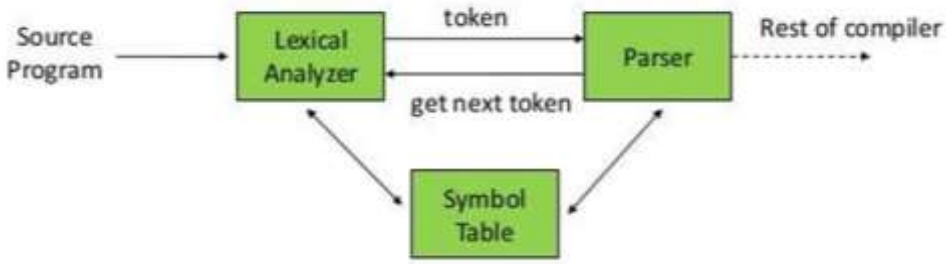
Lexical Analysis

Explain structure of Lexical Analyzer.

Structure of Lexical Analyzer

- Lexical Analysis is the initial phase of a compiler, also referred to as a scanner. Its purpose is to convert high-level input programs into a sequence of tokens.
- Lexical Analysis can be implemented using Deterministic Finite Automata (DFA).
- The output of the Lexical Analyzer is a sequence of tokens sent to the parser for syntax analysis.



What is a Token?

- A lexical token is a sequence of characters treated as a unit in the grammar of programming languages.
- Examples include type tokens (id, number, real), punctuation tokens (IF, void, return), and alphabetic tokens (keywords).

Lexeme:

- A sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters comprising a single token.
- Examples include “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;”.

How Lexical Analyzer Works?

1. **Input Preprocessing:** Clean up the input text by removing comments, whitespace, and non-essential characters.
2. **Tokenization:** Break the input text into a sequence of tokens by matching characters against predefined patterns or regular expressions.
3. **Token Classification:** Determine the type of each token such as keywords, identifiers, operators, and punctuation symbols.
4. **Token Validation:** Check each token for validity according to the rules of the programming language.
5. **Output Generation:** Generate the output of the lexical analysis process, typically a list of tokens for further processing.

Error Detection:

- The lexical analyzer identifies errors using automation machine and grammar of the given language, providing row and column numbers of the error.

Token Sequence Example:

- For a statement like **a = b + c;**, the token sequence is **id=id+id;**, with each **id** referencing its variable in the symbol table.

Advantages:

- **Simplifies Parsing:** Breaking down the source code into tokens makes it easier for computers to understand and work with the code.
- **Error Detection:** Detects lexical errors early in the compilation process, improving overall efficiency.
- **Efficiency:** Once source code is tokenized, subsequent phases of compilation or interpretation operate more efficiently.

Disadvantages:

- **Limited Context:** Lexical analysis operates on individual tokens, potentially leading to ambiguity in complex languages.
- **Overhead:** Adds computational overhead to the compilation process.

- **Debugging Challenges:** Lexical errors may not always provide clear indications of their origins, posing challenges during debugging.

How error recovery is handled in lexical analysis?

Error Recovery in Lexical Analysis

1. **Error Recovery Technique:**
    - When the lexical analyzer encounters a situation where none of the patterns for tokens matches any prefix of the remaining input, error recovery becomes necessary.
    - The simplest recovery strategy is called "panic mode" recovery. It involves deleting successive characters from the remaining input until the lexical analyzer can identify a well-formed token at the beginning of what remains.
2. **Error-Recovery Actions:**
    - Transpose of two adjacent characters.
    - Insert a missing character into the remaining input.
    - Replace a character with another character.
    - Delete one character from the remaining input.
3. **Handling Errors in Lexical Analysis:**
    - **Leaving the Statement:** The lexical analyzer may identify the error, correct it internally, and then proceed to call the parser for syntax analysis.
    - **No Matching Pattern:** In the event of no matching pattern, the simplest recovery strategy, "panic mode recovery," is employed. This involves not processing the remaining characters. However, this approach is not ideal as tokens still need to be sent to the parser for syntax analysis.
4. **Other Possible Recovery Methods:**
    - Inserting a missing character.
    - Replacing an incorrect character with the correct one.
    - Transposing (changing the place of) adjacent characters.

