Input Buffering :-

→ Deals about speedup the reading process.

Reading Tokens :-

look one or more characters beyond the next Lexme before be sure that the right lexme are seen.

Ex :- look atleast one additional characters before before deciding an Identifier.

Other issues in Reading input :-

* Single character operator like -, =, < could be the begning of a two character operators. like,
  !>, ==, <>, >=.

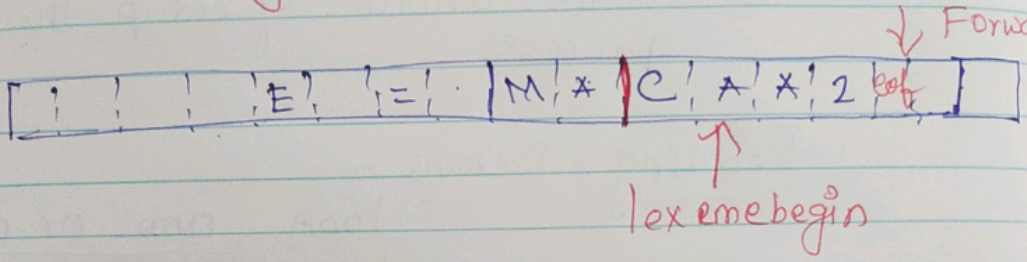
* Solution :- Two buffer scheme to Handle large lookahead safely.

Buffer Pair :-

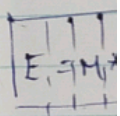
* Reduce the amount of overhead to process a single input character

* a two buffer scheme that are loaded with input alternatively.

Using a pair of Input Buffers.



* Let buffer size is N, Usually size of a disk block e.g 4096 bytes.
* Instead of one read system command to N characters, into a buffer, instead of one system call per character.
* EOF :- * if N characters remain in the input file, then a special character represented by eof.
  ↓
  marks the end of the source program
  * Eof is a character, different from character present in source program.



Pointers used in Buffer :-

① Lexeme Begin :- * marks the beginning of the current lexeme
* who extent we are attempting to determine.

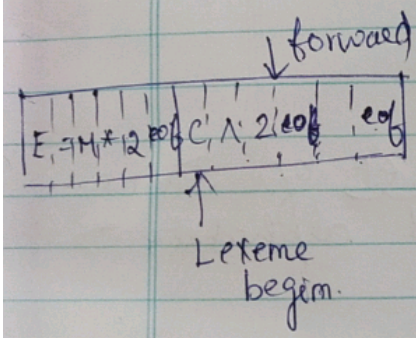
② forward :- scans until a pattern match is found.

* Once the next lexeme is determined, forward is set to the character at its right end.

* Then record the lexeme.

* Then lexeme begin is set to the next character immediately after the lexeme is found.

Sentinels :-



Check, each time when advance the forward pointer, that we have not moved off one of the buffers.

if so load the other buffer.

Specifications of Tokens :- Regular expression are an important notation for specifying lexeme patterns

Strings & Languages :- An alphabet is a any finit set of symbols

Examples of Symbols :-

* Letters
* digits
* punctuations.
* {0, 1} → binary alphabet.
* ASCII
* UNICODE

String:- String over alphabet is a finite sequence of symbols drawn from that alphabet.

Sentence & word are same, used instead of string.

Let s is a string

* |s| ⇒ length of the string.

if s = "banana" then
|s| = 6 //

* ∈ → empty string (or) ϕ.

|∈| = 0

Language:- any countable set of string over some fixed alphabet.

Terms for Parts of Strings:

prefix:- string obtained by removing zero or more symbols from the end of s.

Example:- s = "banana"

prefix of s:-
ban, banana, ∈

Sufix :- a string obtained by removing zero or more symbols from the begning of s.

Sufix of s :- 'nana, banana, ϵ, .

Suffix :- s_1 is obtained by deleting any prefix and any sufix from s

Sufix of s :- banana, na, ϵ

Proper prefix, Sufix and Substring of s :- s (string itself), ϵ are not part of prefix, suffix, substring.

