Lexical Analysis

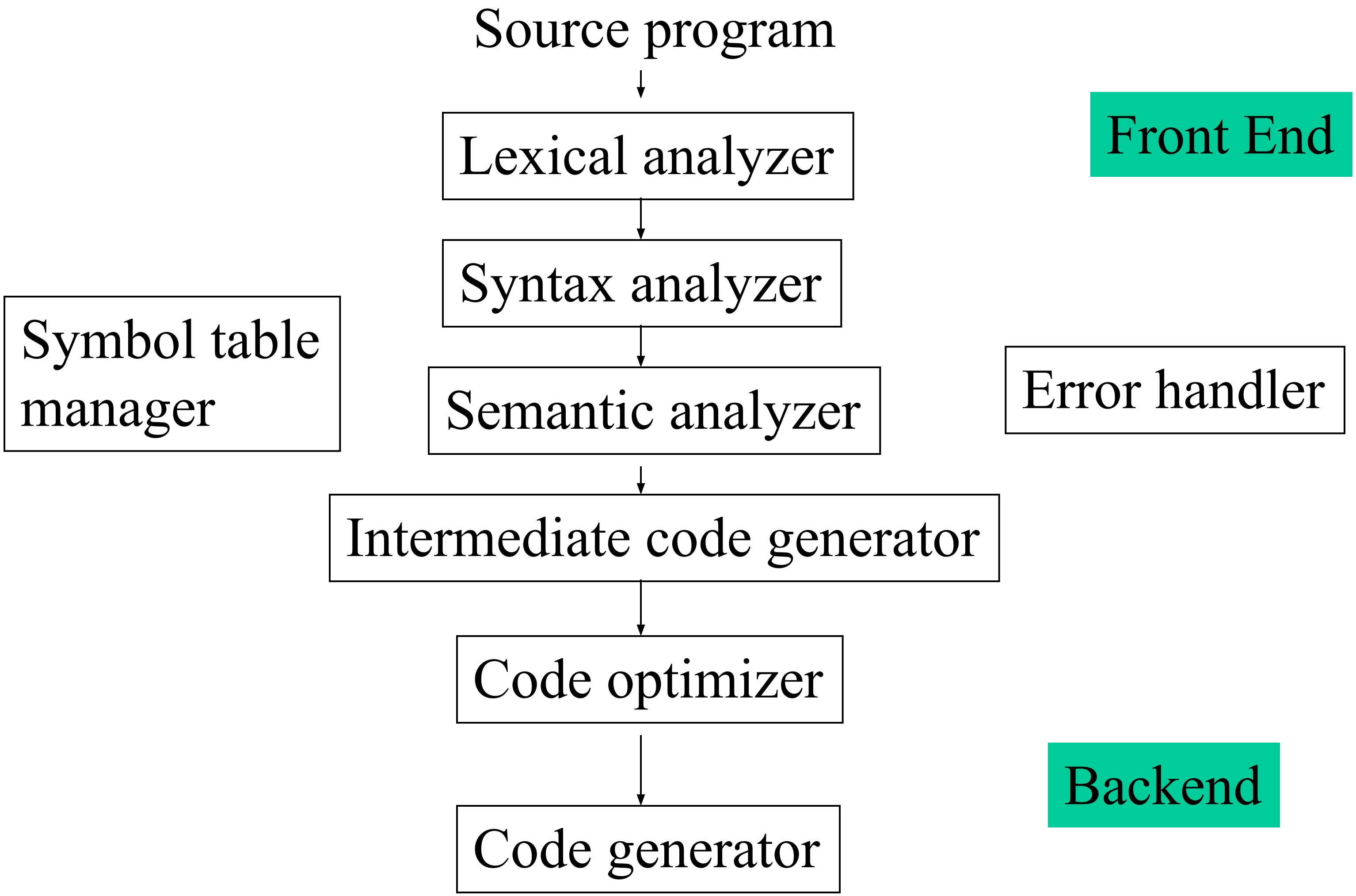
Chapter 2 – part 1

Dr G Sudha Sadasivam
Prof & Head,CSE, PSGCT

Agenda

- Role of LA
- Terms related
- Scanner + parser
- RE and regular definitions
- Input buffering – single and dual buffers

Review: Compiler Phases:



Source program

Lexical analyzer

Syntax analyzer

Semantic analyzer

Intermediate code generator

Code optimizer

Code generator

Symbol table manager

Error handler

Front End

Backend

Lexical Analysis

- Lexical analyzer: reads input characters and produces a sequence of tokens as output (nexttoken()).
  - Trying to understand each element in a program.
  - *Token*: a group of characters having a collective meaning.

