**EXPT 2**

**Algorithm 2.1**

1. Include necessary header files.

2. Define patterns to recognize keywords, numbers, words, and any other character.

3. In the main function, print a prompt.

4. Invoke the lexer (yylex).

5. In the lexer, recognize tokens and print them.

6. If the lexer returns, end the program.

**Algorithm 2.2**

1. Include necessary header files.

2. Define patterns to recognize vowels and consonants.

3. In the main function, print a prompt.

4. Invoke the lexer (yylex).

5. In the lexer, recognize vowels and consonants and print them.

6. If the lexer returns, end the program.

**Algorithm 2.3**

1. Include necessary header files.

2. Define patterns to recognize small letters and capital letters.

3. In the main function, print a prompt.

4. Invoke the lexer (yylex).

5. In the lexer, recognize small letters and capital letters and print them.

6. If the lexer returns, end the program.

**Algorithm 2.4**

1. Include necessary header files and declare counters for vowels and consonants.

2. Define patterns to recognize vowels, consonants, and newline characters.

3. In the main function, print a prompt.

4. Invoke the lexer (yylex).

5. In the lexer, count vowels and consonants.

6. When a newline is encountered, print the counts and end the program.

**THEORY**

Lex is a vital tool in compiler construction and text processing, used to create lexical analyzers or lexers. Lexical analyzers break down input streams into tokens, which are fundamental units of information. These tokens serve as the basis for subsequent phases of a compiler or for text processing tasks.

Lex programs are structured into three sections: definitions, rules, and user-defined functions. In the definitions section, constants, data types, and regular expressions are declared. The rules section defines patterns to match against the input, and each pattern is associated with actions to be executed when a match occurs. User-defined functions can also be used to customize the lexer's behavior.

The core of Lex's functionality lies in pattern matching using regular expressions. These expressions describe the structure of tokens and enable Lex to identify and separate language constructs like keywords, identifiers, operators, and literals in programming languages. When a pattern matches, an associated action is executed. These actions typically result in the creation of tokens with attributes. For example, Lex can recognize keywords, assign them a type, and provide them to the parser for further analysis.

Lex tools are particularly essential in compiler development. They simplify the task of creating lexers, which are the first phase of a compiler. Lexers convert source code into tokens, which are subsequently passed to a parser for further analysis and code generation.

**EXPT 3**

**Algorithm 3.1**

1. Include the necessary headers for Lex and Yacc.

2. Define the token types and associativity in the Yacc file.

3. In the Lex file, define patterns to match numbers, identifiers, whitespace, newline, and other characters.

4. Associate actions with Lex patterns to return tokens and attribute values.

5. In the Yacc file, specify grammar rules to parse and evaluate mathematical expressions.

6. Define actions within the Yacc rules to perform calculations.

7. In the main function, print a prompt.

8. Invoke the Yacc parser (yyparse).

9. Handle errors using the yyerror function.

10. Compile the Lex and Yacc files and create the executable.

**Algorithm 3.2**

1. Include headers and declare lexer and parser functions.

2. Define tokens for numbers and operators.

3. In yylex:

   a. Read character from input.

   b. If it's an operator or digit, return it as a token.

4. In yyparse:

   a. Define valid expression grammar.

   b. Print "Valid" for a valid expression, or "Invalid" for syntax errors.

5. In main, call yyparse

6. Compile the program using yacc and lex tools.

**Algorithm 3.3**

1. Include headers and declare lexer and parser functions.

2. Define the VAR token.

3. In yylex:

   a. Read character from input.

   b. If it's a letter or underscore, read the entire variable and return it as VAR.

4. In yyparse:

   a. Define valid variable grammar.

   b. Print "Valid" for valid variables, or "Invalid" for syntax errors.

5. In main, call yyparse.

6. Compile the program using yacc and lex tools.

**THEORY**

Yacc, or Yet Another Compiler Compiler, is a powerful tool for generating parsers and syntax analyzers. It plays a pivotal role in the construction of compilers, interpreters, and other software systems that require the analysis of complex structured languages.

