Compiler Design - 10CS63

UNIT 1: Introduction

Translator - Any program that converts a high level language program to Machine (Low Language) code.

Compiler - Program that reads code in one language i,e source code and translates it into another language i,e target language is a compiler.

Translator

Source prog → [Compiler] → Target prog

Interpretter - A kind of language processor which does not produce target program as a translation, but directly execute the operations specified in source program, on inputs supplied by the user

Source prog →
Input → [Interpretter] → output

Language Pre-processing system:

Source prog
↓
[Preprocessor]
↓ modified src prog
[Compiler]
↓ target assembly prog
[Assembler]
↓ relocatable M/c code
[Linker/Loader] ← library files, relocatable object files
↓ target M/c code

- A Hybrid compiler :

Ex: Java lang processor (Just-in-Time)

src prog
↓
Translator
↓ (bytecodes)
Intermediate prog → Virtual Machine → output
input → Virtual Machine

Structure of a compiler :

↓ character stream
Lexical Analyzer
↓ token stream
Syntax Analyzer
↓ syntax tree
Semantic Analyzer
↓ syntax tree
Intermediate code Generator
↓ intermediate representation
Machine Independent code optimizer
↓ intermediate representation
Code Generator
↓ target machine code
Machine Dependent code optimizer
↓ target machine code

Front End Pass

Back End Pass

Symbol table

Error Table

→ 2 main parts:

① Analysis – breaks up source prog into constituent pieces & imposes grammatical structure on them. Based on the structure it creates intermediate representation of source prog. Collects information about prog, stores it in the "Symbol Table". (Front End of compiler)

② Synthesis – constructs the desired target prog from the intermediate representation and information in the symbol table. (Back End of compiler)

→ 7 phases:

① Lexical Analysis – Scanning

- On reading character stream of src prog, it groups them into meaningful sequences called "Lexemes".
- for each lexeme, Analyser produces as output a token of form:

< token-name, attribute value >

abstract symbol used in parser ← token-name

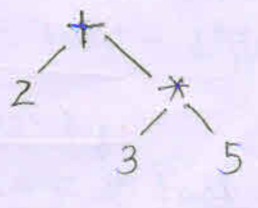
attribute value → points to entry in the symbol table for this token

② Syntax Analysis – Parsing

- Parser uses tokens i.e output of scanner and creates a tree like intermediate representation that depicts the "Grammatical Structure" of token stream

Ex: For Grammar $E \rightarrow E+E \mid E*E \mid num$

For Input $2+3*5$



interior node: operators
exterior node: arguments

(3) Semantic Analysis

- uses syntax tree and information in symbol table to check source prog for semantic (meaning) consistency with lang definition.
- It gathers type information and saves it in either syntax tree or symbol table for use in ICG.
- Type checking - compiler checks whether each operator has the matching operands
- coercions - Lang specification may permit some type conversion

(4) Intermediate Code Generation (ICG)

- The intermediary code during processing may be in the form of syntax tree or reduced form of source code.
- properties :

→ should be easy to produce
→ should be easy to translate into target M/C.

(5) Code optimization (M/C independent)

- to improve intermediate code to get better target code

→ Better in terms of : faster, shorter, less power consuming code

- Instead of using int to float operation, replace integer by its floating-point value directly

(6) code Generation

- input from intermediate representation maps to target lang.

- If target lang is M/C code - the instructions are translated into sequences of M/C instruction to perform same task.
- Judicious assignment of registers to hold variables is done.

→ Compiler construction Tools :

① Parser Generators - automatically produce syntax analyzers from a grammatical description of a prog lang.

② Scanner Generators - produce lexical analyzers from a regular expression description of tokens of lang.

③ Syntax directed translation engines - produce collections of routines for walking a parse tree and generate ICG

④ code generator generators - produce CG from collection of rules for translating each operation of Intermediate lang into M/C lang for a target M/C

⑤ Data flow analysis engines - facilitate gathering of data about how values are transmitted from one part of prog to every other part.

⑥ compiler construction toolkits - provide integrated set of routines for constructing various compiler phases.

Application of compiler Technology :

① Implementation of high level prog. lang using modern OOPS concept like,

→ Data Abstraction

→ Inheritance properties

② optimization for computer architectures

→ Parallelism

(i) at instruction level - multiple operations executed together

(ii) at preprocessor level - different threads run seperately

→ Memory Hierarchy

Building very <u>Large</u> or <u>Fast</u> storage, but not both

③ Design New computer Architectures

→ RISC - reduces complex memory addressing, support data structure access, procedure invocation ...

