Compiler Design Assignment - CSE V Semester

Unit I

1. Make an NFA for the regular expression (a|b)*abb and convert it intoan equivalent DFA.
2. Describe the major data structures used in a compiler and their roles.
3. Explain in brief

   (i) The role of the lexical analyzer.

   (ii) Lexeme, Token & Pattern.

1. How are tokens specified and recognized in lexical analysis?
2. What is bootstrapping in compiler design? Provide an example.

Compiler Design Assignment - CSE V Semester

Unit - II

1. Differentiate between top-down parsing and bottom-up parsing with suitable examples.
2. Construct LL(1) parsing table for the following grammar :

   SaBDh BcC CbC|

   DEF Eg| Ff|

1. Consider the grammar:

   S AaAb| BbBa A B

   Show that the grammar is LL (1) but not SLR.

1. Construct LALR parsing table for the following set of productions.

   SCC CcC Cd

1. Define syntax-directed definitions (SDDs) and their role in compilers.

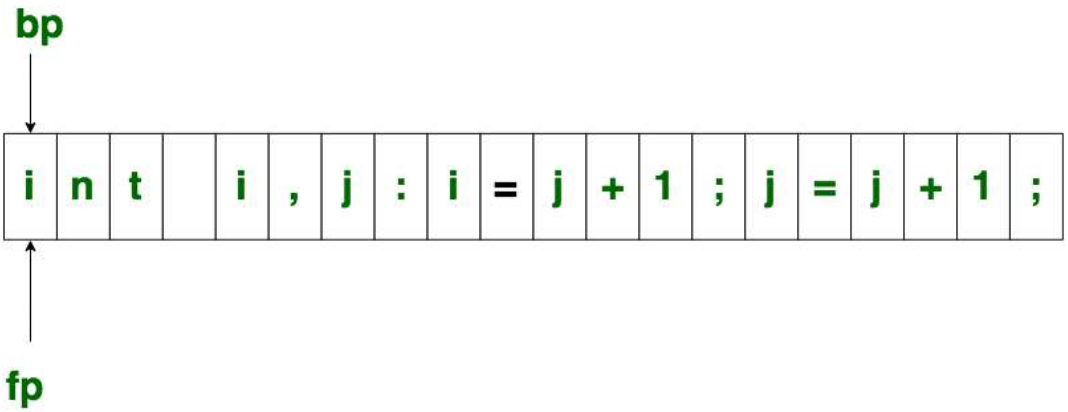
The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward ptr(**fp**) to keep track of the pointer of the input scanned.

Input buffering is an important concept in compiler design that refers to the way in which the compiler reads input from the source code. In many cases, the compiler reads input one character at a time, which can be a slow and inefficient process. Input buffering is a technique that allows the compiler to read input in larger chunks, which can improve performance and reduce overhead.

- The basic idea behind input buffering is to read a block of input from the source code into a buffer, and then process that buffer before reading the next block. The size of the buffer can vary depending on the specific needs of the compiler and the characteristics of the source code being compiled. For example, a compiler for a high-level programming language may use a larger buffer than a compiler for a low-level language, since high-level languages tend to have longer lines of code.
- One of the main advantages of input buffering is that it can reduce the number of system calls required to read input from the source code. Since each system call carries some overhead, reducing the number of calls can improve performance. Additionally, input buffering can simplify the design of the compiler by reducing the amount of code required to manage input.

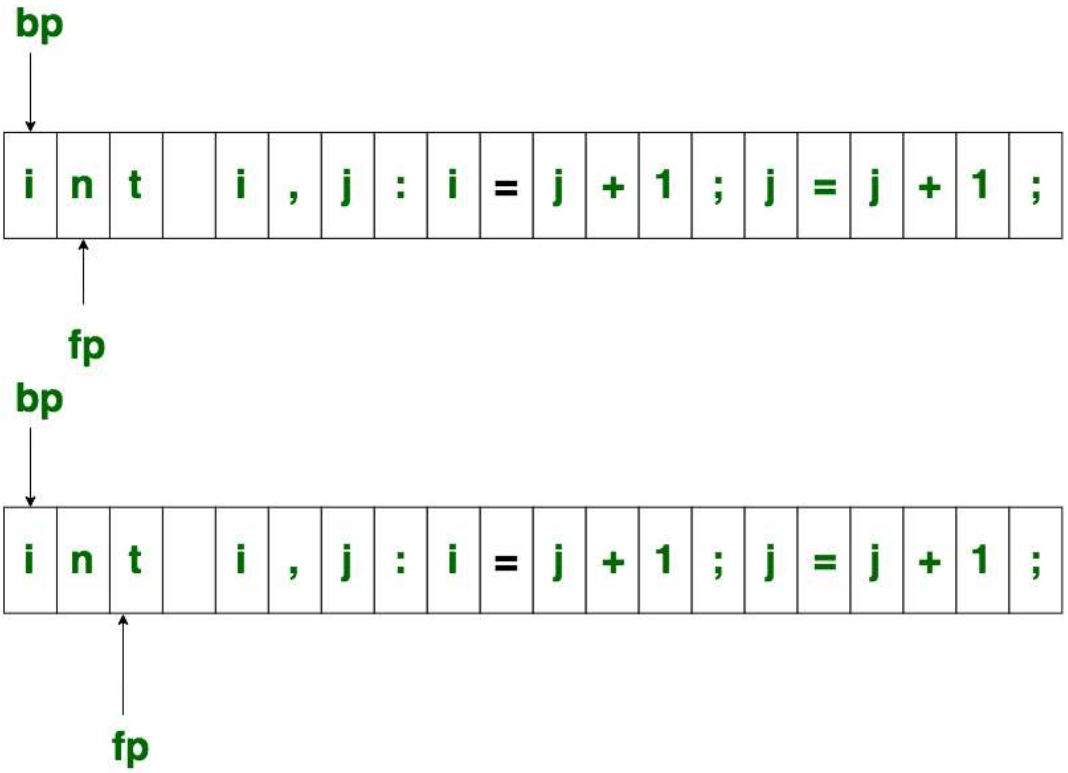
However, there are also some potential disadvantages to input buffering. For example, if the size of the buffer is too large, it may consume too much memory, leading to slower performance or even crashes. Additionally, if the buffer is not properly managed, it can lead to errors in the output of the compiler.