At its core, Yacc operates on context-free grammars, making it suitable for recognizing and processing the syntax of a wide range of programming languages and data formats. Yacc's theoretical foundation is based on formal language theory and automata theory, specifically context-free grammars and pushdown automata.

The main components of a Yacc specification include grammar rules, tokens (terminals), and actions. Grammar rules define the syntax of the language being parsed, tokens represent the language's basic building blocks, and actions define the behavior associated with each rule. Yacc generates parsers based on the provided specifications that can parse input according to the defined grammar.

Yacc parsers use a bottom-up parsing technique, constructing parse trees or abstract syntax trees from the input. These parse trees represent the structure of the input language, allowing for further processing, optimization, or code generation.

**EXPT4**

**Algorithm 4.1**

1. Include headers and define lexer rules.

2. Handle whitespace.

3. Recognize and ignore single-line comments.

4. Recognize and ignore multi-line comments.

5. For everything else, print the text.

6. In the main function:

   a. Open the input file (source.txt).

   b. Set yyin to the file.

   c. Invoke yylex.

   d. Close the file.

   e. Print the result.

7. Compile using lex.

**Algorithm 4.2**

1. Include headers and define lexer rules.

2. Handle whitespace and newline.

3. Print everything else as text.

4. In the main function:

   a. Open the input file (source1.txt).

   b. Set yyin to the file.

   c. Invoke yylex.

   d. Close the file.

   e. Print the result.

5. Compile using lex.

**THEORY**

Lexical analysis is a crucial phase in the compilation process, serving as the initial step in transforming human-readable source code into machine-understandable tokens. Two code examples are provided to illustrate this process, where lex (lexical analyzer generator) is employed.

In the first example, a C comment extractor is implemented, demonstrating the use of regular expressions to recognize and ignore single-line and multi-line comments. The code reads an input file, processes it, and prints the remaining text. This is essential for preprocessing source code files.

The second example showcases tokenization in the context of C programming. Here, lex is used to identify various token types, including keywords, operators, numbers, identifiers, strings, and punctuation marks. Each token is associated with a specific action, such as printing or categorizing. This tokenization is fundamental for subsequent parsing and semantic analysis.

Lexical analysis plays a vital role in error detection and source code transformation. By breaking down source code into tokens, it simplifies the parsing process and facilitates code comprehension. The provided code snippets demonstrate the power of lex in extracting meaningful information from source code and lay the foundation for subsequent compilation phases.

**EXPT 5 (NFA TO DFA)**

**THEORY**

Nondeterministic Finite Automata (NFA) and Deterministic Finite Automata (DFA) are two types of finite automata used in formal language theory and automata theory to recognize patterns or languages. The conversion from an NFA to a DFA is a fundamental concept in automata theory.

NFAs allow multiple transitions from a state on the same input symbol, introducing nondeterminism. In contrast, DFAs have a unique transition for each input symbol from each state. Converting an NFA to a DFA involves representing the NFA's behavior deterministically.

The theoretical process of conversion entails creating a DFA state for each subset of NFA states, where each DFA state corresponds to a set of NFA states reachable on the same input. The transition function is then defined for the DFA to mimic the NFA's transitions, using epsilon-closure for epsilon transitions.

The conversion ensures that DFAs are more efficient for language recognition, as they eliminate nondeterminism. However, the resulting DFAs may have a larger number of states due to the power set construction.

This transformation from NFA to DFA plays a pivotal role in simplifying the design and analysis of regular expressions, lexical analyzers, and parsers in computer science and compilers. It allows more efficient pattern matching and language recognition, making it a cornerstone in automata theory and formal language processing.

**ALGORITHM**

1. Start with the initial state of the NFA as the initial state of the DFA.

2. Create an empty set as the initial state of the DFA, which will represent the set of states reachable from the initial state of the NFA.

3. While there are unmarked sets of states in the DFA:

   a. Mark the set.

   b. For each input symbol, find the set of states reachable from the current set using the NFA's transition function.

   c. If the new set is not in the DFA, add it as an unmarked set.