→ Specialized Architectures -

Data flow M/C, vector M/C, VLIW & SIMD M/C.

④ Program Translations

(i) Binary Translation - Increases S/W availability

(ii) Hardware Synthesis - Verilog, VHDL - reduces time & effort

(iii) Database Query Interpretter - SQL queries effective retrieval

(iv) compiled simulation - model run, to validate design.

(v) Reduce redundancy in code

⑤ Software Productivity Tools

(i) Type checking - to catch program inconsistency

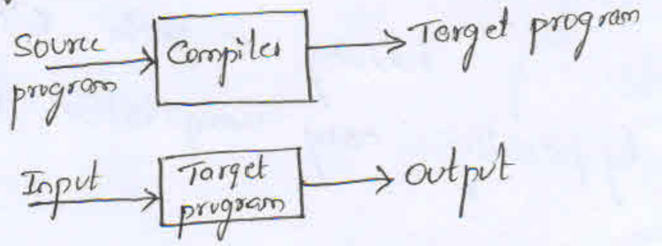
(ii) Bounds checking - Lang. provides range checking like for the buffer overflow, security, optimize range check, sophisticated analyses, error detection tools.

(iii) Memory Management Tools - (Garbage collection)

- Automatic memory management tracks all memory related errors - leaks...

1. Write the difference between compiler and Interpreter

Compiler	Interpreter
1. Compiler translates the entire program in one go and then executes it	1. Interpreter first converts high level language into an intermediate code and then executes it line by line. The intermediate code is executed by another program
2. It produces efficient object code therefore programs runs faster	2. No intermediate object code is generated
3. Error reporting is time consuming (displayed after entire pgm is checked)	3. Errors are displayed for every instruction interpreted if any. (Error reporting is immediate)
4. Conditional control statements are executed faster	4. Conditional control statements executed slower
5. Memory requirement is more ∵ single object code is generated	5. Memory requirement is less
6. Program need not be compiled every time	6. Everytime high level program is converted into lower level pgm
7. Difficult to use	7. Easy to use for beginners
8. Translate once and then run the result (stand-alone code, faster exn)	8. read - check - execute loop -> slower, not stand-alone
9. Eg:- C, C++	9. Eg:- python, prolog

10. 

10. 

→ Examples showing detail phases of compiler :

① position = initial + rate * 60

↓

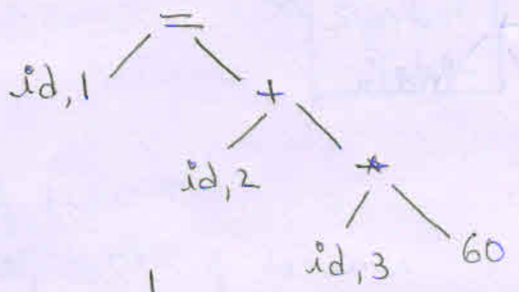
Lexical Analysis

<id,1> <=> <id,2> <+> <id,3> <*> <60>

↓

Syntax Analysis

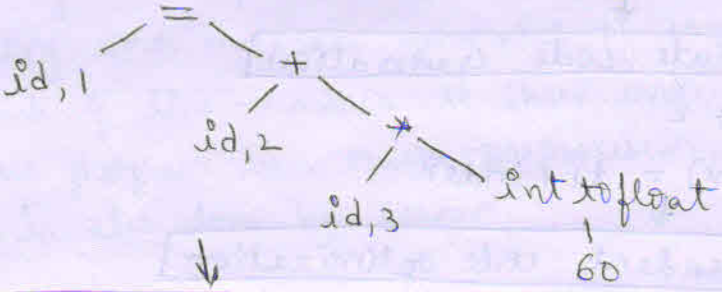
1	position	
2	initial	
3	rate	



Syntax Tree

↓

Semantic Analysis



↓

Intermediate code Generation

t_1 = inttofloat(60)
t_2 = id_3 * t_1
t_3 = id_2 + t_2
id_1 = t_3

↓

M/c independent code optimization

t_1 = id_3 * 60.0
id_1 = id_2 + t_1

⟶ **Code Generation**

```
LDF  R1, id3
MULF R1, R1, #60.0
LDF  R2, id2
ADDF R2, R2, R1
STR  id1, R2
```