- Deleting an extra character.

5. Goal of Error Recovery:

- The primary goal of error recovery in lexical analysis is to recover from errors and continue the lexical analysis process so that tokens can be passed to the parser for further processing.
- Simply abandoning the input is not a viable solution as it would hinder the parser from receiving the necessary tokens.

What are tokens? Explain specification & recognition of tokens.

Tokens: Specification & Recognition

- In programming languages, tokens represent various lexical units such as keywords, constants, identifiers, strings, numbers, operators, and punctuation symbols.
- For example, in C language, the line **int value = 100;** contains tokens like **int** (keyword), **value** (identifier), **=** (operator), **100** (constant), and **;** (symbol).

Specifications of Tokens:

1. **Alphabets & Strings:**
    - Alphabets are finite sets of symbols used in languages.
    - Strings are finite sequences of alphabets.
2. **Special Symbols:**

    Programming languages consist of various symbols like arithmetic symbols, punctuation, assignment symbols, comparison symbols, and preprocessor directives.
3. **Language:**

    A language is a finite set of strings over a finite set of alphabets.
4. **Longest Match Rule:**

    The lexical analyzer follows the longest match rule, wherein it identifies tokens until it encounters whitespace, operator symbols, or special symbols.
5. **Operations & Notations:**

    Operations on languages include union, concatenation, and Kleene closure, denoted by regular expressions.
6. **Finite Automata:**

Finite automata are state machines that process input symbols and transition between states. They consist of states, input symbols, start state, final states, and a transition function.

Some key operations on regular expressions include:

- Union: (r)|(s) is a regular expression denoting the language L(r) ∪ L(s).
- Concatenation: (r)(s) is a regular expression denoting the language L(r)L(s).
- Kleene closure: (r)* is a regular expression denoting the language (L(r))*.

The precedence and associativity of these operations are as follows:

- * has the highest precedence.
- Concatenation . has the second highest precedence.
- | (pipe) has the lowest precedence.
- All operators are left-associative.

Recognition of Tokens:

1. **Finite Automata:**
    - Recognition of tokens is typically performed using finite automata, which represent the actions of the lexical analyzer.
    - The lexical analyzer reads input from the input buffer, starting from the '***lexemeBegin***' pointer and advancing using a forward pointer.
    - It calculates the set of states at each point based on the input symbols.
    - If no next state is available for an input symbol, the analyzer determines the longest prefix that matches a pattern.
    - This process continues until one or more accepting states are reached.
2. **Decision Making:**

If multiple accepting states are reached, the pattern that appears earliest in the list of the Lex program is chosen.

Create a lexical analyzer with LEX as tool.

To create a lexical analyzer using LEX as a tool, you can follow these steps:

1. **Write the Lex Program**: Start by writing the Lex program that describes the lexical analyzer you want to generate. Define the regular expressions for the tokens you want to recognize and specify the corresponding actions to be taken when a token is matched.
2. **Compile the Lex Program**: Use the Lex compiler to transform your Lex program into a C program. The Lex compiler will generate a C file, typically named lex.yy.c, based on your Lex program.
3. **Compile the C Program**: Compile the generated C program using a C compiler like GCC. This will produce an executable file that represents your lexical analyzer.
4. **Test the Lexical Analyzer**: Run the executable file on input text to test the lexical analyzer. Ensure that it correctly identifies and processes the tokens according to the rules defined in your Lex program.