4. Repeat step 3 until all sets are marked.

5. The resulting marked sets of states in the DFA represent the states of the converted DFA.

6. Define the transition function for the DFA using the NFA's transition function and the marked sets of states.

7. Determine the set of final states in the DFA based on the final states of the NFA that are contained in the marked sets.

8. The DFA is now ready for language recognition and pattern matching.

**EXPT 6 (first and follow)**

**THEORY**

In the context of formal language theory and parsing, "First" and "Follow" sets are essential concepts used to analyze and construct parsers for context-free grammars (CFGs).

The "First" set of a non-terminal or terminal symbol is the set of terminal symbols that can begin a string derived from that symbol in one parsing step. It helps determine which production rule to apply when parsing. "First" sets are crucial in LL(1) parsing, a type of top-down parsing, as they allow the parser to predict the correct production based on the next input symbol.

On the other hand, the "Follow" set of a non-terminal represents the set of terminal symbols that can immediately follow the non-terminal in a valid sentence. It is fundamental in LL(1) and LR(1) parsing, aiding in handling productions with empty (ε) strings and reducing parser conflicts.

Constructing these sets involves systematically analyzing the CFG, considering nullable variables, recursive definitions, and various production rules. Both sets help identify parsing errors, reduce ambiguity, and optimize parser generation. Efficient parsers for programming languages, like LL, LR, or LALR parsers, rely on these sets to automatically generate parser tables, facilitating syntactical analysis. In summary, "First" and "Follow" sets are indispensable tools in parsing theory, enabling the development of robust, unambiguous parsers for various programming languages.

**ALGORITHM**

1. Initialize empty First and Follow sets for each non-terminal.

2. For each production P in the grammar:

a. Calculate First(P) by examining the first symbols of the right-hand side.

b. Add First(P) to First of the left-hand side non-terminal.

c. If epsilon is in First(P), consider the next symbol for First(P).

d. If epsilon is not in First(P), stop and move to the next production.

3. Repeat step 2 until no more changes occur in First sets.

4. Initialize Follow set for the start symbol with '$' (end-of-input marker).

5. For each production P in the grammar:

a. For each non-terminal in P:

i. Add Follow of the left-hand side non-terminal to the Follow set of the current non-terminal.

ii. If epsilon is in First of the symbols following the current non-terminal, add Follow of the left-hand side non-terminal to the Follow set of the current non-terminal.

6. Repeat step 5 until no more changes occur in Follow sets.

7. The First and Follow sets are now calculated for the grammar.

**EXPT 7 (Recognize Strings)**

**THEORY**

The task of recognizing strings under specific regular expressions 'a', 'a\*b+', and 'abb' involves the development of a finite automaton that accepts or rejects input strings based on these patterns. Each of these regular expressions corresponds to different types of languages and can be recognized using different finite automata.

1. The regular expression 'a' represents the language that only accepts the string "a." It is the simplest regular expression, and its finite automaton consists of a single state, accepting "a" as the only valid input.

2. The regular expression 'a\*b+' represents a language that accepts strings containing zero or more 'a's followed by one or more 'b's. This corresponds to a non-deterministic finite automaton (NFA) where the automaton can either stay in the 'a' state or transition to the 'b' state after reading one 'a'. However, it must transition to the 'b' state after reading at least one 'a'.

3. The regular expression 'abb' represents a language that accepts the string "abb" only. This is a deterministic finite automaton (DFA) with three states: an initial state, a state for the first 'a', and a final state.

In summary, recognizing strings under these regular expressions requires finite automata with varying complexities, from a simple DFA for 'a' to more complex NFAs for 'a\*b+'. Understanding the theory of regular expressions and automata is essential for designing such recognizers. These automata can be used for pattern matching, lexical analysis, and validating input strings in various applications, such as compilers and text processing tools.