② a[index] = 4+2+ index

↓

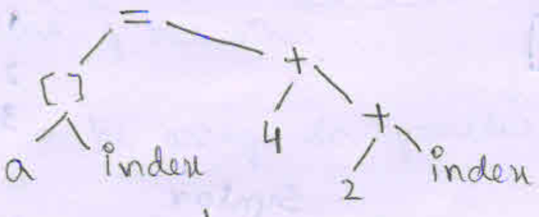
[Lexical Analysis]

<id,1> <[> <id,2> <]> <=> <4> <+> <2> <+> <id,2>

↓

[Syntax Analysis]

1	a	
2	index	



↓

[Semantic Analysis]

↓

[Intermediate code Generation]

$t_1 = 4 + 2$

a[index] = t_1 + index

↓

[M/C independent code optimization]

a[index] = 6 + index

↓

[Code Generation]

```
mov index, R0      // R0 = index
mov &a , R1        // R1 = Starting address of array a
add R0 , R1        // R1 = R0+R1
mov #6 , R2        // R2 = 6
add R1, R2         // R2 = R1+R2 = R1 + 6
mov R2, &R1        // Store R2 in &R1 i,e &a's value
```

→ Environments and States:

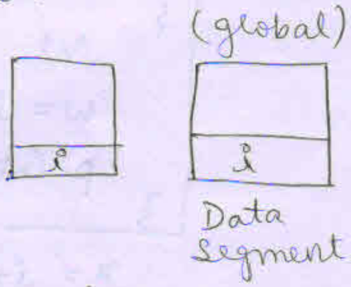
names $\xrightarrow{\text{environment}}$ locations (variables) $\xrightarrow{\text{state}}$ values

- Environment is mapping from names to locations in the store
- State is mapping from locations in store to their values.

Dynamic Mapping Exceptions:

(i) Static Binding of Names to Locations - global variable declaration - location in store once for all.

Ex:
```
int i;                  // global i
void fun (...) {
    int i;              // local i
}
```



(ii) Static Binding of Locations to values - declared constants

Ex: `# define ARRAYSIZE 1000` //static bind

Static Scope and Block Structure:

(1)
```
main () {
    int a=1;                      B1
    int b=1;
    {
        int b=2;                  B2
        {
            int a=3;
            cout <<a<< b;         B3
        }
        {
            int b=4;
            cout <<a<< b;         B4
        }
        cout <<a<<b;
    }
    cout <<a<<b;
}
```

→

Declaration	Scope
int a=1 ;	B1–B3
int b=1 ;	B1–B2
int b=2 ;	B2–B4
int a=3 ;	B3
int b=4 ;	B4

(2)

```
main ()
{
  int w,x,y,z;
  int i=4; int j=5;
  {
    int j=7; i=6;
    w = i+j;
    printf (w);                 → 6+7 = 13
  }
  x = i+j;
  printf(x);                    → 6+5 = 11
  { int i=8;
    y = i+j;
    printf(y);                  → 8+5 = 13
  }
  z = i+j;
  printf(z);                    → 6+5 = 11
}
```

(3)

```
#define a (x+1)
int x = 2;
void b() { x=a; printf("%d",a); }        O/P 3
void c() { int x=1; printf("%d", a); }       2
void main ()
{ b(); c(); }
```

④

```
int w,x,y,z;
int i=3;
int j=4;
{
  int i=5;
  w = i+j;            5+4=9
}
  x = i+j;            3+4=7
{ int j=6;
  i = 7;
  y = i+j;            7+6=13
}
  z = i+j;            7+4=11
```

10. what is printed by the following c cod.

```
a) #define a (x+1)
   int x=2;
   void b() { x=a; printf("%d\n", x); } → 3
   void c() { int x=1; printf("%d\n", a); } → 2
   void main() { b(); c(); }
```

```
b) # define a (x+1)
   int x=2;
   void b() { x=a; printf("%d\n", x); } → 3
   void c() { printf("%d\n", a); } → 4        ∵ redesignment for the
                                               same variable x
   void main() { b(); c(); }                   x=3 , a=3+1=4
```

```
c) #define a (x+1)
   int x=2;
   void b() { int x=1; printf("%d\n", a); } → 2
   void c() { printf("%d\n", a); } → 3
   void main() { b(); c(); }
```

```
d) #define a (x+1)
   int x=2;
   void b() { int x=a; printf("%d\n", a); } → 4 ∵ ( x=2+1=3
                                                   again a=3+1=4)
   void c() { printf("%d\n", a); } → 4
   void main() { b(); c(); }
```

