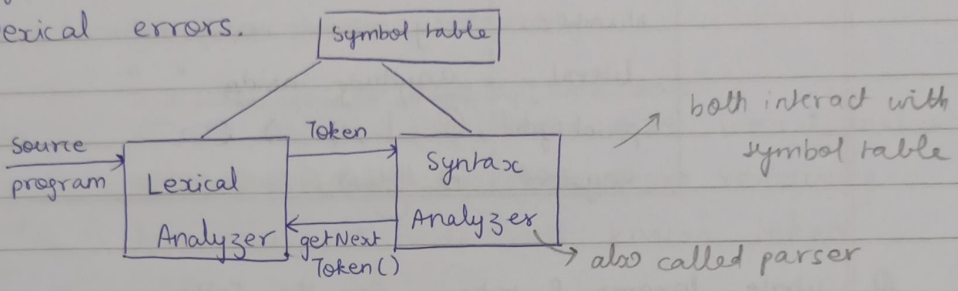
# CH.3 LEXICAL ANALYSER (SCANNER)

- 1st phase of compiler
- reads input (source program)
- identifies lexemes
- generates tokens
- creates symbol table ~~for~~ entries for identifiers (variables)
- detects lexical errors.




i.e ~~reads~~ syntax analyzer invokes lexical analyzer, only then lexical analyzer reads a set of characters and identifies ~~the~~ next lexeme, converts it to token. Once syntax analyzer processes a token it asks for next token, only then lexical analyzer reads next set of chars.

- Discards comments & whitespaces (tab, blank, \n)
- Correlates error message with source program (i.e identifies which position error takes place)
- Lexical analyser is divided into 2 parts
  - Scanning
    - Expansion macros - (#define etc.)
    - preprocessing directories
    - discarding whitespaces
  - Tokenization

→ Token is a pair consisting of a token name and optional attribute value.

→ Pattern is a description of the form that the lexeme of a token may take (i.e possible values for token)
They are represented using regular expressions.

eg:

| Tokens | Patterns |
|---|---|
| if | 'i' followed by 'f' |
| else | 'e' followed by 'l' followed by 's' followed by 'e' |
| comparison | <, >, <=, >=, ==, != |
| id | letter/underscore followed by letters/digits |
| Number | any numeric constant |
| literal | anything inside " " |
| Special symbol | , @, [, ;, ), ( (comma also special) |
| ~~keyword~~ | ~~int, float, const, etc.~~ |

(key words)

Q Write lexeme & token for the following C instruction

printf ("Total = %d\n", score);

| Lexeme | Token |
|---|---|
| printf | id |
| ( | LCB or LP or ( (left common bracket, left parenthesis) → which token depends on the chosen grammar. |
| "Total = %d\n" | literal |
| , | comma or , |
| score | id |
| ) | RCB or RP or ) |
| ; | SC → semicolon |

## LEXICAL ERROR

→ Lexical Errors are detected when none of the patterns listed for the given language is matched.

eg: int 1abc ;

here, 1abc is wrong but it won't be considered a lexical error.

| lexeme | token |
|---|---|
| int | <int> |
| 1 | <num, 1> |
| abc | <id> |
| ; | <special symbol> |

1abc is a syntax error.

→ Any language other than english is a lexical error.
eg: int मैंने ;

## LEXICAL ERROR RECOVERY TECHNIQUES

→ It recovers using panic mode recovery.

→ Panic Mode - if it finds any not matching (listed) chars ~~if it notices any errors~~ i.e. lexical analyser deletes successive characters from remaining input unit it finds well known token
↳ basically delete and move on

→ Other recovery strategies are:
1. Delete 1 char from remaining input
2. insert missing char to remaining input
3. Replacing char by another char
4. Transferring this char to the next char, 1abc becomes ab1c.

↳ how is this lexical error?

## INPUT BUFFERS

→ Instead of reading each char from hardware again & again we use input buffers to store input (source) code.

→ This helps with faster retreival and processing.

→ After reading a set of characters we still need to look ahead to understand if it is a keyword, variable or something else.

eg: int a;
float interested;

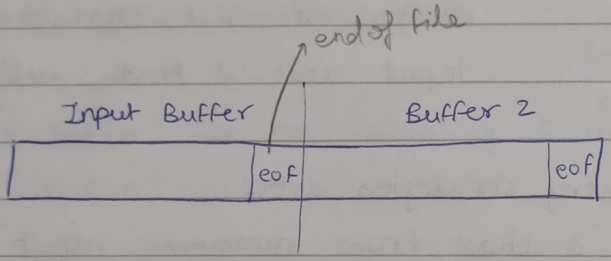
here after reading "int" of interested we still need to look ahead since it is a variable else it will be misunderstood

eg: if ( a < b )
if ( a <= b )

in second statement we need to look ahead after <
as it is <=

→ Input buffers help with making look ahead easier.