**ALGORITHM**

1. Start an infinite loop to continually receive user input strings.

2. Initialize a flag to 0.

3. Prompt the user to enter a string and read the input.

4. Remove the newline character from the input.

5. Check for each pattern:

a. If the input is empty, print "String recognized under 'a\*'" and set the flag to 1.

b. If the input consists of only 'a's, print "String recognized under 'a\*'" and set the flag to 1.

c. If the input is "abb," print "String recognized under 'abb'" and set the flag to 1.

d. If input has at least one 'b' and contains only 'a' and 'b', print "String recognized under 'a\*b+'" and set the flag to 1

**EXPT 8 (Recursive Descent Parsing)**

**THEORY**

A Recursive Descent Parser is a top-down parsing technique widely used in syntax analysis to validate and understand the structure of programming languages based on a given context-free grammar (CFG). This parsing method can be implemented directly from the production rules of the grammar, making it an intuitive and relatively straightforward approach.

At its core, the Recursive Descent Parser starts at the highest-level rule, often the start symbol of the CFG, and recursively applies production rules to parse and validate the input. Each non-terminal symbol is associated with a specific parsing function that corresponds to its production rule. These functions are invoked recursively as the parser processes the input tokens.

One of the key advantages of Recursive Descent Parsing is its direct correspondence to the CFG, which simplifies the transition from grammar rules to parsing code. However, the technique may face challenges with left-recursive or ambiguous grammars, which require additional handling techniques.

To create a Recursive Descent Parser, you need to construct parsing functions for each non-terminal in the grammar, making it a powerful tool for custom language design and implementation. Despite its limitations with certain grammars, Recursive Descent Parsing remains a valuable method for building parsers for a wide range of programming languages and domain-specific languages.

**ALGORITHM**

1. Define grammar rules for `E`, `E'`, `T`, `T'`, and `F`.

2. Read these rules from the user for the respective non-terminals.

3. Read the input string from the user.

4. Initialize a cursor to the start of the input string.

5. Start parsing the input by calling the function `E` with provided rules and pass the cursor.

6. Inside each non-terminal function (`E`, `E'`, `T`, `T'`, and `F`), check if the grammar rule matches the input.

7. If the rule matches, print the corresponding action, advance the cursor, and continue parsing.

8. If a non-terminal function returns `SUCCESS`, proceed to the next rule in the production.

9. If the cursor reaches the end of the input string and the parsing is successful, print "String is successfully parsed."

10. If there's any error in parsing, print "Error in parsing String."

**EXPT 9 (Shift Reduce Parsing)**

**THEORY**

Shift-Reduce Parsing is a bottom-up parsing technique widely used in syntax analysis for processing context-free grammars. It aims to construct a parse tree by repeatedly shifting input symbols onto a stack or reducing them based on a set of predefined production rules. This process continues until the parse tree is fully constructed, or an error is encountered.

The shift operation involves moving an input symbol onto the stack, representing a partial match to a grammar rule. In contrast, the reduce operation replaces a portion of the stack with a non-terminal, indicating the completion of a grammar rule. The decision to shift or reduce is based on a parsing table generated from the given grammar.

Shift-Reduce Parsing operates on a bottom-up strategy, meaning it starts with the input symbols and works towards the start symbol of the grammar. The stack holds a mix of terminals and non-terminals, reflecting the recognized portions of the input string and their corresponding grammar structures.

One popular variant of Shift-Reduce Parsing is LR (Lookahead Rightmost derivation) parsing, where 'L' stands for left-to-right scanning of the input, and 'R' denotes constructing a rightmost derivation in reverse order. LR parsers, including LALR and SLR parsers, are widely employed due to their efficiency and ability to handle a broad class of grammars.

Despite its efficiency, constructing a parsing table for Shift-Reduce Parsing can be complex, and conflicts may arise, leading to ambiguities in the grammar. Resolving these conflicts is crucial for ensuring accurate parsing of input strings. Overall, Shift-Reduce Parsing provides a powerful and flexible approach for syntactic analysis in compiler design and natural language processing applications.

**ALGORITHM**

1. Initialize: Set `input`, `stack` to empty strings, and `i` to 0.

2. Input Production Rules: Get `rule\_count` and production rules.

3. Parsing Loop: While true,

- If more chars in `input`, shift to `stack`. Print `stack`, remaining `input`, shift.