→ Parameter Passing Mechanisms :

(1) Actual parameters - parameters used in call of procedure
(2) Formal parameters - parameters used in procedure definition

① Call by value

- The actual parameter is evaluated (if an expression) or copied (if a variable), The value is placed in the location belonging to corresponding formal parameter of called procedure.
- It has all computations involving formal parameter done by called procedure is local to that procedure.

② Call by reference

- The address of actual parameter is passed to the callee as the value of corresponding formal parameter
- Uses of formal parameter in code of callee are implemented by following this pointer to location indicated by caller.
- changes to formal parameter ⇒ Appear as changes in actual param

- If actual parameter is expression, it is evaluated before the call and its value stored in a location of its own.
- changes to formal parameter change value in this location, But - No effect on data of caller.

③ Call by name

- used in early prog - Algol 60.
- it requires callee execute as if actual parameter were substituted literally for formal parameter in the code of the callee as if formal parameter were macro standing for the actual parameter.

→ Examples :

① call by value

```
int add ( int a, int b )
  {
     return (a+b);
  }
main ()
  {
    :
     c = add ( 10, 20)
    :
  }
```

② call by reference

```
int add ( int *a, int *b)
  {
     return (a+b);
  }
main ()
  {
    :
       int p = 10;
       int q = 20;
       c = add (&p, &q);
    :
  }
```

③ call by Name - Aliasing

```
int add (int a , int b)
  {
     return (a+b);
  }
main ()
  {
       int p = 10;
       int q = 20;
       c = add ( &p, &q);
  }    :
```

• Aliasing:

→ Interesting consequence of call by reference parameter passing where references to objects are passed by value.

• It is possible that two formal parameters refer to the same location – such variables are ALIAS to one another.

• Though they may be distinct formal parameters, they may be Alias of one other.

Ex: Let a be array in procedure p

```
p
{
    // q(x,y)   call
    q(a,a);
}
```

array names are references to location ⇒ Alias

$x[i] = y[i]$

Questions

Chapter-1 Introduction

1. Define Compilers?
2. Differentiate b/w compilers & Interpreter?
3. Explain The long processor System?
4. Describe the analysis-Synthesis model of the Compiler or Explain in detail the Various phases of Compiler with an example?
5. Explain in detail the Various phases of Compilation for The i/p string

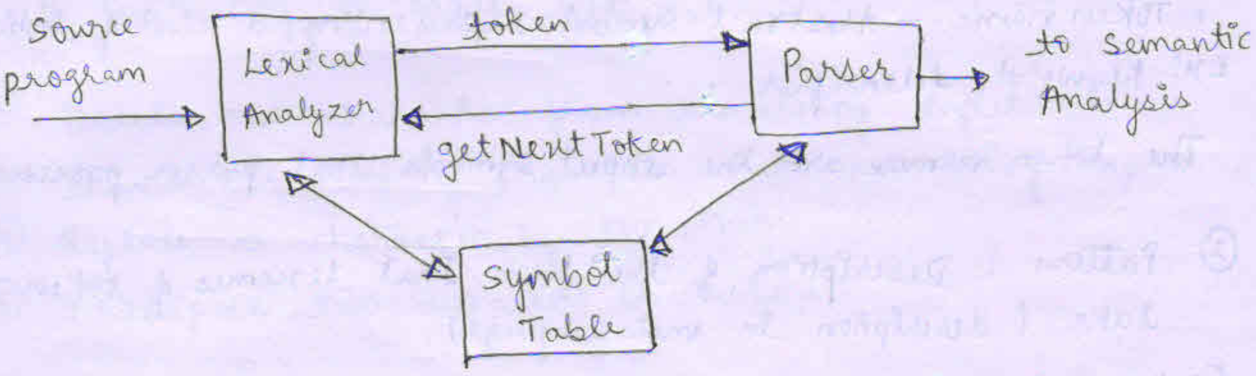
   a. $p = i + r * 60$ c. $a = (b+c)*(b+c)*2$

   b. $x = a*b + a*b$ d. $a[index] = 4 + 2 + index$
6. why is it necessary to group phases of Compiler
7. what Is the purpose of Compiler Constn tool. Describe the different Compiler Construction tool we used?
8. Analyse the s/w productivity tool and explain
9. Explain the different parameter passing technique with an example?