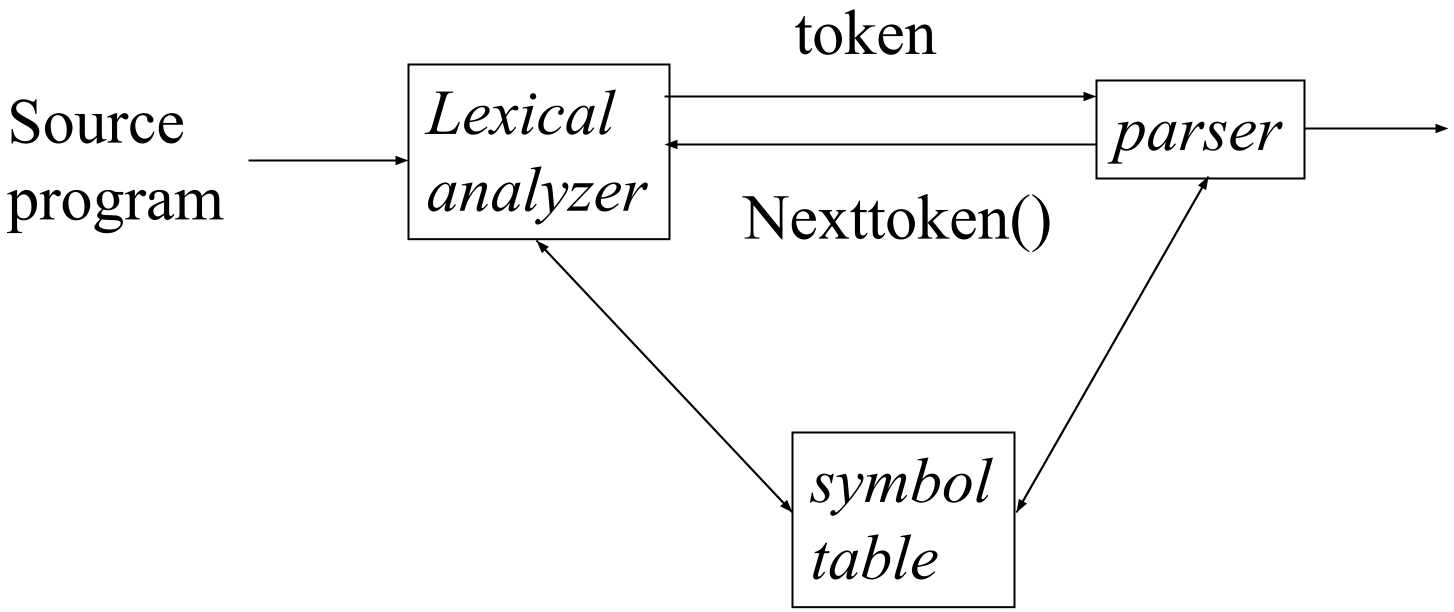
    const pi = 3.14159;

    Token 1: (const, -)
    Token 2: (identifier, ‘pi’)
    Token 3: (=, -)
    Token 4: (realnumber, 3.14159)
    Token 5: (;, -)

Role of Lexical Analyser

- Read input characters
- To group them into lexemes
- Produce as output a sequence of tokens
  - input for the syntactical analyzer
- Interact with the symbol table
  - Insert identifiers
- to strip out
  - Comments
  - whitespaces: blank, newline, tab, …
  - other separators; compaction of consecutive white spaces into one
- to correlate error messages generated by the compiler with the source program
  - to keep track of the number of newlines seen
  - to associate a line number with each error message

Interaction of Lexical analyzer with parser



- Some terminology:
  - *Token*: a group of characters having a collective meaning. A *lexeme* is a particular instant of a token.
    - E.g. token: identifier, lexeme: pi, etc.
  - *pattern*: the rule describing how a token can be formed.
    - E.g: identifier: ([a-z]||[A-Z]) ([a-z]||[A-Z]||[0-9])*

- Lexical analyzer does not have to be an individual phase. But having a separate phase simplifies the design and improves the efficiency and portability.