Here is a simple example of a Lex program that recognizes integers and identifiers:

```
lex
%{
#include <stdio.h>
%}

DIGIT [0-9]
ID [a-zA-Z][a-zA-Z0-9]*

%%

{DIGIT}+    { printf("Integer: %s\n", yytext); }
{ID}        { printf("Identifier: %s\n", yytext); }
.           { /* Ignore other characters */ }

%%

int main() {
    yylex();
    return 0;
}
```

In this example:

- %{ ... %} is used for including C code.
- DIGIT and ID are regular expressions for integers and identifiers.
- Rules are defined to match integers and identifiers and print them.
- The . rule ignores any other characters.
- The main function calls yylex() to start the lexical analysis.

After writing your Lex program, you can follow the steps mentioned above to create and test your lexical analyze

Explain Lex specification

The Lex specification is a crucial component in the creation of a lexical analyzer using Lex as a tool. It defines the rules, patterns, and actions that the lexical analyzer will follow to recognize tokens in the input stream. Here is an explanation based on the provided sources:

1. **Regular Expressions**: The Lex specification involves defining regular expressions to describe patterns for tokens. Regular expressions are mathematical notations that represent sets of strings and are used to define the patterns for tokens in the input stream.
2. **Token Definition**: Tokens are defined by specifying regular expressions that match the lexemes of the tokens. Tokens represent different categories of input strings such as identifiers, keywords, and constants.
3. **Lexemes and Patterns**: Lexemes are sequences of characters in the source program that match the patterns defined for tokens. Patterns describe the structure that the lexemes of a token may take.
4. **Token Structure**: Each token generated by the lexical analyzer consists of a token name and an optional attribute value. The token name represents the kind of lexical unit, and the attribute value provides additional information about the token.
5. **Actions**: In the Lex specification, actions are defined to be taken when a token is recognized. These actions are typically code fragments written in C that specify what should be done when a particular token is identified.
6. **Symbol Table**: The Lex specification may include instructions to enter information about tokens into a symbol table. The symbol table is a data structure that stores information about identifiers used in the program.
7. **Error Handling**: The Lex specification may also include error handling mechanisms to deal with errors encountered during lexical analysis. This ensures that the lexical analyzer can recover from errors and continue processing the input.

Design a lexical analyzer generator.

To design a lexical analyzer generator, we can follow the principles outlined in the content. Here's a breakdown of the key components and steps involved:

The control flow of a lex program is given in Figure 8.1.

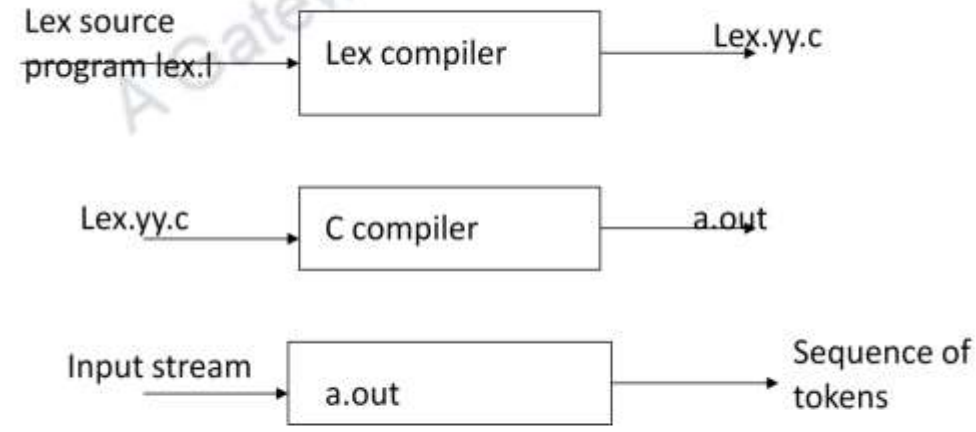


Figure 8.1 Control flow of a lex program

1. **Define the Need**: Understand the necessity of a lexical analyzer generator in the context of compiler construction. Highlight the challenges of designing a lexical analyzer from scratch and emphasize the benefits of using a tool.
2. **Choose a Tool**: Select an appropriate lexical analyzer generator tool based on requirements and preferences. Mention popular tools like LEX, FLEX, or JLEX, each tailored for different environments (e.g., C, Java).
3. **Understand Components**: Explain the core components of a LEX program, including the declaration section, translation rules section, and auxiliary procedures section. Describe how these sections are delimited and their respective roles in the lexical analysis process.
4. **Declaration Section**: Provide examples of regular expressions defined in the declaration section to represent token patterns. Explain the significance of meta characters and their usage in defining patterns.

