Week - 1: Lexical analysis using lex tool

Lex is a tool or a computer program that generates Lexical Analyzer (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns**.**

Function of Lex:

1. In the first step the source code which is in the Lex language having the file name 'File.l' gives as input to the Lex Compiler commonly known as Lex to get the output as lex.yy.c.
2. After that, the output lex.yy.c will be used as input to the C compiler which gives the output in the form of an 'a.out' file, and finally, the output file a.out will take the stream of character and generates tokens as output.

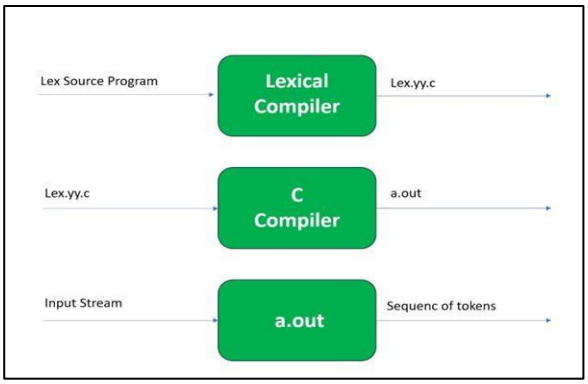


Fig: Block Diagram of Lex

Lex File Format:

A Lex program consists of three parts and is separated by %% delimiters: -

Declarations
%%
Translation rules
%%
Auxiliary procedures

Declarations: The declarations include declarations of variables.
Transition rules: These rules consist of Pattern and Action.
Auxiliary procedures: The Auxiliary section holds auxiliary functions used in the actions.

1.1) Write a lex program whose output is same as input.

Aim: Write a lex program whose output is same as input.

Description:

- Declarations Section:
  - The %{ and %} delimiters enclose C code that will be copied directly into the generated C source file. Here, we include <stdio.h> for printing.
- Rules Section:
  - The rule . matches any single character { printf("%s", yytext); }, which prints the matched character.
- Main Function:
  - The main function calls yylex(), which processes input according to the defined rules.

//Input file "input.txt" having following text

Hi hello how are you?
I hope you are doing good.

Actual Input/Output:

1.2) Write a lex program which removes white spaces from its input file.

Aim: Write a lex program which removes white spaces from its input file.

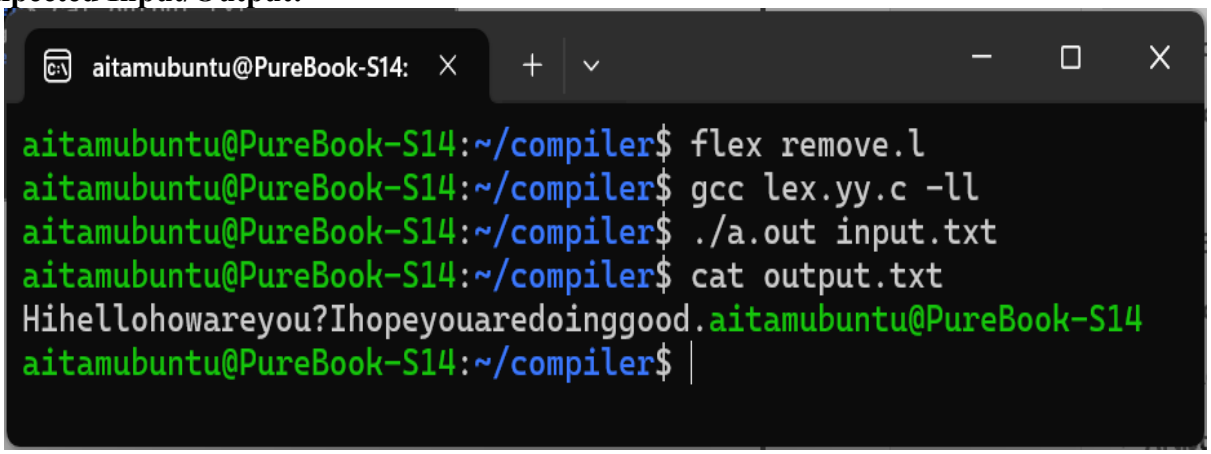
Description:

- Declarations Section:
  - The %{ and %} delimiters enclose C code that will be copied directly into the generated C source file. Here, we include <stdio.h> for file handling.
  - We declare extern FILE *yyin, *yyout; to use these variables for input and output file streams.
- Rules Section:
  - The rule [\t\n] matches spaces, tabs, and newline characters. The action is empty ({ /* Ignore whitespace */ }), meaning it effectively removes these characters from the output.
  - The rule . matches any other character and executes { fprintf(yyout, "%s", yytext); }, which writes the matched character to output.txt.
- Main Function:
  - The program opens input.txt for reading and checks if it was successful.
  - It opens output.txt for writing and checks if that was successful as well.
  - It then calls yylex() to process the input according to the defined rules.
  - Finally, it closes both files before exiting.

//Input file "input.txt" having following text

Hi hello how are you?
I hope you are doing good.

Expected Input/Output:



Viva Questions:

1. What is the input for LEX tool?

Answer: Lex is a tool or a computer program that generates Lexical Analyzer (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns.

2. Define Lexical Analysis?

Answer: The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens. Identifiers, keywords, constants, operators and punctuation symbols are typical tokens.

3. How does the program achieve this?

Answer: It uses regular expressions to identify whitespace characters and discards them while printing all other characters. The `[ \t\n]+` rule matches any sequence of whitespace characters and the semicolon `;` indicates that they should be discarded.

4. Can this program be modified to remove specific types of whitespace characters only?

Answer: Yes, you can modify the regular expression in the rule section to target specific whitespace characters. For example, `[\t\n]+` would only remove tabs and newlines.