Subsequence :- s is any string formed by deleting zero or more not necessarily consecutive positions of s

Subsequences of s :- baan

Concatenation :-

Let x, y are two strings then xy is the concatenation.

Example :-

x = "dog", y = "house"

xy = "doghouse".

Exponentiation :-

- $s^0 = \epsilon$
- if $i > 0$, s^i to be $s^{i-1}s$.
- $\epsilon s = s$
- $s^1 = s$
- $s^2 = ss$, $s^3 = sss$.

Operations on Languages :-

Most important operations on languages :-

(i) Union $\cup$
(ii) Concatenation
(iii) closure. $*$, $+$

$*$ ↙ Kleen closure
$+$ ↓ positive closure

Operation	Definition and Notation.
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ (or) } s \text{ is in } M\}$
Concatenation of L and M.	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleen closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive closure of L.	$L^+ = \bigcup_{i=1}^{\infty} L^i$ { L^0 not included }

Regular Expressions :-

Regular expressions describing all the languages that can be build from the operators ($\cup, *, +$, .) applied to symbols of some alphabets.

letter → any letter or underscore
digits → any digit

Identifiers :- (Description):-

letter_ (letter_ | digit)*

(Starting with letter then followed by letter (zero or more times).

Rules that define Regular expressions :-

* Regular expression r
* Regular expression denoted by a language L(r).
* Regular expression r is recursively build from smaller regular expression
* L(r) is also recursively defined from the r.

Two Rules :-

(1) ∈ is a regular expression.
L(∈) is {∈}. → language whose sole member is the empty string.

(2) If a is a symbol in Σ.

(Σ ↓ set of alphabet)

$L(a) = \{a\}$ the language with one string.

Use italics for symbols
boldface for their corresponding regular expression.

Induction :-

* Four parts of induction
* larger regular expressions are build from smaller ones.

Let r, s are regular expressions denoting languages $L(r)$, $L(s)$,

(i) $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$

(ii) $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$

(iii) $(r)^*$ is a regular expression denoting $(L(r))^*$

(iv) (r) is a regular expression denoting $L(r)$

↳ add additional pair of parantheses around expressions without changing the language they denote.

NOTE:-

(a) The Unary operator * has highest precedence and is left associative

(b) Concatenation has the second highest precedence and left associative

(c) | has the lowest precedence and is left associative.

or

Unnecessary paranthesis must be removed.

RE (a) | ((b)*(c)) is replaced by a|b*c.

Denotes set of string, single a or zero or more b followed by one c.

Examples:

(i) a|b denotes the language {a, b}

(ii) (a|b)(a|b) denotes {aa, ab, ba, bb}

String of length two over Σ.

(or)

aa|bb|ba|bb. denotes the same language.

(iii) a^* → all strings of zero or more a's.
(i.e) $\{\epsilon, a, aa, aaa \ldots\}$

(iv) $(a|b)^*$ set of string consisting of zero or more instances of a and b.

$\{\epsilon, a, b, aa, ab, bb, ba, aaa \ldots\}$

(or)

$(a^* b^*)^*$

(v) $a|a^*b$ denotes

$\{a, b, ab, aab, aaab \ldots\}$

Zero or more a and ending in b.

NOTE:-

* A language that can be defined by
* $r = s$ RE r and s are equivalent

Algebric Laws for RE:-

LAW	Describtions.
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative

~~r(st)|~~

$r(st) = (rs)t$ Concatenation is associative

$r(s|t) = rs|rt$; Concatenation is distributive over |.
$(s|t)r = sr|tr$

$\epsilon r = r\epsilon = r$ ϵ is identity for Concatenation

$r^* = (r|\epsilon)^*$ ϵ is guaranteed in closure.

$r^{**} = r^*$ * is idempotent

Example:-

(1) Regular definition for the language of C identifiers.

letter → |A|B|C|D| . . |Z|-
digits → 0|1|2 . . |9|

id → letter_ (letter_|digit)*