5. **Translation Rules Section**: Discuss the translation rules section where patterns are matched with the input stream, and corresponding actions are executed upon a match. Illustrate examples of pattern-action pairs for token recognition.
6. **Auxiliary Procedures Section**: Detail the auxiliary procedures section, which contains additional functions and the main program logic. Explain how to handle lexeme processing, symbol table management, and other necessary tasks.
7. **Example LEX Program**: Present example LEX programs demonstrating how to implement basic functionalities such as counting lines, characters, and words in an input stream. Include explanations of each program segment and its purpose.
8. **JLEX Program**: Optionally, introduce JLEX as an alternative to LEX for Java-based environments. Discuss the similarities and differences between LEX and JLEX, and provide an example JLEX program.

What is input buffering? Explain with example.

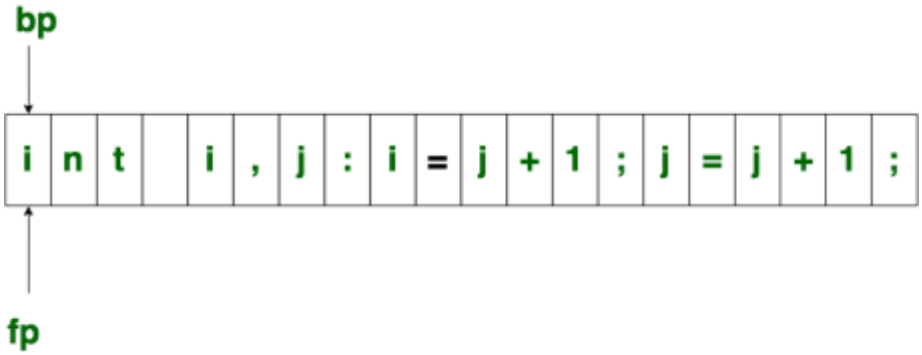
The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward ptr(**fp**) to keep track of the pointer of the input scanned.

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.

1. 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.
2. 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.

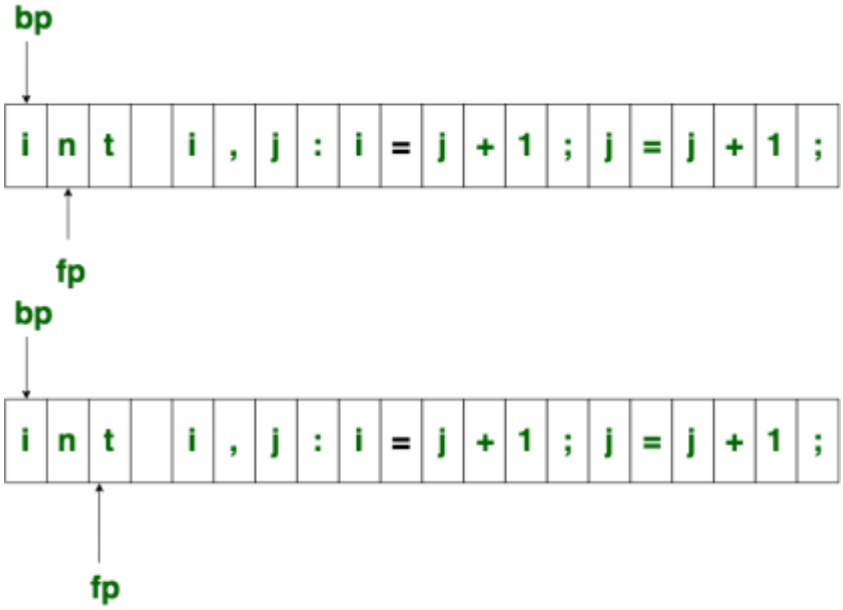
However, there are also some potential disadvantages to input buffering. For example, if the size of the buffer is too large, it may consume too much memory, leading to slower performance or even crashes. Additionally, if the buffer is not properly managed, it can lead to errors in the output of the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