Why to separate scanner and parser phases?

- **Simplicity of design**
  - Separation of lexical from syntactical analysis -> *simplify* at least one of the tasks

    e.g. parser dealing with white spaces -> complex
  - Cleaner overall language *design*
- **Improved compiler efficiency**
  - Liberty to apply *specialized techniques* that serves only lexical tasks, not the whole parsing
  - *Speedup* reading input characters using specialized buffering techniques
- **Enhanced compiler portability**
  - Input device peculiarities are restricted to the lexical analyzer (poratble)

Lexeme, pattern, token

- **Lexeme**
  - a sequence of characters in the source program matching a pattern for a token eg… (total = a+ b) …. Total,=,a,+,b
- **Pattern**
  - description of the form that the lexeme of a token may take

  ◦ e.g. Id = {[a-z][a-z,0-9]*}
- **Token - pair** of:

  ◦ token name – abstract symbol representing a kind of lexical unit <id,ptr> -- attb – value pair

- Two issues in lexical analysis.
  - How to specify tokens (patterns)?
  - How to recognize the tokens giving a token specification (how to implement the nexttoken() routine)?
- How to specify tokens:
  - all the basic elements in a language must be tokens so that they can be recognized.

```
main() {
    int i, j;
    for (I=0; I<50; I++) {
        printf("I = %d", I);
    }
}
```

    - Token types: constant, identifier, reserved word, operator and misc. symbol.
  - Tokens are specified by **regular expressions**.

• Some definitions

- *alphabet* : a finite set of symbols. E.g. {a, b, c}
- A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet (sometimes a string is also called a sentence or a word).
- A *language* is a set of strings over an alphabet.
- Operation on languages (a set):
  - union of L and M, L U M = {s|s is in L or s is in M}
  - concatenation of L and M

    LM = {st | s is in L and t is in M}
  - Kleene closure of L, $$L^* = \bigcup_{i=0}^{\infty} L^i$$
  - Positive closure of L, $$L^+ = \bigcup_{i=1}^{\infty} L^i$$
- Example:
  - L={aa, bb, cc}, M = {abc}

- **Formal definition of Regular expression:f**
  - Given an alphabet $\sum$,
  - (1) ε is a regular expression that denote { ε }, the set that contains the empty string.
  - (2) For each $a \in \sum$, a is a regular expression denote {a}, the set containing the string a.
  - (3) r and s are regular expressions denoting the language (set) L(r) and L(s). Then
    - (r) | (s) is a regular expression denoting L(r) U L(s)
    - (r) (s) is a regular expression denoting L(r) L (s)
    - (r)* is a regular expression denoting (L (r)) *

  - Regular expression is defined together with the language it denotes.

- **Examples**:
  - let $\sum = \{a,b\}$

    a | b

    (a | b) (a | b)

    a *

    (a | b)*

    a | a*b

  - We assume that ‘*’ has the highest precedence and is left associative. Concatenation has second highest precedence and is left associative and ‘|’ has the lowest precedence and is left associative
    - (a) | ((b)*(c)) = a | b*c

- **Regular definition.**
  - gives names to regular expressions to construct more complicate regular expressions.

    d1 -> r1

    d2 ->r2

    …

    dn ->rn
  - example:

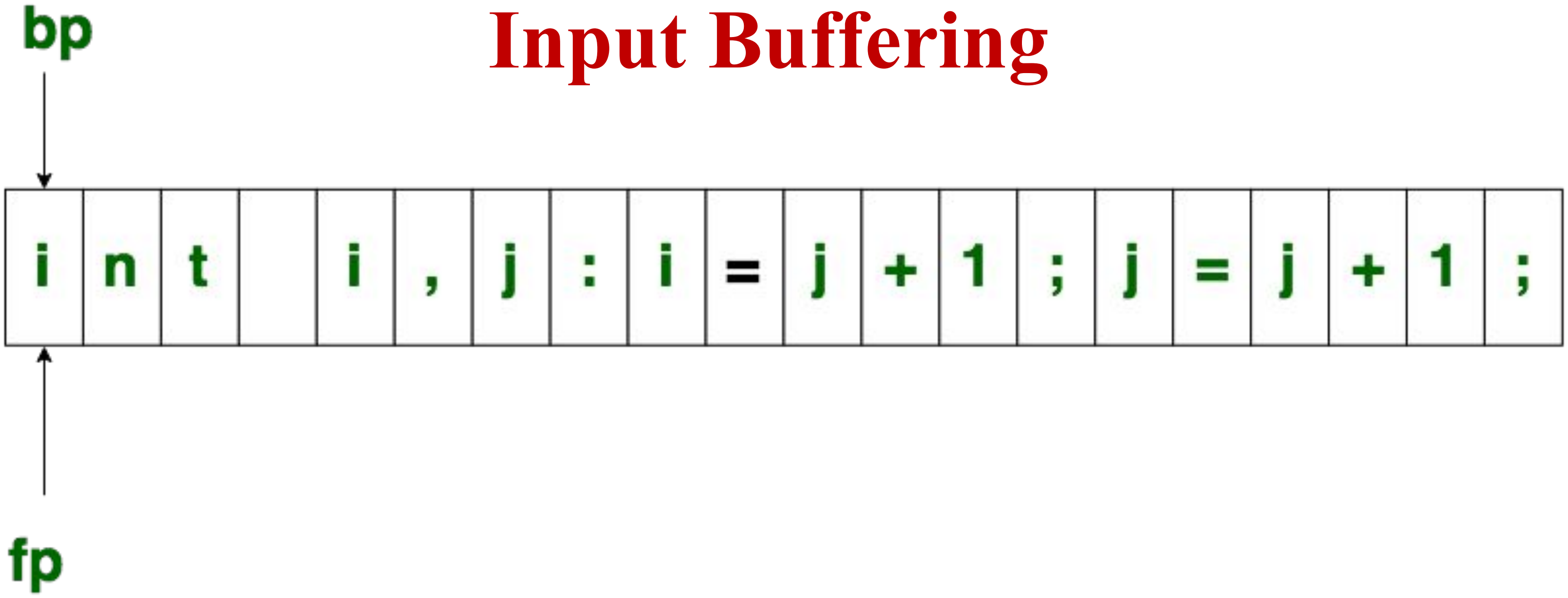
    letter -> A | B | C | … | Z | a | b | …. | z

    digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

    identifier -> letter (letter | digit) *

    - more examples: integer constant, string constants, reserved words, operator, real constant.