Chapter-3 - Lexical Analysis

→ Lexical Analysis :

Interaction between Lexer and Parser :



→ Task of Lexer :

1. Identification of Lexemes
2. Stripping out comments
3. Removing whitespace (blank, \n, \t)
4. corelating error messages generated by compiler
5. Keep track of line numbers to show error
6. If source program uses Macro-preprocessor, the expansion of macros is also done by scanner.

→ Lexer - cascade of 2 processes :

1. Scanning consists of simple processes that do not require tokenization of input, such as deletion of comments & compaction of consecutive whitespace characters into one.

2. Lexical Analysis proper in more complex portion, where scanner produces sequence of tokens as output.

Lexer versus Parser : Seperate phases because :

1. simplicity of compiler design - important consideration
2. compiler efficiency improved - use specialized technique for lexical analysis (Input Buffering)
3. compiler portability is enhanced.

→ Tokens, Patterns, Lexemes :

① Token : A pair consisting of a token name and an optional attribute value.

• Token name – Abstract symbol representing a kind of lexical unit
Ex: Keyword, identifier ....

The token names are the input symbols that parser processes.

② Pattern : Description of the form that lexemes of token may take (description in metalanguage).

Ex: token name : identifier
pattern : [_a-zA-Z]$^+$ [a-zA-Z0-9]*

③ Lexeme : Sequence of characters in source program that Matches the pattern of a token and is identified by the lexer as an instance of that token.

Ex: token name : Keyword
pattern : [i][f]
lexeme : if

Token	Informal Description	Sample Lexemes
if	characters i, f	if
else	characters e, l, s, e	else
comparision	<, >, <=, >=, ==, !=	<=, <>
id	letter followed by letter and digits	pi, score
number	numeric constants	3.14, 0, 6.9e8.
literal	enclosed within " "	"core dumped"

→ Lexical Errors : Recovery options

① Panic mode recovery - delete successive characters from remaining input until lexer finds well known token at beginning of input left out.

② Delete one character from remaining input
③ Insert one missing character into remaining input
④ Replace a character by the other
⑤ Transpose two adjacent characters.

Examples :

fi (a<b) ⇒ if (a<b)
int a, ; ⇒ int a; or int a, b;

→ Input Buffering : To speed up reading of src prog.

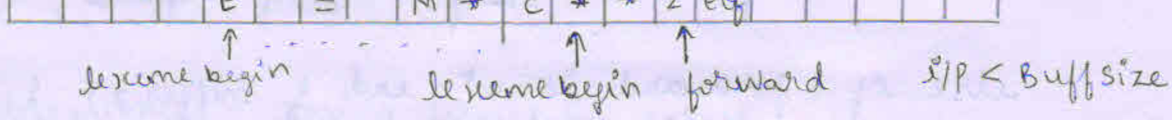
① Single Buffer | 1-Buffer Technique
We use only one single buffer to store processed character from large no. of characters from source prog.
Main overhead is that if,

$\boxed{\text{lexeme size > Buffer size}}$ World → | W | o | r | l | (d)

we loose the lexeme 5 Bytes 4 Bytes

· It reloads data, removes old data.

② 2-Buffer Technique < without sentinel / with sentinel
We use two buffers that are alternately reloaded,
Each buffer of same size N, N = size of a disk block. (4096 Byte)
· Using read system call, N characters are read.



→ Special char eof marks end of src file and this char is different from any other char of src prog.

→ Two pointers maintained :

① Lexeme Begin – marks beginning of current lexeme whose extent we are attempting to determine.