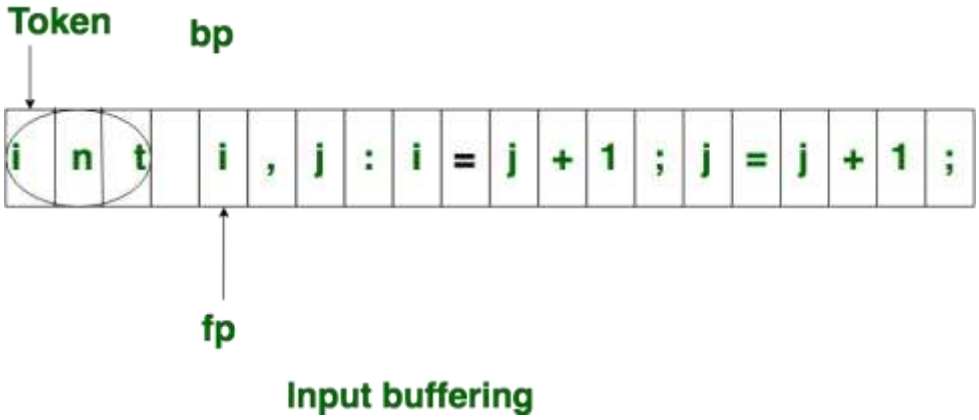
Initial Configuration

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



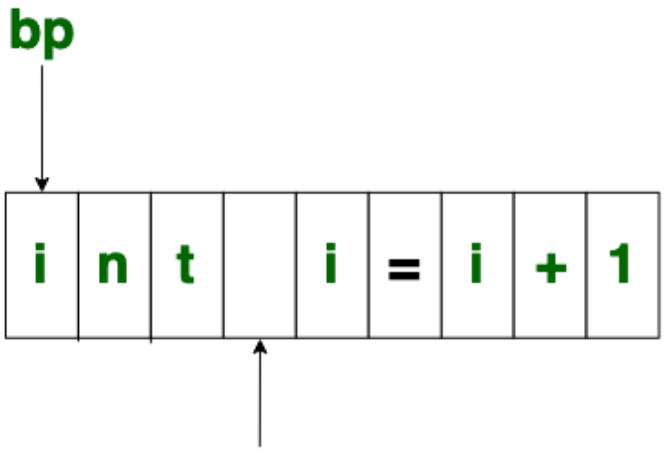
Input Buffering

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

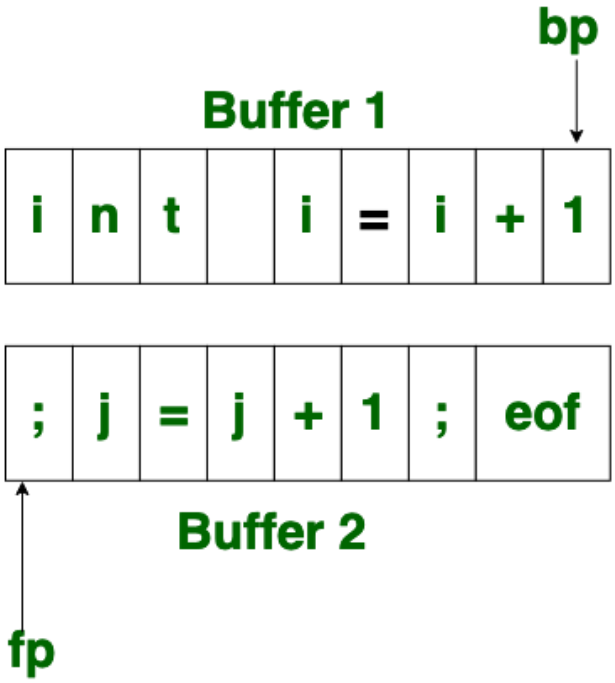
1. **One Buffer Scheme:** In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