Overall, input buffering is an important technique in compiler design that can help improve performance and reduce overhead. However, it must be used carefully and appropriately to avoid potential problems.



Initial Configuration

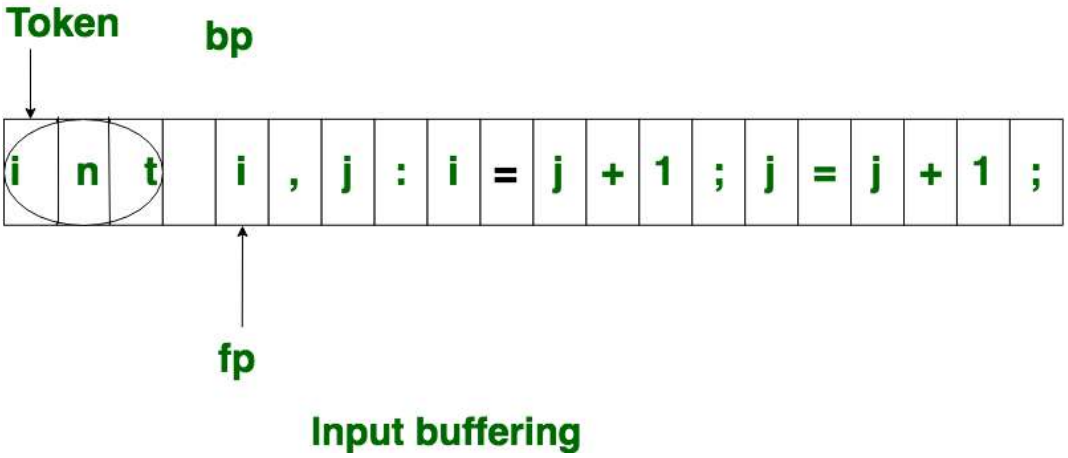
Initially both the pointers point to the first character of the input string as shown below



Input Buffering

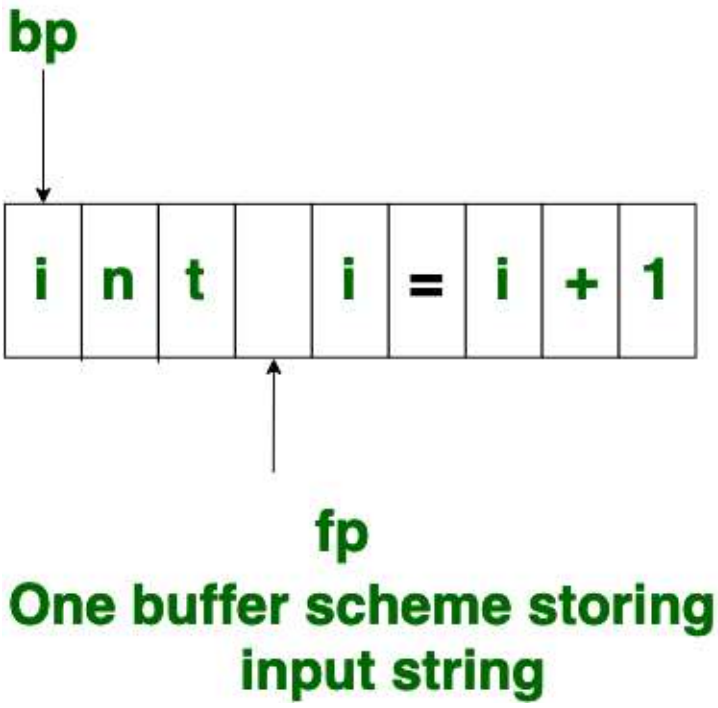
The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme "int" is identified. The fp will be moved ahead at white space, when

fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token. The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A block of data is first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below.