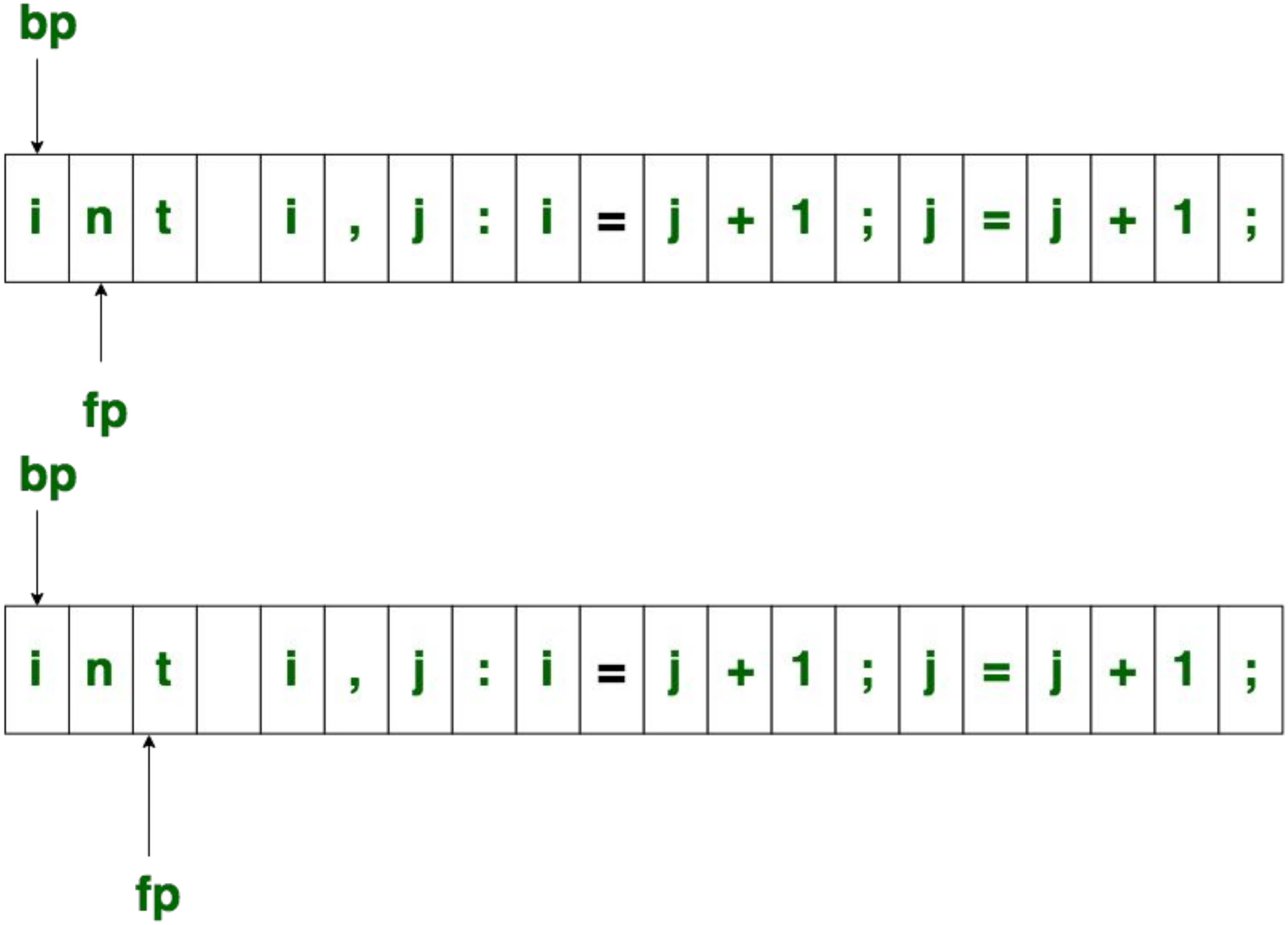
Input Buffering



i	n	t		i	,	j	:	i	=	j	+	1	;	j	=	j	+	1	;

Initial Configuration

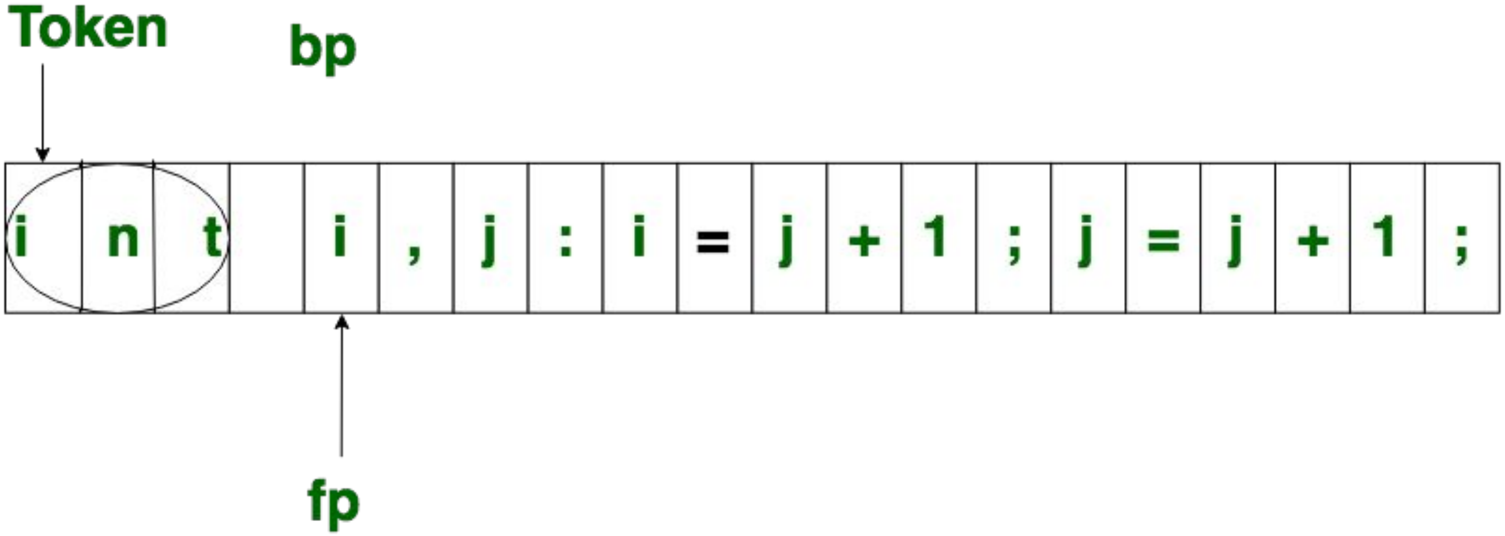
- The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward to keep track of the pointer of the input scanned.
- Initially both the pointers point to the first character of the input string as shown below