Week - 2: Lexical analysis using lex tool

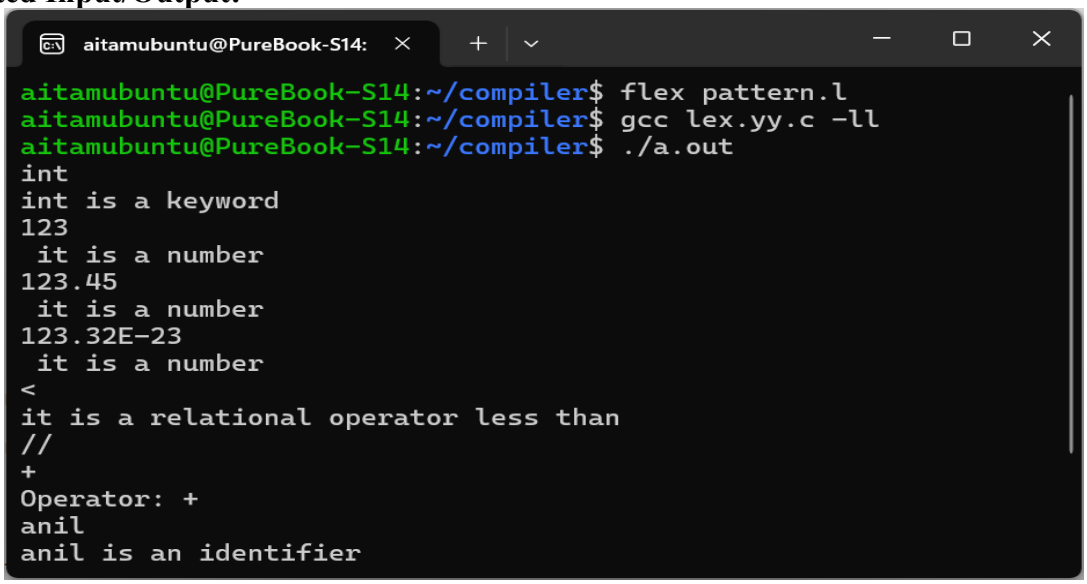
2.1) Write a lex program to identify the patterns in the input file.

Aim: Write a lex program to identify the patterns in the input file.

Procedure:

- Declarations Section:
  - The %{ and %} delimiters enclose C code that will be copied into the generated C source file. Here, we include <stdio.h> for printing and define a helper function print pattern to format output.
  - We define regular expressions for various patterns:
    - keyword: Matches specific keywords like if, else, etc.
    - identifier: Matches variable names that start with a letter or underscore.
    - number: Matches integer values.
    - whitespace: Matches spaces and tabs (ignored).
    - comment: Matches single-line (//) and multi-line (/* ... */) comments (ignored).
    - string: Matches string literals enclosed in double quotes.
- Rules Section:
  - Each pattern has an associated action:
    - For keywords, identifiers, numbers, and strings, we call print pattern to output the type and matched text.
    - Comments and whitespace are ignored.
    - Any unknown character is reported.
- Main Function:
  - Opens input.txt for reading.
  - Calls yylex() to process the input according to defined rules.
  - Closes the input file before exiting.

Expected Input/Output:



2.2) Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines.

Aim: Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines.

Description:

- It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token**: It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.
    - Example: a + b = 20
    - Here, a,b,+,=,20 are all separate tokens.
    - Group of characters forming a token is called the **Lexeme**.
- The lexical analyzer not only generates a token but also enters the lexeme into the symbol table if it is not already there.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
    - Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

isdigit(): The isdigit() in C is a function that can be used to check if the passed character is a digit or not. It returns a non-zero value if it's a digit else it returns 0. For example, it returns a non-zero value for '0' to '9' and zero for others.

The isdigit() function is declared inside ctype.h header file.

C isdigit() Syntax isdigit(int arg);

C isdigit() Parameters

This function takes a single argument in the form of an integer and returns the value of type int.

C isdigit() Return Value: This function returns an integer value on the basis of the argument passed to it.

If the argument is a numeric character then it returns a non-zero value(true value).

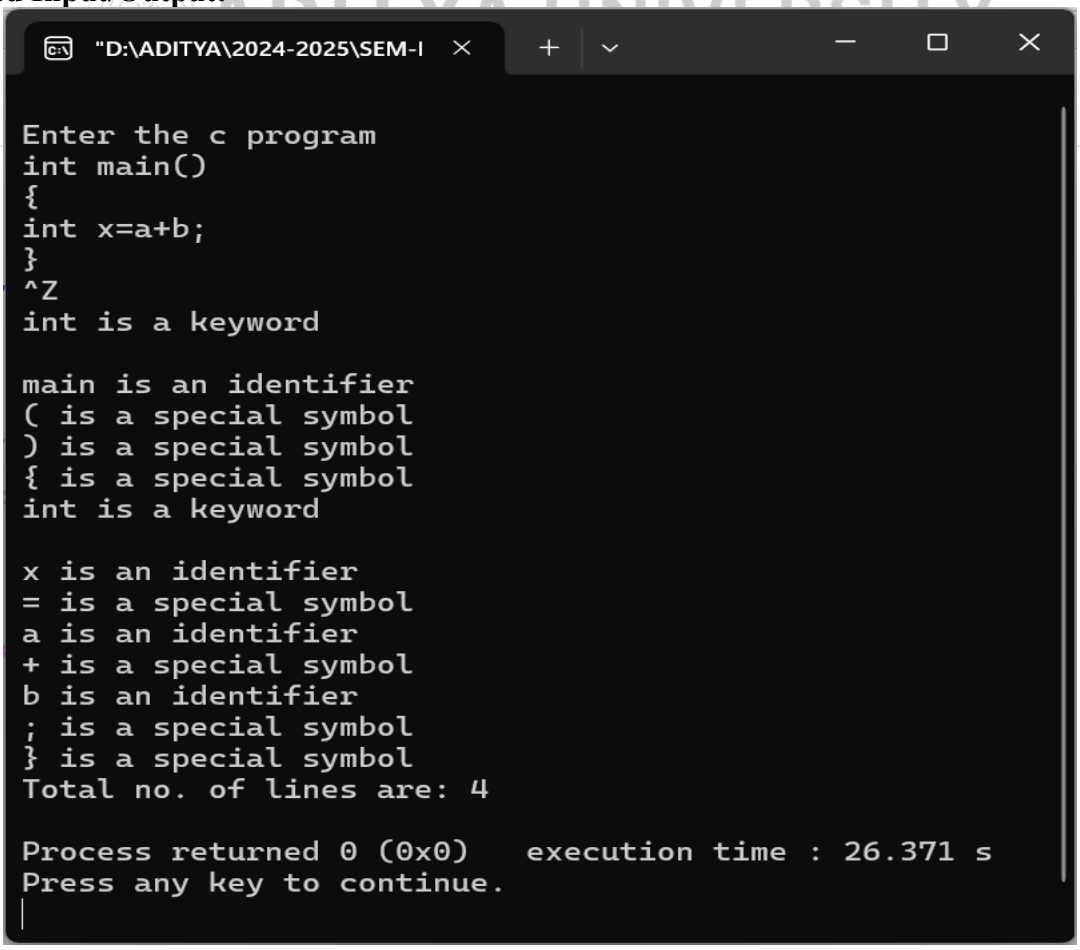
It returns zero(false value) if the argument is a non-numeric character.