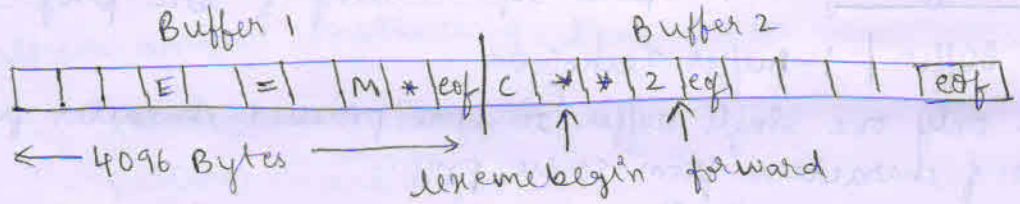
② Forward – scans ahead until a pattern match is found. When forward reaches end of next lexeme, ** we <u>retract</u> <u>one</u> <u>position</u> <u>back</u> and return token.

• We need 2 checks in 2 Buffer without sentinels :

1) Advancing forward requires whether we reached the end of one of the buffer, if Yes Reload other buffer and make forward point to newly loaded buffer beginning.

2) Before returning token check whether valid or not.

→ Sentinels : (2 Buffer technique with Sentinels)
Using sentinel character at the end which is a special char that is not part of src prog (usually eof)



Here check if reached end of Buffer or not.
Look Ahead is atmost 1 char, make previous char as returned valid token.

→ <u>Look Ahead code with sentinel</u> :

```
Switch ( * forward ++ )
{
   case eof : if (forward is at end of Buffer 1) {
                  reload Buffer 2 ;
                  forward = Beginning of Buffer2 ; }

   else if (forward is at end of Buffer 2) {
                  reload Buffer 1 ;
                  forward = Beginning of Buffer 1 ; }
```

else /* eof with in a Buffer marks end of input */
terminate lexical Analysis
break;
cases for other char
}

---

(1) Alphabet - finite set of symbols Ex: $\Sigma = \{0,1\}$
String - finite sequence of symbols from Σ Ex: 0101
Language - countable set of strings over Σ.

(2) Prefix of string - string obtained by removing zero or more symbols from end of string.
Ex: ban, banana, ϵ are prefixes of banana.

(3) Suffix of string - string obtained by removing zero or more symbols from beginning of string.
Ex: nana, banana, ϵ are suffixes of banana

(4) Substring - string obtained by deleting any prefix and any suffix from string.
Ex: banana, nan, ϵ are substrings of banana

(5) Proper prefix - prefixes, which is not ϵ or equal to string
Ex: ban, banan

(6) Proper suffix - suffix which is not ϵ or equal to string itself
Ex: anana, na

(7) Proper substring - substring from string which is not ϵ or the string itself
Ex: anan, banan, anana

Subsequence - string formed by deleting zero or more not necessarily consecutive positions of string.
Ex: baan, anaa -- for banana

→ Operations on Languages :

operation	definition & notation
Union of L & M	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L & M	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

→ Regular Definition :

For some alphabet set Σ, sequence of regular definition ;

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

⋮

$d_n \rightarrow r_n$ where

1) each d_i is new symbol (not in Σ & other d_i)
2) r_i is regular expression over $\Sigma \cup \{ d_1, d_2 \ldots d_{i-1} \}$

Ex ① C identifiers :

letter → A | B | - - | Z | a | b | - - | z | _

digit → 0 | 1 | - - | 9 |

id → letter (letter | digit)*

Ex ② Unsigned numbers :

digit → 0 | 1 | —— | 9 |

digits → digit digit*

optional fraction → . digits | ϵ

optional exponent → (E (+ | - | ϵ) digits) | ϵ

number → digits optional fraction optional exponent

Algebraic laws for Regular Expressions

LAW	DESCRIPTION
1. $r\|s = s\|r$	| is commutative
2. $r\|(s\|t) = (r\|s)\|t$	| is associative
3. $r(st) = (rs)t$	Concatenation is associative
4. $r(s\|t) = rs\|rt$; $(s\|t)r = sr\|tr$	concatenation distribute over |
5. $\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
6. $r* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
7. $r** = r*$	* is idempotent