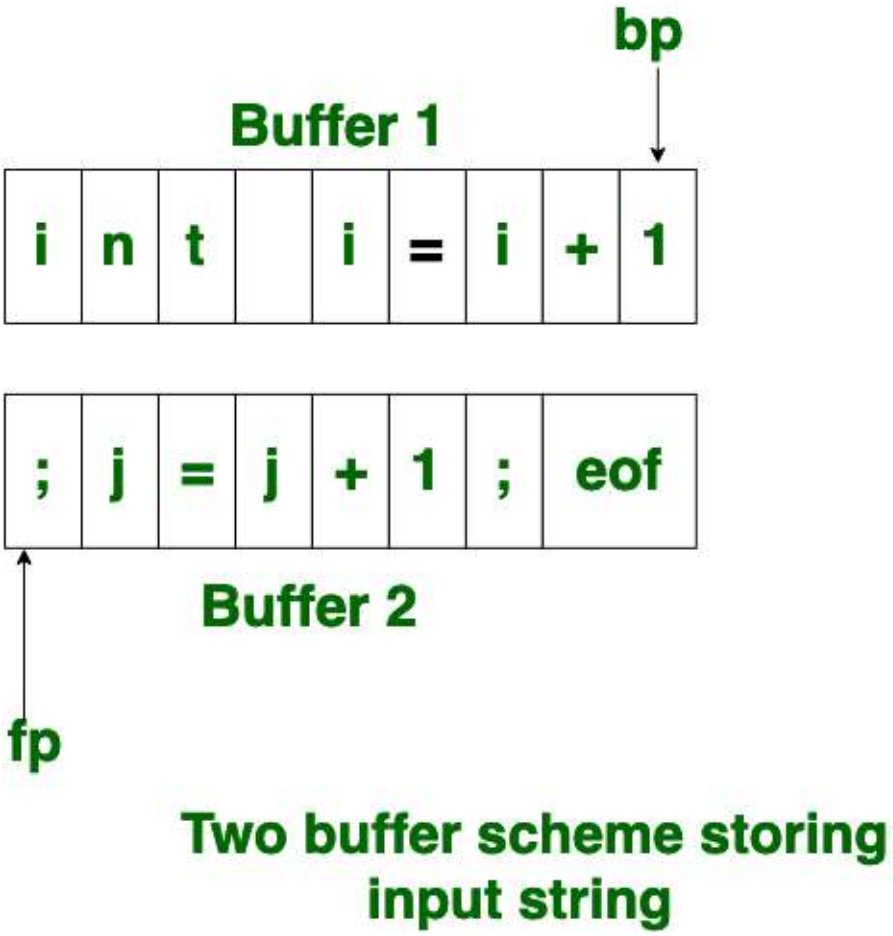
Input buffering

One Buffer Scheme: 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.



One buffer scheme storing input string

Two Buffer Scheme: 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 input 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 first buffer. 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 second buffer is started. in the same way when second **eof** is obtained then it indicates of second 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 calling **Sentinel** which is used to identify the end of buffer.



Two buffer scheme storing input string

Advantages:

Input buffering can reduce the number of system calls required to read input from the source code, which can improve performance.
Input buffering can simplify the design of the compiler by reducing the amount of code required to manage input.

Disadvantages:

If the size of the buffer is too large, it may consume too much memory, leading to slower performance or even crashes.
If the buffer is not properly managed, it can lead to errors in the output of the compiler.
Overall, the advantages of input buffering generally outweigh the disadvantages when used appropriately, as it can improve performance and simplify the compiler design.

Compiler Design Syllabus - CSE V Semester

Unit-1

Introduction of Compiler, Major data Structure in compiler, BOOT Strapping & Porting, Compiler structure: analysis-synthesis model of compilation, various phases of a compiler, Lexical analysis: Input buffering, Specification & Recognition of Tokens, LEX.

Unit-2

Syntax analysis: CFGs, Top down parsing, Brute force approach, recursive descent parsing, Transformation on the grammars, predictive parsing, bottom up parsing, operator precedence parsing, LR parsers (SLR,LALR, LR),Parser generation. Syntax directed definitions: Construction of Syntax trees, Bottom up evaluation of S-attributed definition, L-attribute definition, Top down translation, Bottom Up evaluation of inherited attributes Recursive Evaluation, Analysis of Syntax directed definition.

Unit -3

Type checking: type system, specification of simple type checker, equivalence of expression, types, type conversion, overloading of functions and operations, polymorphic functions. Run time Environment: storage organization, Storage allocation strategies, parameter passing, Dynamic storage allocation , Symbol table.

Unit -4

Intermediate code generation: Declarations, Assignment statements, Boolean expressions, Case statements, Back patching, Procedure calls Code Generation: Issues in the design of code generator, Basic block and flow graphs, Register allocation and assignment, DAG representation of basic blocks, peephole optimization, generating code from DAG.

Unit-5

Introduction to Code optimization: sources of optimization of basic blocks, loops in flow graphs, dead code elimination, loop optimization, Introduction to global data flow analysis, Code improving transformations ,Data flow analysis of structure flow graph Symbolic debugging of optimized code.