isalpha(c): is a function in C which can be used to check if the passed character is an alphabet or not. It returns a non-zero value if it's an alphabet else it returns 0. For example, it returns non-zero values for 'a' to 'z' and 'A' to 'Z' and zeroes for other characters.
Similarly, isdigit(c) is a function in C which can be used to check if the passed character is a digit or not. It returns a non-zero value if it's a digit else it returns 0. For example, it returns a non-zero value for '0' to '9' and zero for others.
Avoiding common errors : It is important to note this article does not cover strings! Only Cstrings. Cstrings are an array of single characters (char) in their behaviour. There are advantages and disadvantages to this.
isalnum(): The function isalnum() is used to check that the character is alphanumeric or not. It returns non-zero value, if the character is alphanumeric means letter or number otherwise, returns zero. It is declared in "ctype.h" header file.
Syntax as int isalnum(int character);
Here, character − The character which is to be checked.

Procedure:

1. Read the C program as input and stores in a file.
2. Check all the characters from the file from left to right whether character is alphabet or digit or special symbol.
3. If the input is operator prints as special symbol.
4. If the input is number prints as number.
5. If the input is identifier prints as identifier.
6. If the input is keyword prints as keyword.

Expected Input/Output:



Viva Questions:

1. Why should redundant whitespace be ignored?

Answer: Ignoring redundant whitespace simplifies further processing stages like parsing and allows for more flexible coding styles without affecting the program's functionality.

2. What is a finite automaton used for in lexical analysis?

Answer: A finite automaton is used to recognize patterns of characters in the input stream and identify matching tokens based on predefined regular expressions.

3. Can you give an example of a regular expression that might be used in a lexical analyzer?

Answer: `[a-zA-Z][a-zA-Z0-9]*` could represent the pattern for identifiers (variable names) consisting of letters followed by zero or more letters or digits (any example).

4. What are tokens?

Answer: Tokens are meaningful units in a programming language, such as keywords (like "if" or "while"), identifiers (variable names), operators (+, -, *, /), literals (numbers and strings), and special symbols.

5. Why is line counting important for a lexical analyzer?

Answer: Line counting enables the lexical analyzer to provide precise error messages by indicating the line number where an error occurred in the source code.

Week - 3: First and Follow

3.1) Simulate First and Follow of a Grammar.

Aim: Simulate First and Follow of a Grammar.

FIRST Procedure:

If the given grammar as

E → TE'
E' →+TE' | ϵ
T → FT'
T' →*FT' | ϵ
F → (E) | id

Rules / Algorithm for FIRST( ):

1. If *X* is terminal, then FIRST(*X*) is {X}.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(*X*).
3. If *X* is non-terminal and $X \rightarrow a\alpha$ is a production then add *a* to FIRST(X).
4. If X is non-terminal and $X \rightarrow Y1\ Y2 \ldots Yk$ is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1); that is, Y1,….Yi-1 => ε. If ε is in FIRST(*Yj*) for all j=1,2,..,k, then add ε to FIRST(*X*).

memset():

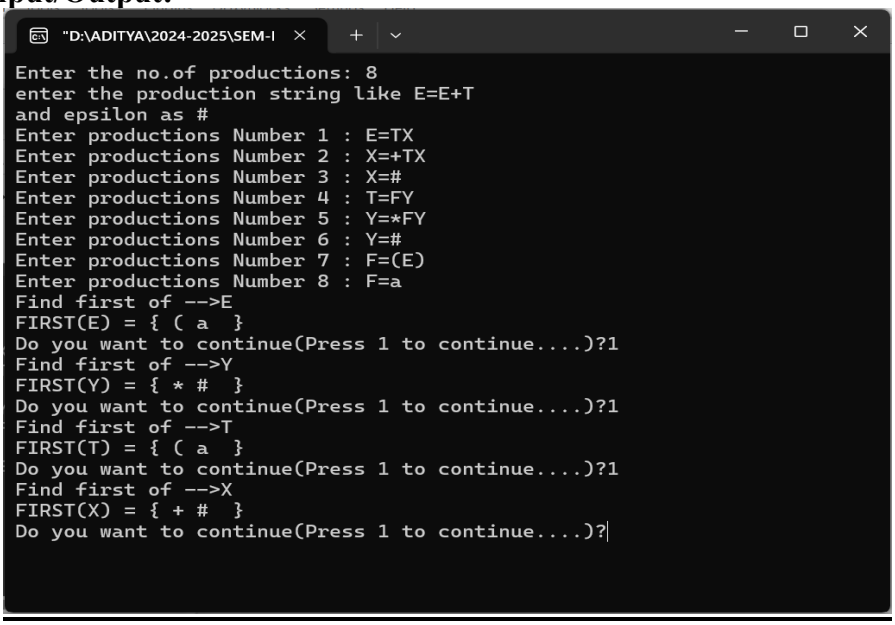
memset() is used to fill a block of memory with a particular value.

The syntax of memset() function is as follows :

```
// ptr ==> Starting address of memory to be filled
// x   ==> Value to be filled
// n   ==> Number of bytes to be filled starting
//         from ptr to be filled
void *memset(void *ptr, int x, size_t n);
```

Note that ptr is a void pointer, so that we can pass any type of pointer to this function.

Expected Input/Output:



FOLLOW Procedure:

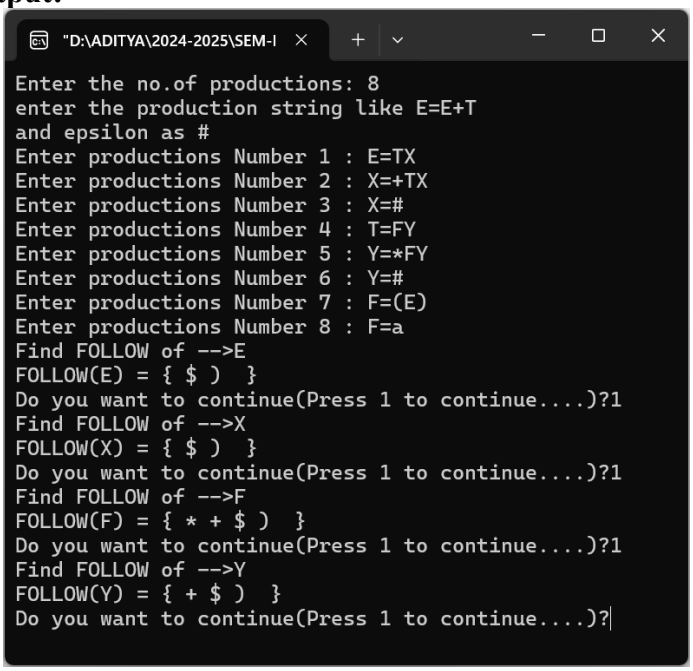
If the given grammar as

E → TE'
E' →+TE' | ϵ
T → FT'
T' →*FT' | ϵ
F → (E) | id

Rules / Algorithm for FOLLOW():

1. If *S* is a start symbol, then FOLLOW(*S*) contains $.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is placed in follow(*B*).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).