- For each production rule,

- If RHS matches substring in `stack`, reduce. Print `stack`, remaining `input`, reduce. Restart.

- If `stack` has start symbol and entire `input` processed, print "Accepted" and exit.

- If input processed but `stack` doesn't match start symbol, print "Not Accepted" and exit.

**EXPT 10 (Intermediate Code Generator)**

**THEORY**

Intermediate Code Generation is a crucial phase in compiler construction, serving as an intermediate representation between the high-level source code and the low-level target code. The primary goal is to simplify the source code into a form that is both closer to machine language and amenable to subsequent optimization. This phase contributes to enhancing the overall efficiency and portability of the compiler.

In this process, the compiler translates the source code, often in a high-level programming language, into an intermediate code that captures the essential semantics of the program. The generated intermediate code retains the key structural elements of the source code, facilitating easier analysis and transformation. One of the key advantages is that the intermediate code abstracts away language-specific details, allowing for the implementation of compiler optimizations independently of the source and target languages.

The intermediate code typically employs a set of instructions that closely resemble machine language, yet remain agnostic to the specifics of the underlying hardware. This facilitates the subsequent stages of optimization and target code generation. By separating the concerns of syntax and semantics during this phase, the compiler gains flexibility in adapting the intermediate code for diverse target architectures.

Overall, Intermediate Code Generation is an indispensable step in the compilation process, laying the foundation for subsequent analyses, optimizations, and the eventual generation of efficient machine code.

**ALGORITHM**

1. Accept the arithmetic expression as input.

2. Set `i` to 1, `j` to 0, and `tmpch` to 90.

3. Traverse the input to find operators like `:`, `/`, `\*`, `+`, and `-`. Store their positions and corresponding operators in the `operators` array.

4. -While there are operators to explore:

- Extract the left and right operands of the current operator position.

- Replace the current operator with a temporary character (decrement `tmpch`).

- Print the intermediate code for the operation.

- Move to the next operator.

5. - If no operators are left:

- Extract the left operand from the end of the input.

- Print the final assignment statement.

- If there are remaining operators, extract the right operand.

**EXPT 11 (Code Generation)**

**THEORY**

Code generation, a pivotal aspect of software development, is rooted in the concept of automating the creation of executable code from high-level representations. This process bridges the gap between abstract algorithmic descriptions, typically written in high-level programming languages, and the low-level machine code that computers execute. By automating this transition, code generation significantly enhances efficiency, consistency, and maintainability in software development.

The theoretical foundation of code generation draws inspiration from the broader field of compiler construction and programming language theory. Compilation involves several stages, including lexical analysis, syntax parsing, semantic analysis, optimization, and code generation. During the code generation phase, the compiler translates the intermediate representation of the source code into machine code or another high-level language. This process can be either ahead-of-time (AOT) or Just-In-Time (JIT), where AOT occurs before execution, and JIT happens dynamically during runtime.

The theoretical underpinnings of code generation encompass formal language grammars, abstract syntax trees, and optimization techniques. Techniques such as loop unrolling, inlining, and register allocation contribute to producing efficient and optimized code. Moreover, code generation principles are embedded in domain-specific languages and model-driven engineering approaches, providing higher levels of abstraction.

In essence, the theoretical framework for code generation revolves around transforming human-readable specifications into machine-executable instructions, embodying principles from formal language theory, compiler construction, and optimization strategies. This intersection empowers developers to focus on algorithmic logic and design, leaving the intricate details of machine code to automated processes, fostering productivity and code quality.

**ALGORITHM**

1. Input: Accepts a set of intermediate codes terminated by "exit."

2. Target Code Generation:

- Initialize variables and prompt for intermediate codes until "exit."

- For each intermediate code, analyze the operation and generate corresponding target code.

- Translate arithmetic operations (+, -, \*, /) into MOV and the respective operation (ADD, SUB, MUL, DIV) instructions.

- Output the target code for each operation: "Mov," operation, and destination.

3. Output: Display the generated target code for each intermediate code.