bp
i n t   i , j : i = j + 1 ; j = j + 1 ;
fp
bp
i n t   i , j : i = j + 1 ; j = j + 1 ;
fp

Input Buffering

- fp moves until a blank is encountered, it comes back and identifies int as a token.

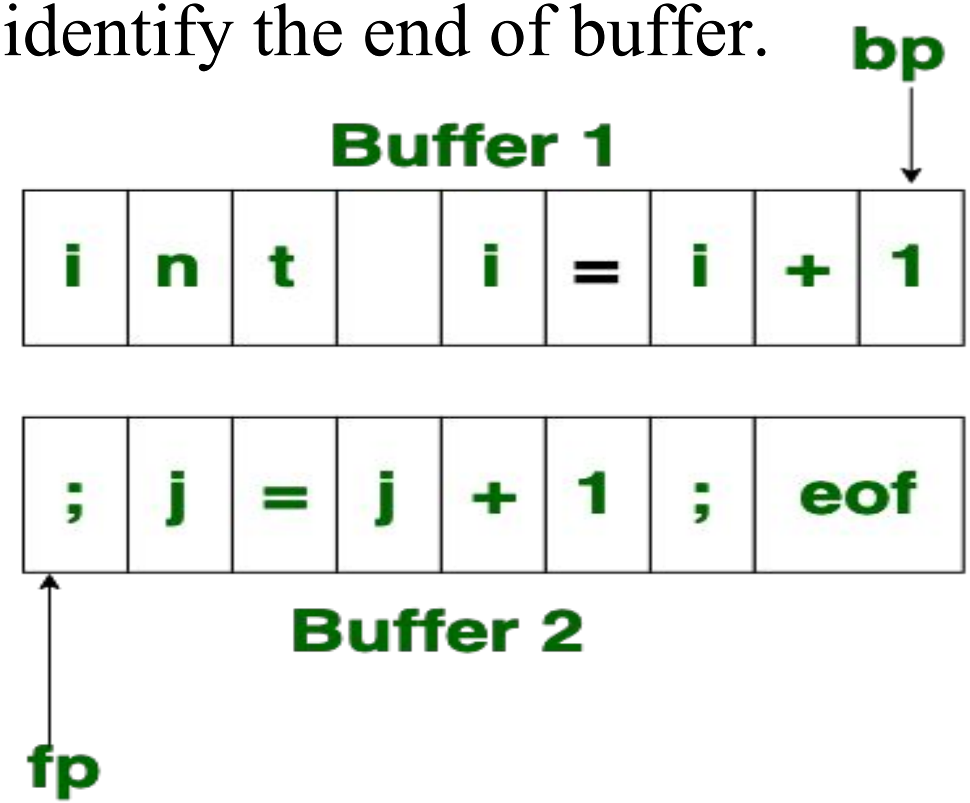


Input buffering

- then both the begin ptr(bp) and forward ptr(fp) are set at next token.
- **Single buffer**: In this 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 of lexeme.

Two buffer scheme

- the first buffer and second buffer are scanned alternately
- to identify, the boundary of first / second buffer end of buffer character should be placed at the end first / second buffer.
- **eof** character introduced at the end is calling **Sentinel** which is used to identify the end of buffer.


bp
Buffer 1
i
n
t
i
=
i
+
1
;
j
=
j
+
1
;
eof
Buffer 2
fp