Expected Input/Output:



Viva Questions:

1) What is the purpose of FIRST and FOLLOW sets in compiler design?

Answer: They help in constructing predictive parsers by providing information about the possible terminal symbols that can appear at specific points in a grammar rule. This enables the parser to make decisions based on the current input symbol and predict the next production rule to apply.

2. How is FIRST(X) defined?

Answer: FIRST(X) is the set of all terminal symbols that can appear as the first symbol in a derivation starting from a non-terminal X. This means it includes all the possible initial terminals that can be generated by expanding X.

3. What does FOLLOW(X) represent?

Answer: FOLLOW(X) represents the set of all terminal symbols that can appear immediately after a non-terminal X in a valid derivation.

4. Why are FIRST and FOLLOW sets essential for LL(1) parsers?

Answer: LL(1) parsers are predictive parsers, meaning they must be able to predict the next production rule based on the current input symbol alone. FIRST and FOLLOW sets provide this crucial information by defining the possible starting and following terminals for each non-terminal.

3.2) Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.

Aim: Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.

Procedure:

1. Save the above program as lexer.l.
2. Open a terminal and navigate to the directory containing lexer.l.
3. Compile the program using lex and gcc:
   a. lex lexer.l
   b. gcc lex.yy.c -ll
4. Run the program:
   ./a.out

Expected Input/Output:

```
aitamubuntu@PureBook-S14:
aitamubuntu@PureBook-S14:~/compiler$ flex lexfun.l
aitamubuntu@PureBook-S14:~/compiler$ gcc lex.yy.c -ll
aitamubuntu@PureBook-S14:~/compiler$ ./a.out file.c
// FUNCTION:sample
 FUNCTION:program

int is a KEYWORD
 FUNCTION:main
(
        )

BLOCK BEGINS

int is a KEYWORD
 FUNCTION:a
= is an ASSIGNMENT OPERATOR
20 is a NUMBER
;
char is a KEYWORD
 FUNCTION:c
;
BLOCK ENDS
```

Viva Questions:

1) How does Lex work?

Answer: Lex uses regular expressions to define patterns for different types of tokens. When given an input stream, it matches the characters against these patterns. If a match is found, a corresponding token is generated along with its value (lexeme).

2) What are some benefits of using Lex for implementing lexical analyzer?

Answer: Efficiency: Lex-generated scanners are efficient and fast.
Flexibility: Regular expressions allow for powerful and flexible pattern matching.
Maintainability: The separation of token definition from the parsing logic makes it easier to maintain and modify the lexical analyzer.

3) Can you give an example of a simple Lex rule?

Answer:

```
%%
[0-9]+     { printf("NUMBER: %s\n", yytext); }
\n         { /* ignore newline characters */ }
.          { printf("UNKNOWN: %s\n", yytext); }
%%
```

4) What is Lex?

Answer: Lex is a tool used for generating lexical analyzers. It takes a set of regular expressions as input, defining the structure of tokens, and generates C code that implements the lexical analyzer.

Week - 4: Top-Down Parsing

4.1) Develop an operator precedence parser for a given language.

Aim: Develop an operator precedence parser for a given language.

Procedure:

A grammar is said to be operator precedence grammar if it has two properties:

No R.H.S. of any production has a∈.
No two non-terminals are adjacent.

Logic as follow:

Read the given input string.

Construct a Operator precedence parsing table for the given grammar by using below rules.

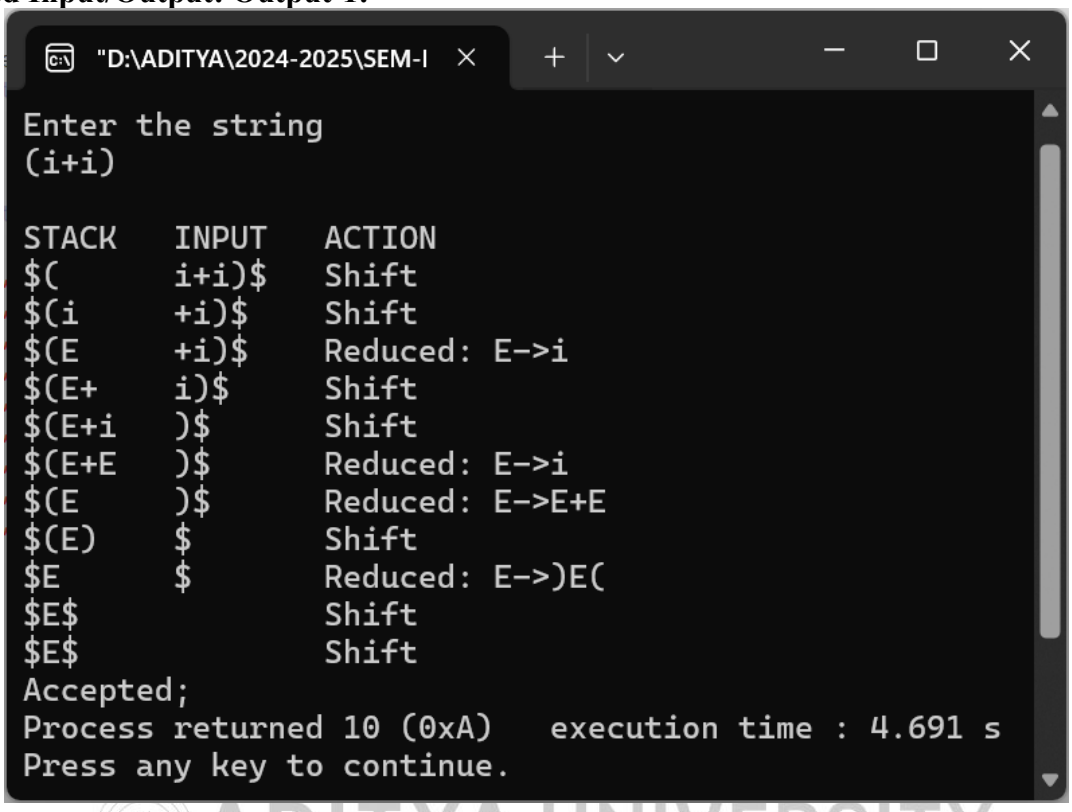
- If a has higher precedence over b; a .> b
- If a has lower precedence over b; a <. b
- If a and b have equal precedence; a =. b

Verify the given input string is accepted or not using below rules.

