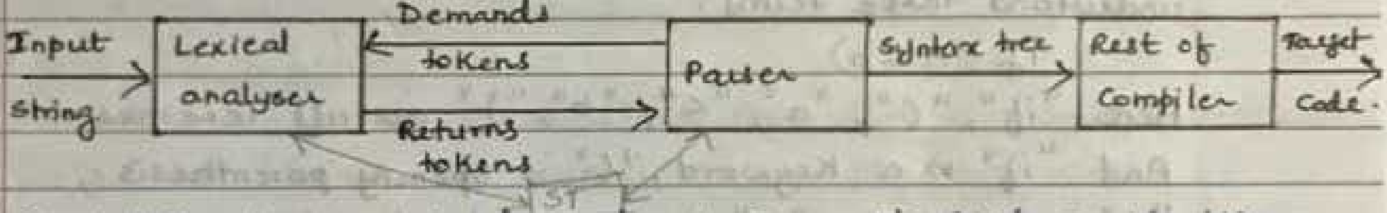
LEXICAL ANALYSIS

- The process of compilation starts with the first phase called lexical analysis. In this phase the input is scanned complitely in order to identify the tokens.
- The token structure can be recognized with the help of some diagrams. These diagrams are popularly known as finite automata.
- And to construct such finite automata regular expressions are used.
- These diagrams can be translated into a program for identifying tokens.

Role of Lexical Analyzer

- LA is the first phase of compiler. The LA reads the input source program from left to right one character at a time and generates the sequence of tokens.



- Each token is a single logical cohesive unit such as identifier, Keywords, operators and punctuation marks.
- Then the parser is used to determine the syntax of the source program using these tokens.
- As the LA scans the source program to recognize the tokens it is also called as scanner.
- Apart from token identification LA also performs foll. functions:

Functions of LA :-

1) It produces stream of tokens.
2) It eliminates blank and comments
3) It generates symbol table which stores the inf^{m} about identifiers, constants encountered in the i/p.
4) It keeps track of line nos.
5) It reports the error encountered while generating the tokens.

- The LA works in 2 phases. In first phase it performs scan and in the second phase it does lexical analysis., means it generates the series of tokens.

Tokens, Patterns, Lexemes

Tokens : It describes the class ~~of~~ or category of i/p string. for eg., identifiers, Keywords, constants are called tokens.

Patterns: Set of rules that describe the token. It can be defined by regular expressions or grammar rules.

Lexemes: Sequence of characters in the source pgm that are matched with the pattern of the token.
eg:- int, i, num, ans, choice.

- Let us take one eg of programming statement to clearly understand these terms:

```
if (a<b)
```

Here "if", "(", "a", "<", "b", ")" are all lexemes.
And "if" is a Keyword, "(" is opening parenthesis, "a" is identifier, "<" is an operator and so on.

- Now to define the identifier pattern could be -
  1) Identifier is a collection of letters
  2) Identifier is collection of alphanumeric characters and identifier's beginning character should be necessarily a letter.

Eg. The piece of source code is given below.

```
int MAX (int a, int b)
{
   if (a>b)
   return a;
   else
   return b;
}
```

Lexeme	Token.
int	Keyword
MAX	identifier
(	operator
int	Keyword
a	identifier
,	operator
int	Keyword
b	identifier
)	operator
{	operator
if	Keyword
⋮	⋮

- The blank and new line characters can be ignored. These stream of tokens will be given to syntax analyzer.

Input Buffering

L→R 1 chr at time

- The LA scans the input string from left to right one character at a time. It uses two pointers begin-ptr (bp) and forward-ptr (fp) to keep track of the portion of the input scanned.
- Initially both the pointers point to the first character of the input string as shown below:-

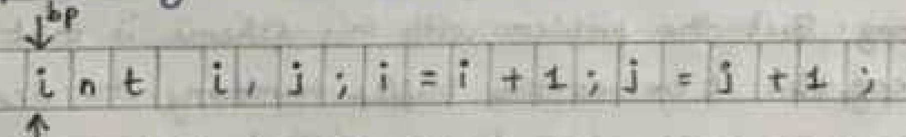


fig: initial configuration.

- The forward-ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered it indicates end of lexeme. In above eg as soon as forward-ptr (fp) encounters a blank space the lexeme "int" is identified.
- The fp will be moved ahead at white space. When fp encounters white space it ignore and moves ahead.

i/p chr read secondary storage (costly)
hence buffering.
blk of data is read & then scanned.