One buffer scheme storing input string

2. **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

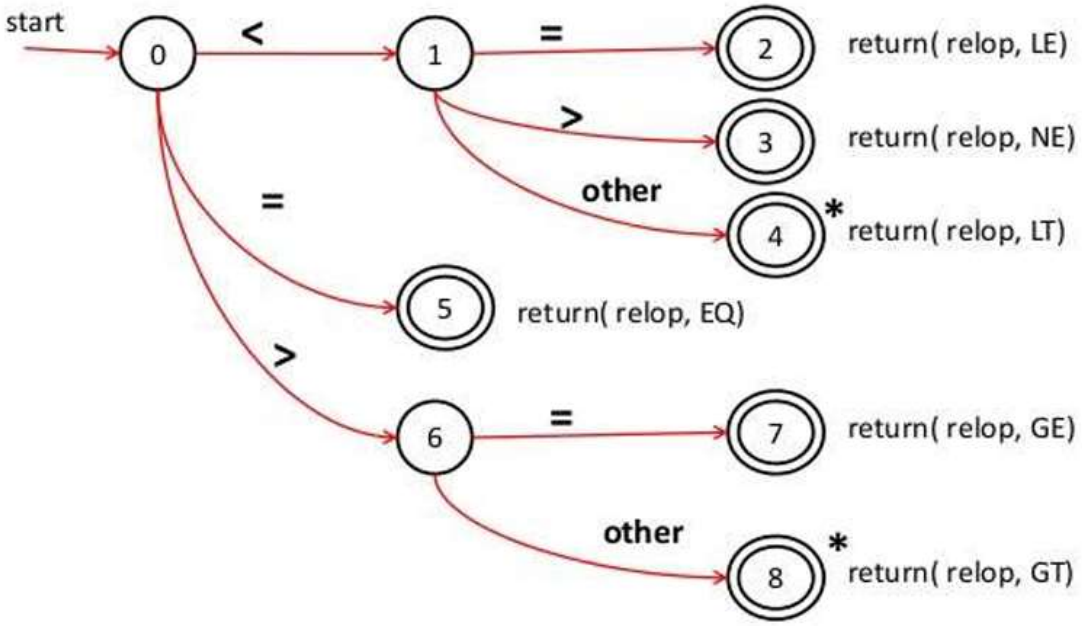
Advantages:

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.

What is input buffering? Explain with example.



1. **The diagram represents the process of identifying different relational operators based on input characters. Here are the key points:**
    - The diagram starts at State “0” labeled “start.”
    - Circles represent states, numbered from 0 to 8.
    - Arrows indicate transitions between states based on input characters like ‘=’, ‘<’, ‘>’, etc.
    - Annotations next to some states indicate the identified relational operators.
2. **Transitions:**
    - If the input character is ‘=’, it moves to state 5, identifying it as an equality operator (EQ).
    - If ‘<’ is encountered, it transitions to state 1. From there:
        - If ‘=’ follows, it goes to state 2 and identifies as a less than or equal to operator (LE).

        - If '>' follows, it goes to state 3 and identifies as a not equal operator (NE).
        - Otherwise, if no other character follows, it goes to state 4 and identifies as a less than operator (LT).
    - If '>' is encountered, it transitions to state 6. From there:
        - If '=' follows, it goes to state 7 and identifies as a greater than or equal to operator (GE).
        - Otherwise, if no other character follows or another character that doesn't form a recognized relop is encountered, it goes to state 8 and identifies as a greater than operator (GT).

3. **Example:**
    - The top of the image shows an example of relational operators: "Ex: RELOP = < <= != > >=."