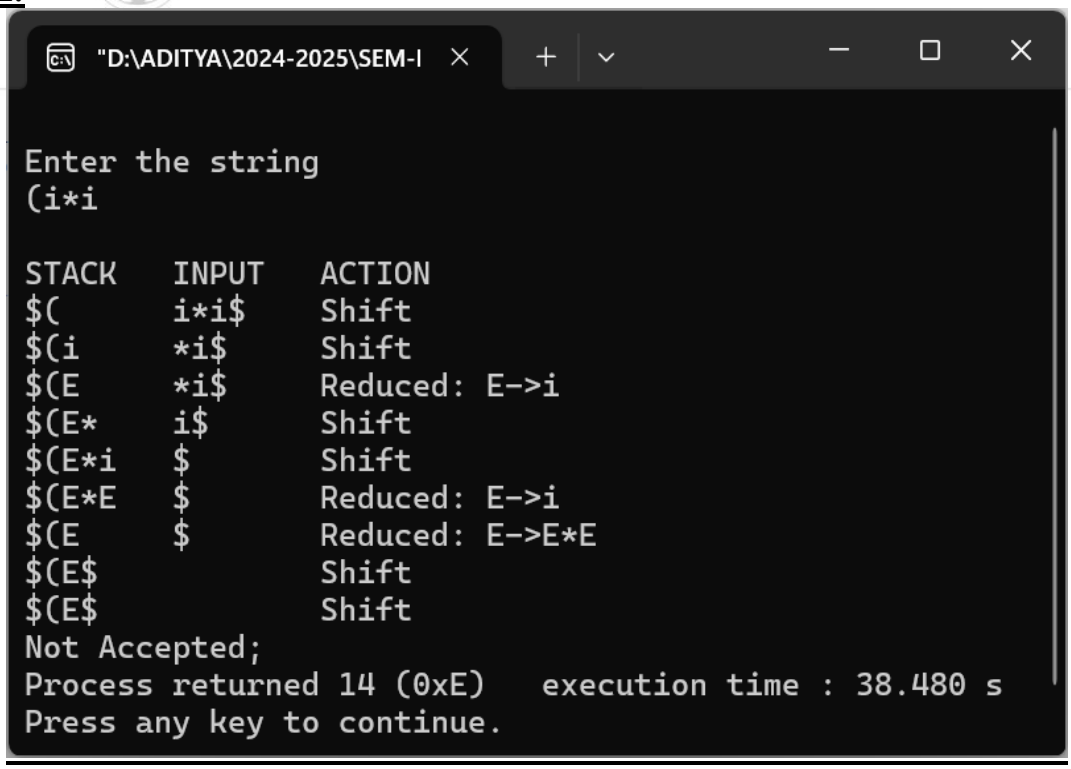
Scan input string left to right, try to detect .> and put a pointer on its location.
Now scan backwards till reaching <.
String between <. And .> is our handle.
Replace handle by the head of the respective production.
REPEAT until reaching start symbol.

If it accepts prints as accepted
Else not accepted.

Expected Input/Output: Output-1:



Output-2:



Viva Questions:

1) What does an operator precedence parser use to determine the order of operations?

Answer: It uses a predefined precedence table that defines the relative priority of different operators.
2) How are operands handled during parsing?
Answer: Operands are pushed onto the stack.
3) What happens when an operator with lower precedence is encountered on the input stream?
Answer: Operators with higher precedence are popped from the stack and applied to operands until an operator with lower precedence is found.
4) What data structure is crucial for the functioning of an operator precedence parser?
Answer: A stack is used to store operators and operands during parsing.
5) What is the final output of an operator precedence parser?
Answer: The parser produces the evaluated result of the input expression.

4.2) Construct a recursive descent parser for an expression.

Aim: Construct a recursive descent parser for an expression.

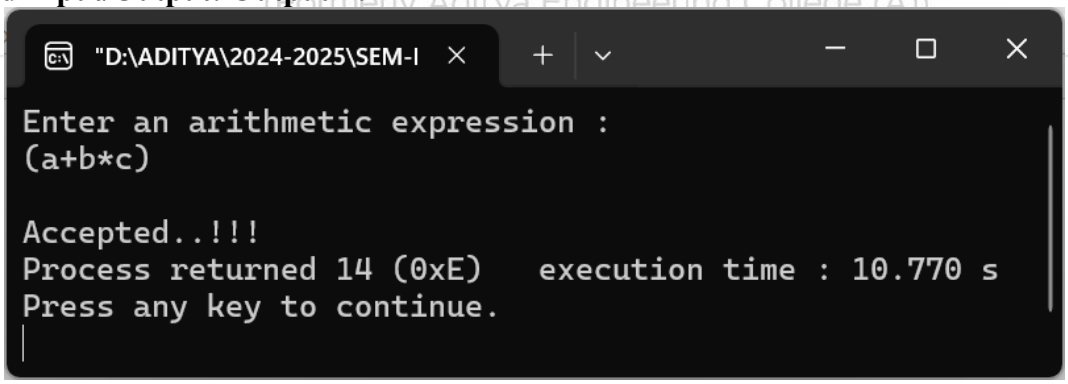
Procedure:

- Recursive descent parsing is one of the top-down parsing techniques, which uses a set of recursive procedures to scan its input.
- This parsing method may involve backtracking, that is, making repeated scans of the input.

