ch using

ungetc(ch,stdin)

Lex Value / Token Value

Sets a global variable to attribute value

(yylval)

LEX (cont_...)

- Commands Used:

```
 # lex filename.l
# cc lex.yy.c –ll
# ./a.out
```

Sample program

```
%{
%{
     #include<stdio.h>
     int wcnt=0, lcnt=0,
char_cnt=0;
%}
   charac       [^\n\t]
   eol         \n
  word           “   "
%%
    {eol}       {lcnt++; wcnt++;}
    {word}      {wcnt++;}
   {charac}   {char_cnt++;}
%%
```

Definition section

Regular definition section

Declaration section

Rules section

Contd....

```
main()
main()
{
        yyin=fopen("sample.txt","r");

          yylex();
    printf("\n\nNumber of lines: %d",lcnt);
    printf("\nNumber of words: %d",wcnt);
    printf("\nNo. ofcharacters:%d",char_cnt);
}
int yywrap()
{
return 1;
}
```

Auxiliary procedure section

Contd…

yylex()

- Entry point
- Call yylex() to start or resume scanning
- If a lex action does a return to pass a value to the calling program, the next call to yylex() will continue from the point where it left off
- All code in the rules section is copied into yylex()

yywrap()

- When EOF is found it calls routine yywrap() to find out what to do next.
- Returns 0 –scanner continues scanning
- Returns 1 – the scanner returns zero token to report the EOF

Contd…

yytext

- Whenever a lexer matches a token the text of the token is stored in the null terminated string yytext
- When flex finds a match, yytext points to the first character of the match in the input buffer
- The value of yytext will be overwritten the next time yylex() is called.
- The value of yytext is only valid from within the matched rule’s action

Regular Expression

.	Matches any single character except new line character
*	Matches 0 or more copies of the preceding expression
[]	char class which matches any char within the bracket
^	Matches the beginning of the line as 1st char of RE
$	Matches the end of line as the last char of a RE
{}	Indicates how many times the previous pattern is allowed to match when containing one or two nos.
\	Used to escape metacharacters & as part of the usual c escape sequences e.g. “\n”
	Matches one or more occurrence of the preceding RE
+	matches zero or one occurrence of the preceding RE
?	e.g. -?[0-9]+

Contd…

| Matches either the preceding RE or the following RE
e.g. is|am|are

"…" Interprets everything within the quotation marks literally

/ Matches the preceding RE but only if the followed by the following RE

() Groups series of RE together into a new RE

Lex Program

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

/* regular definitions */
delim                [ \t\n]
ws                    {delim}+
letter               [A-Za-z]
digit                 [0-9]
Id                     {letter} ({letter}|{digit})*
number               {digit}+(\. {digit}+)?(E+-]?{digit}+)?
```

Lex Program (cont…)

```
%%
{ws}          {/* no action and no return */}
If            {return (IF);}
then          {return (THEN);}
else          {return (ELSE);}
{id}          {yylval = (int) installID(); return (ID);}
{number}      {yylval = (int) installNum(); return (NUMBER);}
“<“           {yyval = LT; return (RELOP);}
“<=“          {yyval = LE; return (RELOP);}
“=“           {yyval = EQ; return (RELOP);}
“<>“          {yyval = NE; return (RELOP);}
“>“           {yyval = GT; return (RELOP);}
“>=“          {yyval = GE; return (RELOP);}
%%
```

Lex Program (cont…)

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 thereto */

int installNum {/* similar to installID, but puts numerical constants into a separate table */

Scanner: Lexical Analysis

– What kind of **ERRORS** can be reported by **LA**?

➢ Issues an Appropriate Error Message

➢ Errors:

1. The Entire Lexeme is read and then truncated to the Specified Length.
2. Error of the Second Type-

   a. Skip Illegal Character.

   b. Pass the Character to the parser which has better knowledge of the context in which Error has occurred.
3. Wait till end of File and issue Error Message.

Like Misspelling of Keywords.