→ Recognition of Tokens:

stmt → if expr then stmt | if expr then stmt else stmt | ϵ

expr → term relop term | term

term → id | number

where,

number → $[0-9]^+ \, (.[0-9]^+)? \, (E[+-]?[0-9]^+)?$

id → $[\_a\text{-}zA\text{-}Z][\_a\text{-}zA\text{-}Z0\text{-}9]^+$

if → if

then → then

else → else

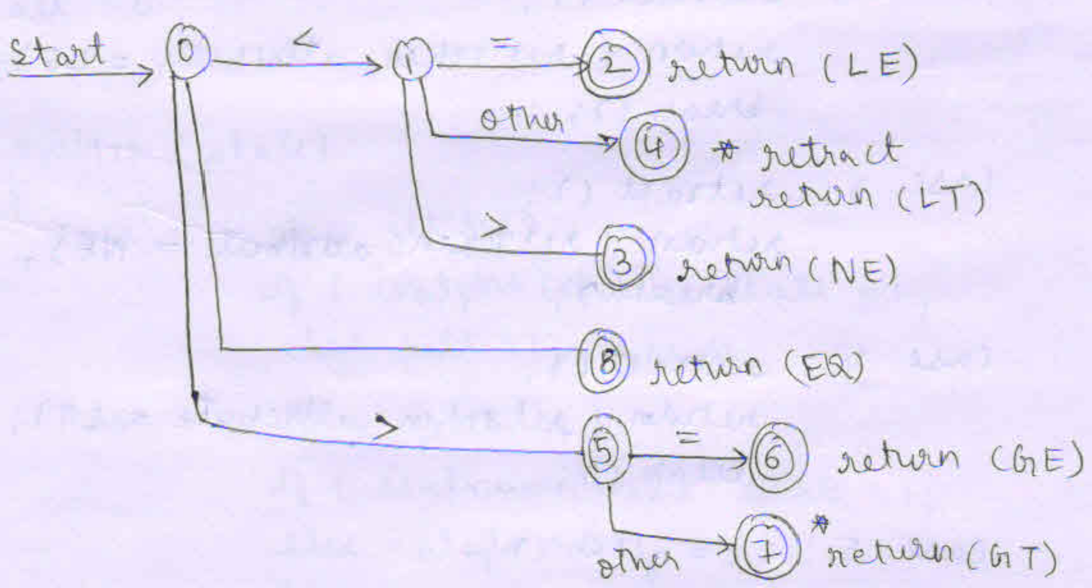
relop → $< \mid > \mid <= \mid >= \mid = \mid <>$

white space : ws → $(\text{blank} \mid \text{tab} \mid \text{newline})^+$

→ Transition diagram:

(1) For relational operator, regular definition is

relop → $< \mid > \mid <= \mid >= \mid = \mid <>$



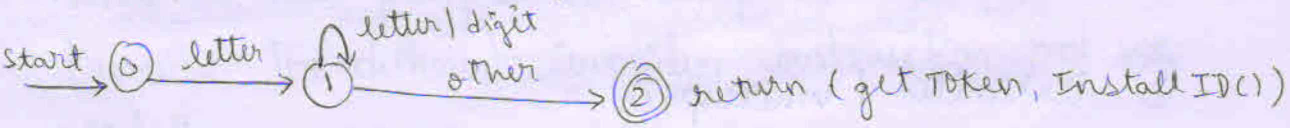
code:

```
state = 0 ;
TOKEN getrelop ()
{
   Token retToken = new (relop);
   while (1)
   {
    switch (state)
    {
      case 0 : c = newchar () ;   or  c = getch ();
      if ( c == '<' ) state = 1;
      else if ( c == '>' )  state = 5 ;
      else if ( c == '=' )  state = 8 ;
       else fail () ;   break ;

    case 1 : c = getch () ;
     if ( c == '=' )  state = 2 ;
     else if ( c == '>' ) state = 3 ;
     else if ( c == '...' )  state = 4 ;   // other
      else fail () ; break ;

    case 2 : retract () ;
             return ( retToken . attribute = LE ) ;
             break () ;

    case 3 : retract () ;
             return ( retToken . attribute = NE ) ;
              break () ;

     case 4 : retract () ;
              return ( retToken . attribute = LT ) ;
              break () ;

     case 5 : c = getch () ;
      if ( c == '=' )  state = 6 ;
      else if ( c == '...' )   state = 7 ;  // other
      else fail () ; break () ;
```

```
case 6 : retract ();
         return ( retToken . attribute = GE);
         break ;
case 7 : retract ();
         return ( retToken . attribute = GT);
         break ;
case 8 : retract ();
         return ( retToken . attribute = EQ) ;
         break ;
      }
   }
}
```