Buffer Pairs




Both of same size.

→ In single buffer scheme once the buffer is full, tokens are generated for it. Once that is done and eof is encountered, the whole buffer is reloaded with the remaining input data that could not be stored earlier. Now tokens are generated for the remaining data.

→ Due to this we face issues with look ahead and wrong tokens may be generated. (i.e wrong lexeme identified)

eg: | f | l | o | a | t | i | n | t | eof |

float interest;

<float> <int>

after eof it is flushed and rest of data produced

| e | r | e | s | t | eof |

it thinks erest is a variable

<id, 1>

∴ Wrong tokens <float> <int> <id, 1>

but correct was <float> <id, 1> where id1 = interest.

→ ∴ whenever eof encountered buffer is flushed and remaining data is loaded.

→ To solve this problem we have 2 buffers.

→ In buffer pairs alternately two buffers are loaded. When eof of 1st buffer encountered, 2nd buffer is loaded.

→ eof is a sentinal used to denote end of buffer. To differentiate b/w end of input and end of buffer (since both denoted by eof) we tally no. of byte read with no. of bytes scanned so far. If scanned = read then end of input
scanned < read then end of buffer

→ Two pointers are used to maintain buffer. i.e.
lexeme begin & forward pointer.

→ lexeme begin marks the beginning of the current lexeme.
→ forward pointer scans ahead until a pattern match is found.

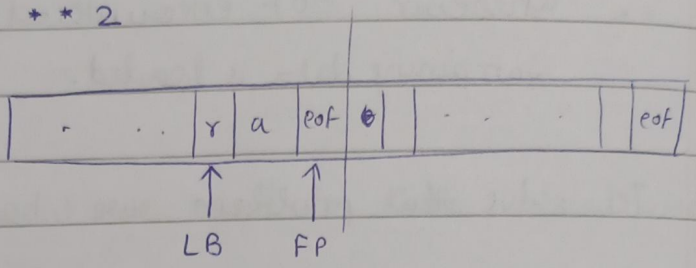
→ lexeme patterns are denoted by regular expressions. Each regex has a finite automata (FA)
when a char is read it is also put into the FA for each of these regex by lexical analyser.

→ As each char is read it might get rejected by some FA while it is still being read by other FAs.
→ This goes on until all except one FA reject the input. And that final FA also accepts the lexeme.
(if no FA accepts it then it is a lexical error)

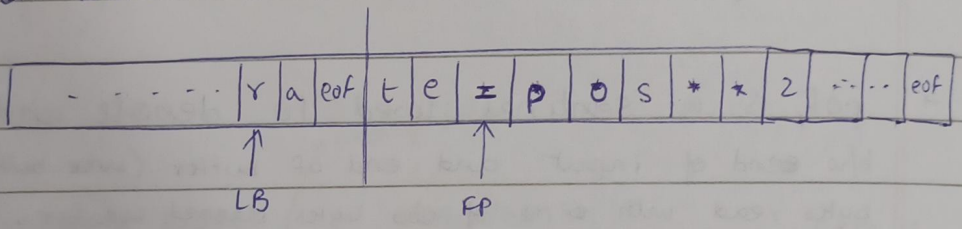
→

→ eg: rate = pos ** 2

initially lets say,




FP keeps reading each character. On reaching eof the second buffer is loaded.




When FP reads = it retracts to previous index containing 'e' & matches 'rate' to identifier pattern.

Note: Retraction is not required for single characters like ( , ) , comma , ; , etc. because it is obvious that they don't need lookahead.

---

## REGULAR DEFINITION

regular definition

letter → A | B | · · · | Z | a | b | · · · | z | _ (underscore)

digit → 0 | 1 | · · · | 9

id → letter ( letter | digit )*

λ or ∈ → null

written as superscript:
* → one or more occurrance
+ → 1 or more occurrance

written normal sized:
? → 0 or 1 occurrance

Regex for identifier: [A-Z a-z _] [A-Z a-z _ 0-9]*

(hyphen not used; _ = underscore)

Q Write regular definition that matches all of these → minus 4

5280, 0.01234, 6.336E4, 5E5, 1.89E-4

Sol"

digit → [0-9] or digit digit* both are same

digits → digit+

~~fraction → digits . digits~~

optional fraction → . digits | ∈ → null

optional Exponent → E (+|-|∈) digits | ∈ → null

number → digits optionalfraction optionalexponent

null is used in optional frac & optional exp as you may or may not have decimal part or exponent

∴ number accepts all above eg:s.