- Then both the begin-ptr (bp) and forward-ptr (fp) is set at next token i

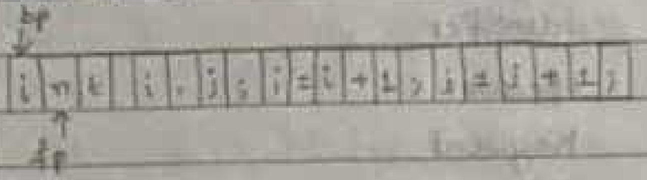
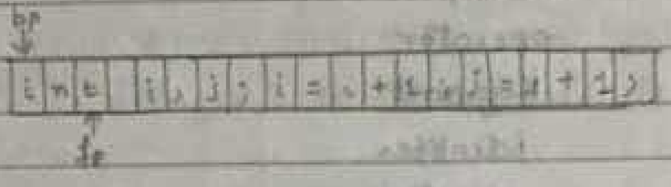


fig:- Input buffering



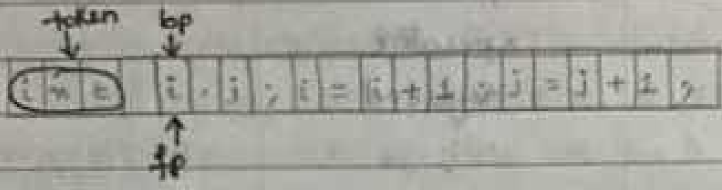
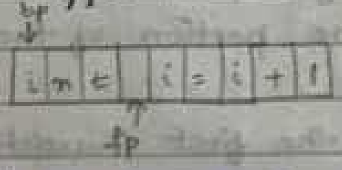


fig:- Input buffering.

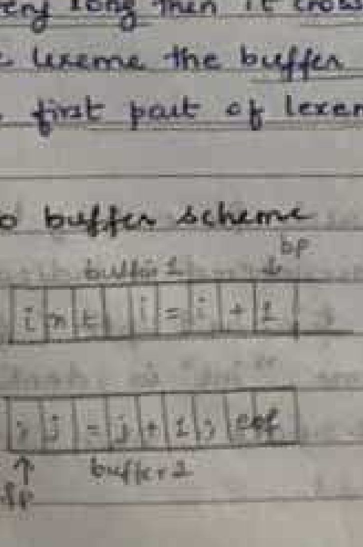
- The input character is thus read from secondary storage. But reading in this way from secondary storage is costly. Hence buffering techniques is used.
- A block of data is first read into a buffer, and then scanned by LA. There are two methods used in this context : one buffer scheme and two buffer scheme.

1). One buffer scheme : problm; if lexeme is very long, it crosses buff bound.



- In this one buffer scheme, only one buffer is used to store the input string. But the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first part of lexeme.

2) Two buffer scheme 1st & 2nd are scanned alternately.
end of current buffr is reached - other is filled



- To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. The first buffer and second buffer are scanned alternately.
- when end of current buffer is reached the other buffer is filled.
- The only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning i/p can not be scanned complitely.

if l(lexeme) > l(buffer) i/p cannot be scanned completely

- Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme.
- As soon as blank character is recognized, the string between bp and fp is identified as corresponding token.
- To identify the boundary of first buffer end of buffer character should be placed at the end of first buffer.

eof buffer char placed

- Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer.
- when fp encounters first eof, then one can recognize end of first buffer and hence filling up of 2nd buffer is started.
- In the same way when second eof is obtained then it indicates end of 2nd buffer.
- Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified.
- This eof character introduced at the end is called Sentinel which is used to identify the end of buffer.

eof is sentinel

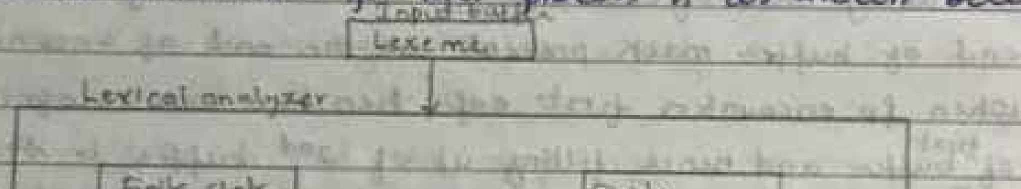
Code for input buffering

```
if (fp==eof(buff1))  /* encounters end of first buffer*/
{
  /* Refill buffer 2*/
   fp++;
 }
 else if (fp==eof(buff2)  /* encounters end of 2nd buffer*/
 {  /* Refill buffer1 */
     fp++;
 }
 else if (fp==eof(input))
  return ;   /* terminate scanning */
```

```
else
  fp++;
    /* still remaining input has to scanned */
```

Block schematic of LA :-

- LA is a process of recognizing tokens from input source pgm.
- Now the question is how does lexical analyzer recognize tokens, from given source pgm ?
- well, LA stores the input in a buffer. It builds the regular expressions for correspondiry tokens.
- from these regular expressions, finite automata is built.
- when lexeme matches with the pattern generated by finite automata, the specific token gets recognized.
- The block schematic for this process is as shown below.



- While constructing the LA we first design the regular expression for recognizing the corresponding token. A diagram resembling the flowchart is built. such a diagram is called transition diagram.
+ The transition diagram elaborates the actions to be taken while recognizing the token.
- The lexeme is stored in an input buffer. The forward pointer scans the input character-by-character moving from left to right.
- The transition diagram is used to keep track of the infm

about characters that are seen as the forward ptr scans the input.

- Positions in a transition diagram are called states and those are drawn by circles and the edges in the diagram represent the transitions from one state to another.
- There is a special state called start state, which denotes the starting of transition diagram

buffer
↓
build RE for token
↓
FA is built
when lexeme matches patten(RA) → token is return.