(2) for identifier

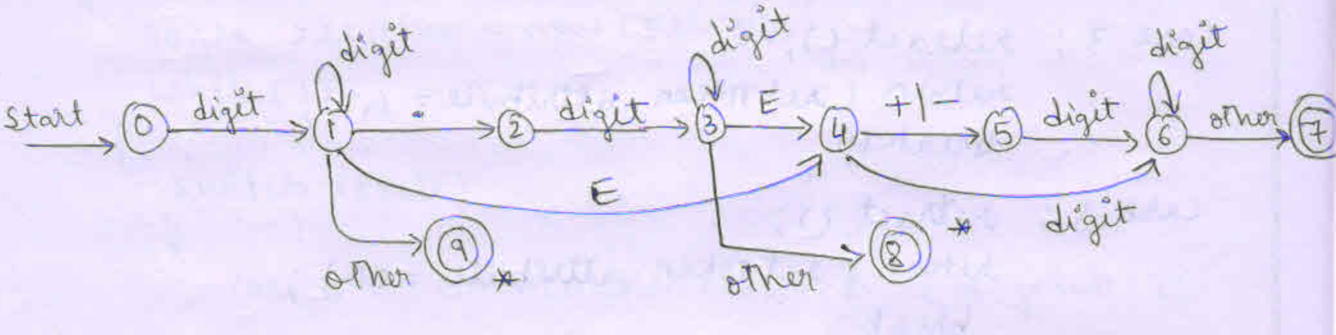
letter → [a-zA-Z_]
digit → [0-9]
id → letter (letter | digit) *




```
state = 0;
for ( ; ; )
{
  switch (state)
  {
    case 0 : ch = getch ();
             if ( isalpha (ch))  state = 1;
             else fail ();  break ;
    case 1 : ch = getch ();
             if ( isalnum (ch))  state = 1;
             else  state = 2 ;
             break;
    case 2 : retract ();
             InstallID ();
             return ( retToken );
             break ;
  }
}
```

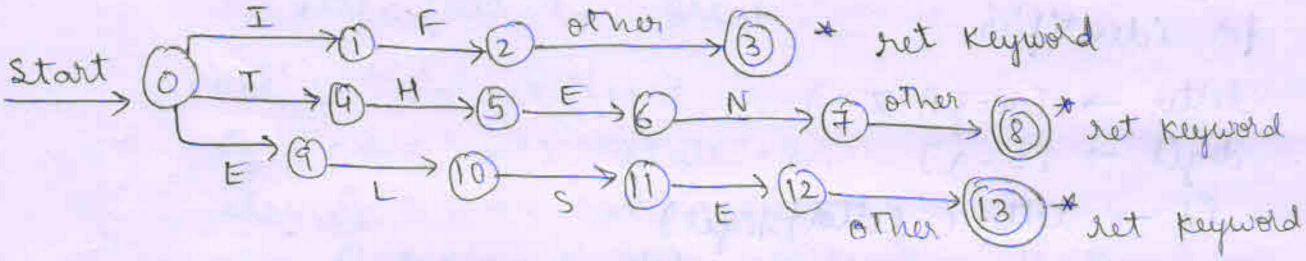
③ unsigned number

digit → [0-9]



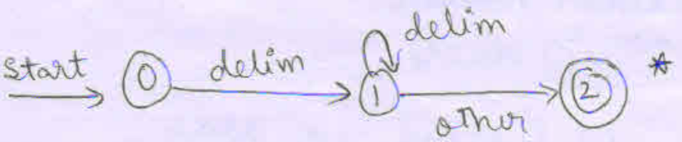
④ Keywords

Ex: keyword → IF | THEN | ELSE



⑤ Delimiter | Whitespace

delim → space | tab | newline



chapter-2 Lexical Analysis

Questions

1. Explain Lexical Analysis in detail with block diagram
2. Explain the reason for separating analysis phase of Compiler for lexical Analysis and Syntax Analysis
3. What do you mean by lexical errors? How do we recover them
4. Define the terms token, pattern, lexeme with an example.
5. why 2-buffer technique is used in LA? write an algorithm for lookahead code with sentinel.
6. Give the formal definitions for operations on languages with notations
7. list the algebraic laws for Regular Expression.
8. Define the term prefix, suffix, substring, proper prefix, proper suffix, proper substring, subsequence with an example.
9. write regular definition for Identifiers, unsigned numbers, keywords, relational operators and whitespace
10. Draw the transition diagram for relop, identifiers, unsigned number, keywords and white spaces.