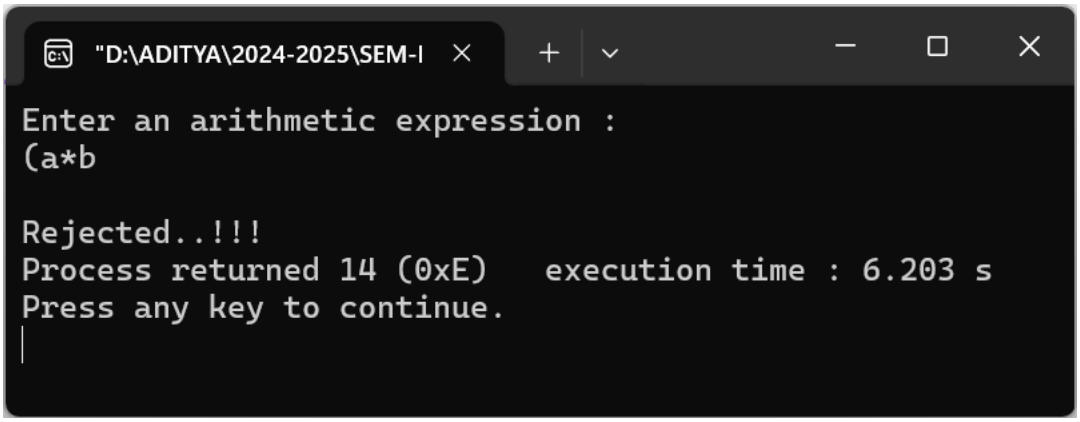
Logic / Algorithm:

- First identify all the non-terminal in the given grammar (which, not having left-recursion)
- Write the procedure/function for every non-terminal.
- Read the input string.
- Verify the next token equals to non-terminals if it satisfies match the non-terminal.
- If the input string does not match print error message.

Expected Input/Output: Output-1:



Output-2:



Viva Questions:

1) What is the basic principle behind a recursive descent parser?
Answer: A recursive descent parser follows the structure of the grammar rules directly. Each non-terminal symbol in the grammar has a corresponding function that attempts to match that part of the input stream. These functions call each other recursively to build up the parse tree.
2) What are some advantages of using a recursive descent parser?
Answer: Recursive descent parsers are relatively easy to understand and implement compared to other parsing techniques like LR(1) or LALR(1). They are also well-suited for handling simple grammars and can be easily extended to handle more complex ones.
3) What is a common issue faced when constructing a recursive descent parser?
Answer: One of the main challenges with recursive descent parsing is dealing with left recursion in the grammar. Left recursion can cause infinite loops during parsing. This issue needs to be addressed by transforming the grammar before implementing the parser.
4) How would you handle error recovery in a recursive descent parser?
Answer: Error recovery strategies vary depending on the specific application. Some common techniques include:
Panic mode recovery: Skip tokens until a synchronizing token is found, allowing parsing to continue from a known good point.
Error productions: Adding special rules to the grammar that handle expected errors gracefully.
5) Define top-down parsing?
A) It can be viewed as an attempt to find the left most derivation for an input string. It can be viewed as attempting to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

Week - 5: Bottom-up Parsing

5.1) Construct a LL(1) parser for an expression.

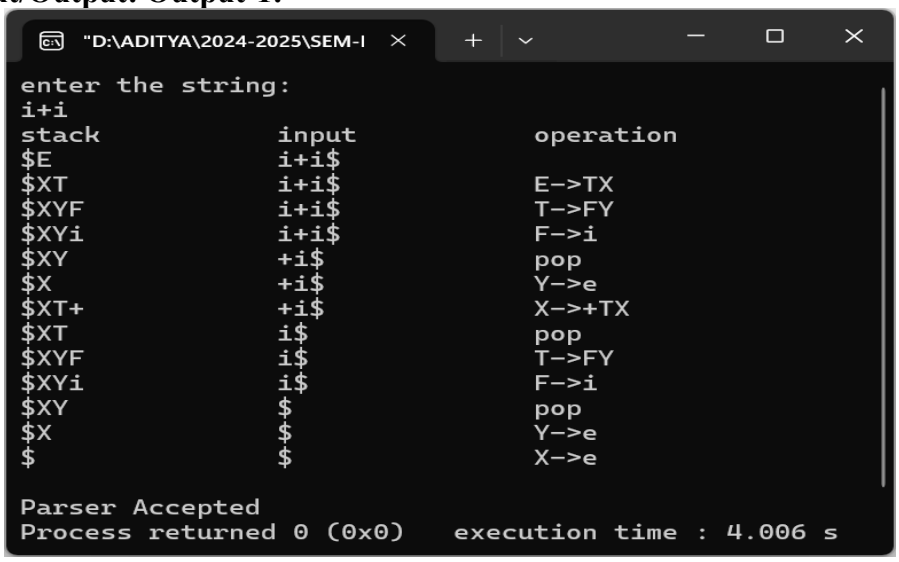
Aim: Construct a LL(1) parser for an expression

Procedure:

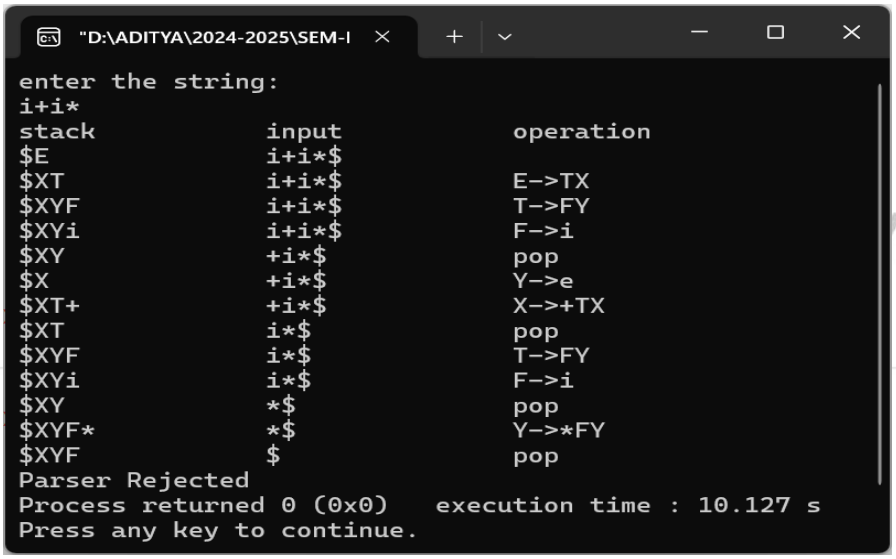
Input: A string *w* and a parsing table *M* for grammar *G*.
Output: If *w* is in *L*(*G*), a leftmost derivation of *w*; otherwise, an error indication.
Method: Initially, the parser has $*S* on the stack with *S*, the start symbol of *G* on top, and *w*$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

set *ip* to point to the first symbol of *w*$; **repeat**
let *X* be the top stack symbol and *a* the symbol pointed to by *ip*; **if** *X* is a terminal
or $ **then**
if *X* = *a* **then**
pop *X* from the stack and advance *ip* **else** e*rror*()
else /* *X* is a non-terminal */
if *M*[*X*, *a*] = *X* → *Y1 Y2* … *Yk* **then begin**
pop *X* from the stack;
push *Yk*, *Yk-1*, … ,*Y1* onto the stack, with *Y1* on top;
output the production *X* → *Y1 Y2* . . . *Yk*
End
else *error*()
until *X* = $ /* stack is empty */

Expected Input/Output: Output-1:



Output-2:



Viva Questions:

1) What does LL(1) stand for in the context of parsing?