The above can be written using regular extensions in a shorter way

digits → [0-9]+

number → digits (. digits)? ( E [+-]? digits )?

(? → 0 or 1 occurrance)

---

## RECOGNITION OF TOKENS

### Grammar

1) Grammar for if statement

~~stmt → if expr then stmt~~ → only if

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

| ∈

expr → term relop term | term

relop → relational operator like <, >, <=, >=, =, <> (= → equal; <> → not equal)

in FORTRAN == is written as .EQ.
!= " " " .NE.
<> " " " .LT.

term → id | num

atomic i.e dont have production rule

Terminals : id, num, if, then, else, relop

Non-Terminals : stmt, expr, term

∈ is not considered as terminal here as it is not considered for tokens

→ every terminal we write regular definition

→ Every compiler is given grammar of the language. The terminals in this grammar are used as token names.

## Regular Definition

→ Writing Regular definition for all terminals

eg:
digit → [0-9]
digits → digit+
number → digits . (digits)? ( E [+-]? digits )?
letter → [A-Z a-z]
id → letter ( letter | digit )*
relop → < | > | <= | >= | = | <> → not equal
if → i followed by f
else → e followed by l followed by s followed by e.
whitespace ws → ( blank | tab | newline )+

When whitespace is encountered, token is generated by lexical analyser but not sent to parser.

## TRANSITION DIAGRAM

→ Similar to Finite Automata (FA)

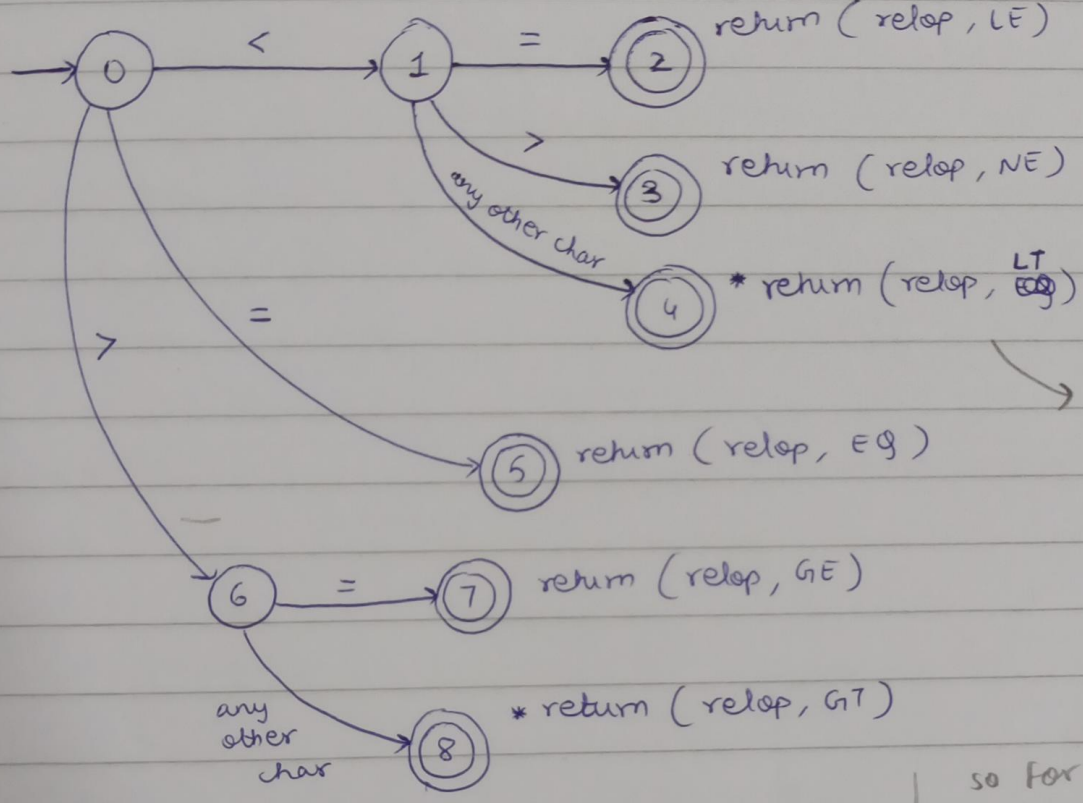
→ Every accepting state returns a token. (FA does not return anything)