Answer: LL(1) stands for Left-to-right, Leftmost derivation, 1 lookahead symbol.

2) Why is a grammar being LL(1) important for parser construction?

Answer: An LL(1) grammar ensures that the parser can make deterministic decisions based on a single lookahead symbol, preventing ambiguity during parsing.

3) What is the role of the parsing table in an LL(1) parser?

Answer: The parsing table guides the parser by mapping non-terminal symbols and lookahead symbols to appropriate production rules from the grammar.

4) Can any context-free grammar be used to construct an LL(1) parser?

Answer: No, only grammars that satisfy certain conditions, known as LL(1) conditions, can be used.

5) What happens if a cell in the parsing table is empty for a given non-terminal and lookahead symbol?

Answer: An empty cell indicates a syntax error in the input expression.

5.2) Design a LALR bottom-up parser for the given language.

Aim: Design a LALR bottom-up parser for the given language.

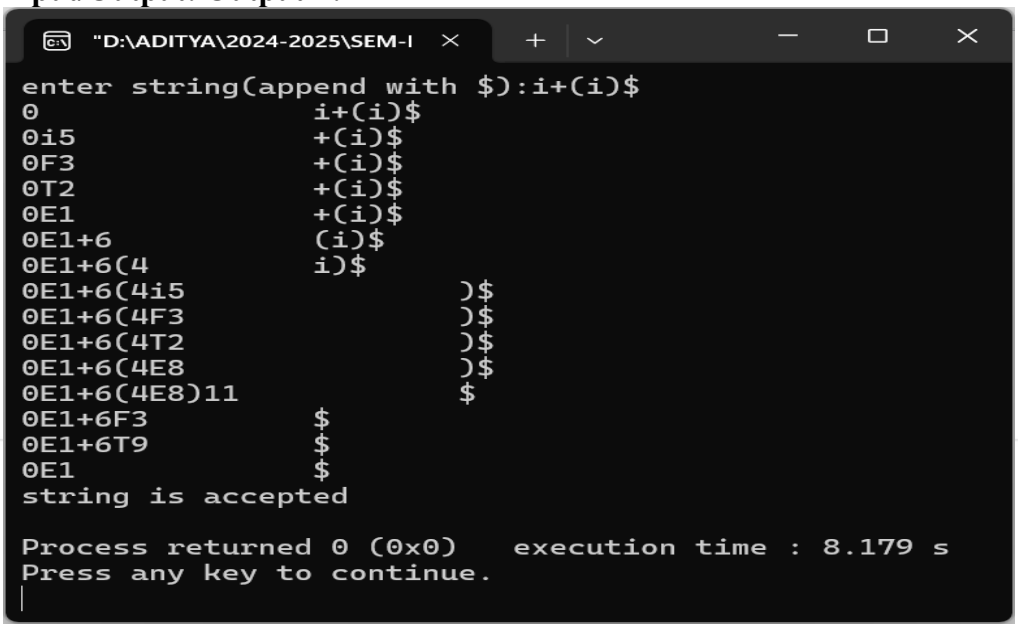
Procedure:

Define Your Grammar: You'll need a formal grammar that describes the language you want your parser to handle. This grammar will consist of terminals (the basic symbols in your language), non-terminals (symbols representing intermediate constructs), and production rules (rules defining how non-terminals can be expanded into terminals and other non-terminals).

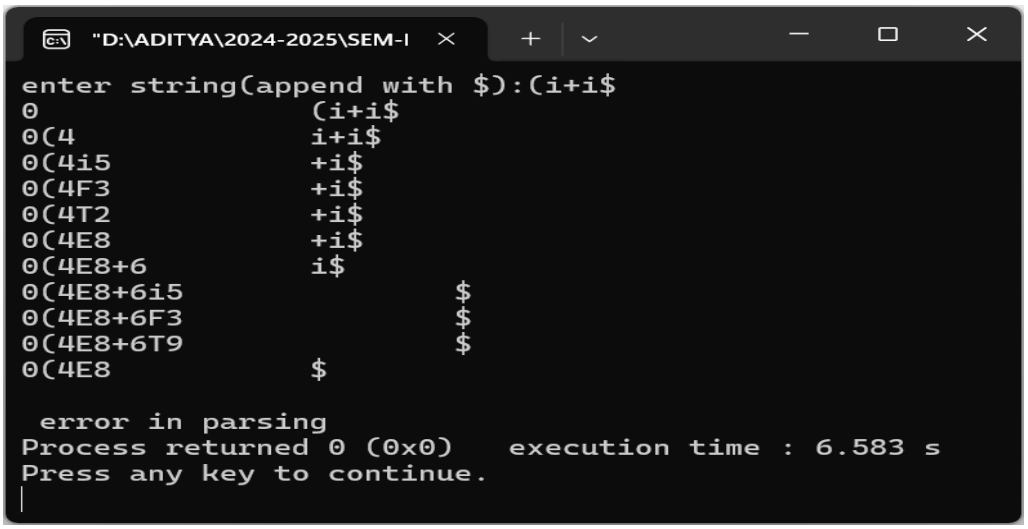
Construct the LALR(1) Parsing Table: This table is the heart of the parser. It's generated from your grammar and specifies what actions to take for each combination of a state (representing progress in parsing), a lookahead symbol, and an input token. Actions: These can include "shift" (move to a new state), "reduce" (apply a production rule to combine symbols on the stack), "accept" (successful parsing), or "error" (indicating a syntax error).

Implement the Parser Logic: The parser follows the instructions in the LALR(1) table, maintaining a stack of symbols and processing input tokens one by one.