→ Whenever retract operation occurs it must be represented using *

→ accepting state denoted by ◎ return ( <tokenname, attribute> )

# TRANSITION DIAGRAM FOR RELOP

- Relop transition diagram is only invoked if LB (lexeme begin) is at <, > or =




<= LE
<> NE
< LT
= EQ
>= GE
> GT

if < is followed by anything but = or > it means it is less than symbol (LT)

so for eg: a < b
here < is followed by b ∴ < is LT
we write * to show retract so that the b being read is not ignored.
This way b will be read again for "id" transition diagram

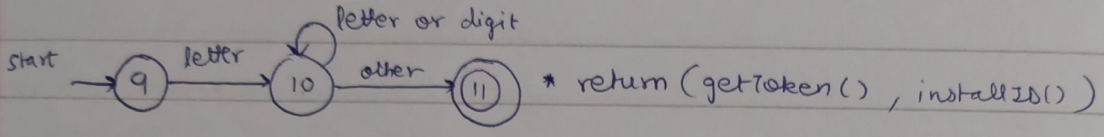
# TRANSITION DIAGRAM FOR ~~UNSIGNED~~ NUMBER

check slides & some more transition diagrams in slides

## RECOGNITION OF RESERVED WORDS & IDENTIFIERS

(transition diagram for id's & keywords)

case 1:

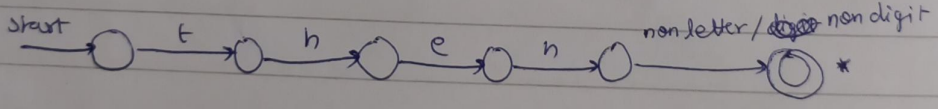



→ only when an identifier is encountered for the first time it is added to symbol table.
If it is encountered again, it is found in symbol table and its type is checked. (if keyword then keyword is returned else if identifier then serial no. of identifier returned)

case : 2

→ Another method is to create individual transition diagrams for each keyword.

eg: then is a keyword




here after "then" if any other non-letter or nondigit is encountered it means keyword has ended. and we backtrack that one symbol and accept "then"

eg: then 1

upon reaching 1 it realises keyword has ended and then is accepted

eg: thence

here after "then" c is not non-letter / non-digit and DFA will not accept this string as then was just a substring in thence not whole word.

# ARCHITECTURE OF A TRANSITION DIAGRAM BASED LEXICAL ANALYZER

## Implementation of relop transition diagram

↳ drawn earlier refer to it for states

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP);
    while(1) { /* repeat char processing until a return or failure
                  occurs */
        switch (state) {
            case 0: c = nextChar();
                    if (c == '<') state = 1;
                    else if (c == '=') state = 5;
                    else if (c == '>') state = 6;
                    else fail(); /* retracts the chars read
                                    during this getRelop() fn as
                                    the lexeme was not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return (retToken);
        }
    }
}
```

start state

↳ eg: in C++ cout << here, < means getRelop() is called but actually it is not a relop so fail().

case 2, 3, 4, 5, 7, 8 are accepting states acc. to transition diagram

accepting state

→ Code not complete we must write on our own.

→

~~case 1 : c = next~~

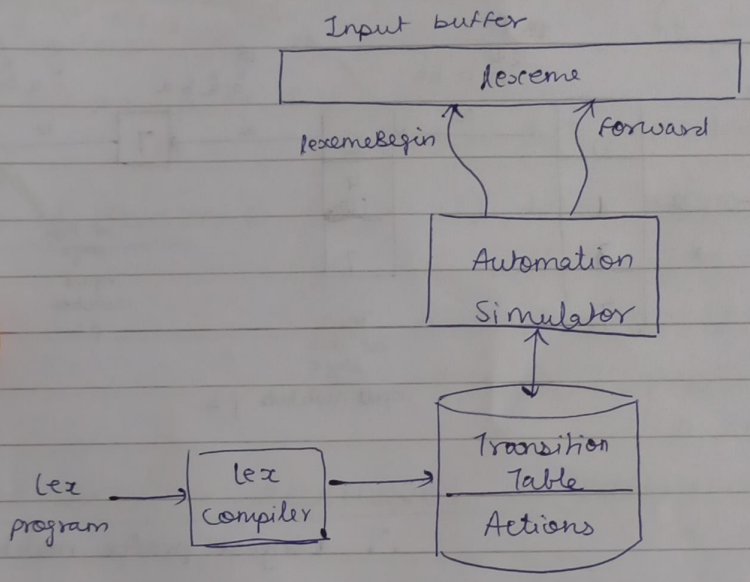
```
case 1 : c = nextChar();
         if (c == '=') state = 2;
         else if (c == '>') state = 3;
         else {
                retract();
                ret
         else state = 4;
         break;

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

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

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

case 5 : retToken.attribute = EQ;
         return (retToken);

case 6 : c = nextChar();
         if (c == '=') state = 7;
         else state = 8;
         break;

case 7 : retToken.attribute = GE;
         return (retToken);
```

# FINITE AUTOMATA

→ Doesn't return anything

→ 2 types: NFA & DFA

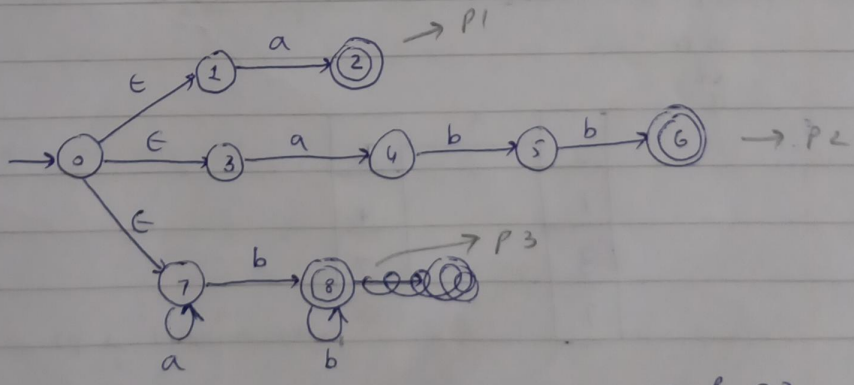
## DESIGN OF LA (lexical analyser) generator using FA




lex compiler creates transition table

→ We will only use NFA
↳ ∈ Transitions (check FLAT)

eg:




i) here, abb is accepted by both matched P2 & P3
This is a conflict.

ii) also out of abb, when a is read, that matches P1 even before reading bb.

To solve problem ii)

→ To prevent conflict we match the longest prefix during pattern matching.

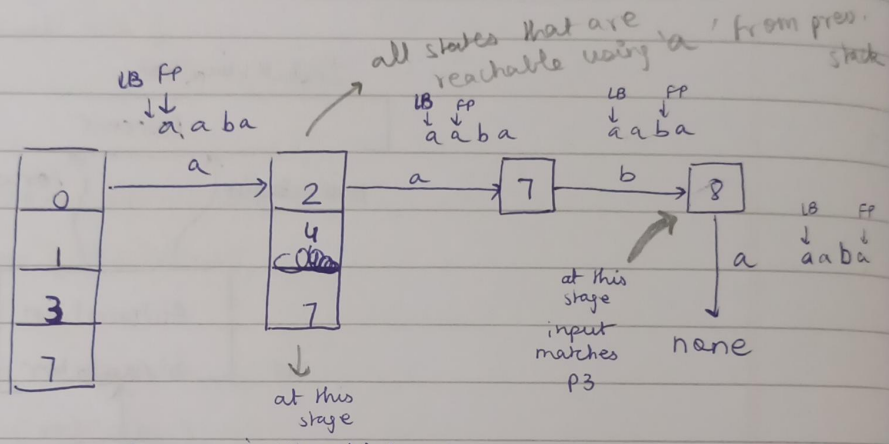
To solve problem i)

→ Also if multiple patterns match then we choose the first pattern that matches.

∴ b/w p2 & p3 , p2 will be chosen.

pattern matching using NFA

eg: a a b a




Stack → with states

This one is epsilon closure of start state 0

i.e all states reachable using ε from state 0

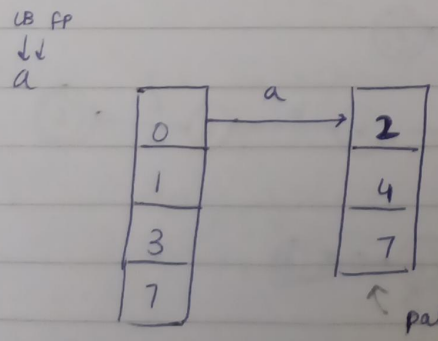
ε-closure (0) = {0, 1, 3, 7}

∴ Longest prefix matched is aab

pattern for that is p3

∴ lexeme = aab

now, LB will reset to a a b a as that was not matched.




after this no more input

∴ 'a' is another lexeme.