Expected Input/Output: Output-1:



Output-2:



Viva Questions:

1) What is the primary advantage of using an LALR(1) parser over a simple recursive-descent parser?
Answer: LALR(1) parsers are more powerful. They can handle a wider range of grammars, including those with left recursion and other ambiguities that might cause problems for simpler recursive-descent parsers.
2) What does the "1" in LALR(1) represent?
Answer: It indicates that the parser looks ahead at one token (symbol) to make parsing decisions. This lookahead capability helps resolve ambiguities in the grammar.
3) How is a shift-reduce conflict resolved in an LALR(1) parsing table?
Answer: Shift-reduce conflicts occur when the parser is uncertain whether to shift a new input token onto the stack or reduce a series of symbols on the stack using a production rule. In such cases, careful grammar design and potentially additional precedence rules are used to resolve the conflict.
4) What role does the stack play in a bottom-up parser?
Answer: The stack stores a sequence of symbols (terminals and non-terminals) that represent the partially parsed input. It's essential for tracking the structure of the input sentence as the parser proceeds.
5) Are LALR(1) parsers suitable for all types of grammars?
Answer: No, there are some grammars that are inherently ambiguous or beyond the capabilities of LALR(1) parsing. For such cases, more powerful parsing techniques might be necessary.

Week - 6: Optimization Phase

6.1) Write a program to perform loop unrolling.

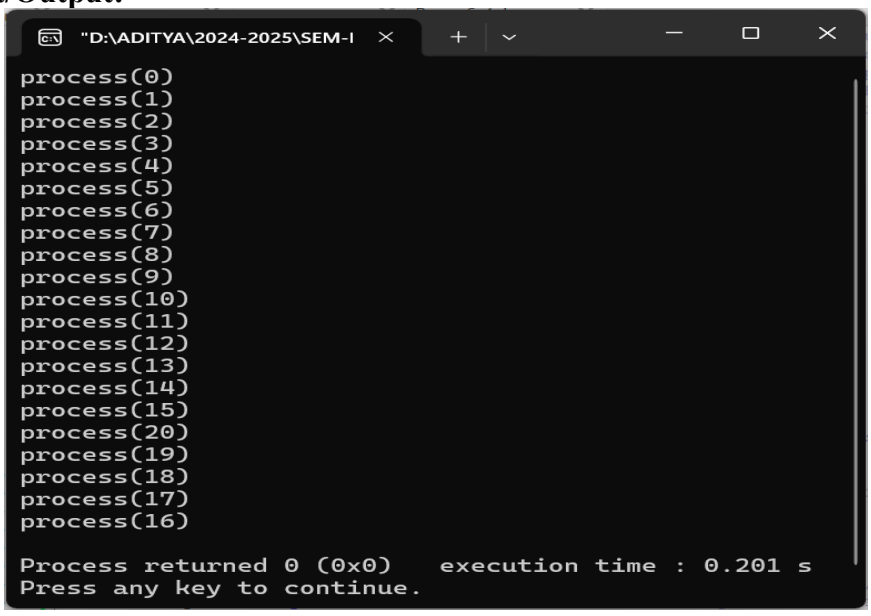
Aim: Write a program to perform loop unrolling.

Procedure:

Loop unrolling, also known as loop unwinding, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, which is an approach known as the space-time trade-off. The transformation can be undertaken manually by the programmer or by an optimizing compiler.

Loop unrolling is a compiler optimization technique that attempts to improve the performance of loops by reducing the number of iterations needed. This is achieved by replicating the loop body multiple times, effectively processing several loop iterations in a single pass.

Expected Input/Output:



Viva Questions:

1) What is loop unrolling?

Answer: Loop unrolling is a compiler optimization technique that reduces the number of iterations in a loop by replicating the loop body multiple times.

2) How does loop unrolling improve performance?

Answer: It eliminates loop control overhead, allowing each iteration to be executed more efficiently.

3) What are the potential downsides of loop unrolling?

Answer: Increased code size and limited applicability to loops with variable iteration counts or complex dependencies.

4) Give an example of a loop that would benefit from unrolling.

Answer: A simple loop with a fixed number of iterations, like summing elements in an array.

5) When might loop unrolling be less effective?

Answer: When dealing with loops that have variable iteration counts or complex dependency relationships between iterations.

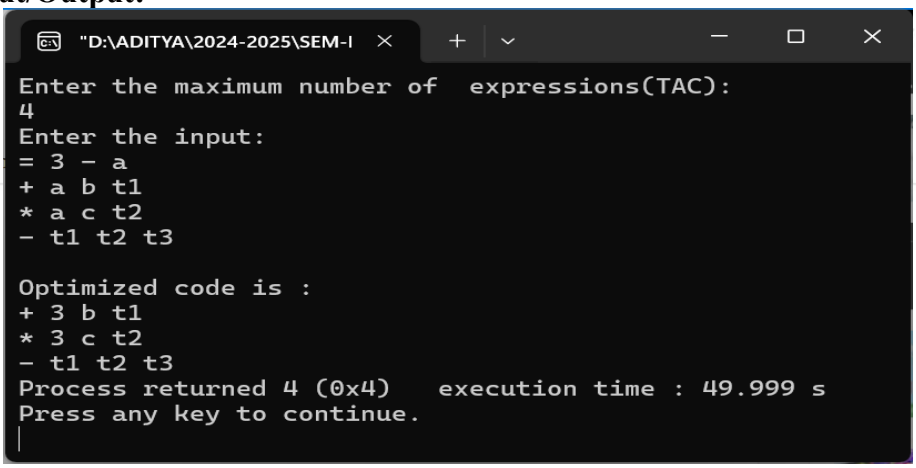
6.2) Write a program for constant propagation.

Aim: Write a program for constant propagation.

Procedure:

Constant propagation is an optimization technique used by compilers to improve the efficiency of code. It involves analyzing your program to identify expressions whose results are always constant values (known at compile time). Once identified, these constants can be directly substituted throughout the code wherever the expression appears.

Expected Input/Output:



Viva Questions:

1)What is the primary goal of constant propagation?

Answer: To improve code efficiency by replacing expressions with known constant values at compile time.

2) How does constant propagation benefit software development?

Answer: It leads to faster execution, smaller code size, and sometimes improved readability.

3) What are some challenges in implementing constant propagation?

Answer: Handling complex control flow (loops, conditionals), dependencies on external factors (user input), and ensuring the correctness of substitutions can be challenging.

4) Why is constant propagation beneficial?

Answer: *Faster Execution:* Replacing complex calculations with simple constant values speeds up runtime.

Smaller Code Size: Eliminating redundant computations often leads to smaller executable files.

Improved Readability: Sometimes, substituting constants can make your code more understandable.