- **Names**
  - Erin Sorbella
  - Alex Chen
  - Steven Bristovitski
  - Michael Rourke
- **Date**
  - 03/20/2024
- **Course Number and details**
  - CS 361

Lexical Analyzer Report

- **Development environment/Languages**
  - The lexical analyzer was developed using Java in VSCode and uses the example.c and example2.c as the source files.
- **Files used:**
  - Example.c: This input takes an int a as input by a user and iterates through if statements whether the input number is even or odd.
  - Example2.c: This input takes two int numbers from a user as input, sums the two inputs into sum and prints it.
- **Description of the code Analyzer.Java**
  - *Package Imports*
    - Java.IO: Used here to process inputs and produce outputs
    - Java.util. *and Java.util.regex* :Are used here to import the common utilities used in Java such as ArrayLists, Data Structures and LinkedLists.
  - *Variables*
    - ArrayList < Token > tokenList: a list that holds all the tokens found in the source file.
  - *String Expressions*
    - defines the type of tokens
  - *Token Class Methods*
    - Type and value are declared for token types and values to be accepted in methods as input
    - Token Methods
      - Declares type and value properties
      - Returns both value and type

        - Uses toString method to print type and value easily.
    - *Tokenizer ArrayList method*
        - Accepts string input that will be used to for comparison with regular expressions.
        - combinedPattern combines all regular expressions into single pattern
        - Pattern and Matcher used to identify the string input and then matches it into a specific token type with the use of a while loop.
        - Adds each token into tokenList
    - *Main*
        - BufferReader method
            - Used to read the text from character input stream and buffer the characters that are read from the specified fileName that is input to be read from the command line.
        - Catch (IOException e)
            - used here during the input and output process when reading the file name. If the file is not found, the code will throw an exception that the file is not found.
        - For loop that prints all the found tokens within the source file
- **Description of the code example.c**
    - *Package imports*
        - stdio.h
            - import for accepting input and output
        - stdlib.h
            - general library for C to import memory allocation, conversions, and process control
    - *Main*
        - Variables
            - int a is the declared variable that will be used as input
        - The program scans for users int input and checks if the number is even or odd using a if loop
- **Description of the code example2.c**
    - *Main*
        - declares 3 ints that are 2 numbers and a sum
        - Asks the user to input 2 numbers
        - Sums the 2 numbers and prints the sum

Parser report

- **Files used**

- Example.c: This input takes an int a as input by a user and iterates through if statements whether the input number is even or odd.
- Example2.c: This input takes two int numbers from a user as input, sums the two inputs into sum and prints it.
- Token.java: Used to encapsulate the types of values that are tokenized by the analyzer.java file to provide a better structure when processing token types.

- **Description of Parser.java**
  - *Package imports*
    - import java.io.BufferedReader;
      - Used to read input from files by minimizing IO operations and storing in internal buffer
    - import java.io.FileReader;
      - Used to read data from files
    - import java.io.IOException;
      - Used to indicate that there is a problem during input/output processing.
    - import java.util.*;
      - used here to import the common utilities used in Java such as ArrayLists, Data Structures and LinkedLists.
  - Parser
    - *Methods*
      - Tokenizer arrayList
        - Takes a string input and analyzes it to identify lexical tokens that contain identifiers such as, keywords, identifiers and operators
      - Parse program
        - Parses the entire input code
      - Parse statement
        - Used to analyze and interpret the statements (if, if else, else) to define the control flow
      - Parse expression
        - analyzes and interprets expressions in the source code that is the input. It analyzes the tokens and determines the type
      - ParseAssigmentStatement
        - Used to analyze and interpret the statements used to assign variables their values in the input code
      - parseBlock
        - Used to interpret and analyze blocked statements (such as braces and brackets) within the input code that define the body of functions
      - isFunctionDecleration

        - Used to analyze and interpret the tokens found from the input code to determine whether the tokens are specifiers that are return types, identifiers and or parameters.
    - ParseVariableDecleration
        - Analyzes the token to check if its a variable declaration statements
    - parseFunctionDeclaration
        - Analyzes and interprets function declaration statements.
    - parseParameter
        - Analyzes and interprets parameters of a function
    - isBinaryOperator
        - Used to analyze and interpret if the token is a binary operator
    - parseFunctionCall
        - Used to check if the input code has call expressions that invoke specific arguments
    - parseEscapeCharacter
        - Analyzes and interprets string escape characters within the input code that control the representation of the output
    - Consume
        - A utility function used to match and check if the token fits the criteria of the value during the parsing
    - peek
        - Checks the token during the stream to see if the token matches the criteria without consuming it(accepting)it
    - isAtEnd
        - Checks to see if the parsing reached the end of the token stream.
- Main
    - Reads the input of the source code
    - Invokes the tokenizer to convert the source code into a stream of tokens
    - Handles the errors with IO Exceptions

Analyzer.java

```
package cs361.Analyzer;
import java.util.*;
import java.util.regex.*;
import java.io.*;

public class Analyzer {
    //maintain single list of tokens for a given program.
    public static ArrayList<Token> tokenList = new ArrayList<>();
```

```
    //regular expressions for each type of token.
    public static final String keywords =
"\\b(if|else|int|double|float|struct|char|long|for|while|return|switch|case)
\\b";
    public static final String identifiers = "[a-zA-Z_][a-zA-Z0-9_]*";
    public static final String operators = "\\+|-|\\*|/|%|=|==|!=|<|>|
<=|>=|&&|\\|\\|";
    public static final String punctuation = "[{}()\\[\\],;.]";
    public static final String constants = "\\b(\\d+\\.\\d+|\\d+\\.\\d*[eE]
[+-]?\\d+|\\d+[eE][+-]?\\d+|\\d+|'[^']*'|\"[^\"]*\")\\b";
    public static final String escapes = "\\\\.";

    //create Token class for each token found in
    //given program. Can return its type and value    //and contains a
toString() method.    public static class Token{
        private String type;
        private String value;

        public Token(String type, String value)
        {
            this.type = type;
            this.value = value;
        }
        public String getType()
        {
            return this.type;
        }
        public String getValue()
        {
            return this.value;
        }
        @Override
        public String toString() {
            return "TYPE: " + this.type + " VALUE: " + this.value + "\n";
        }
    }

    //tokenize the program line-by-line.
    //each token is identified by comparing against each regular expression.
public static ArrayList<Token> Tokenizer(String input)
    {
        String combinedPattern = String.join("|", keywords, identifiers,
operators,
        punctuation, constants, escapes);
        Pattern pattern = Pattern.compile(combinedPattern);
        Matcher matcher = pattern.matcher(input);

        while(matcher.find())
        {
            String tokenType = "";
            String tokenValue = matcher.group();
            if (tokenValue.matches(keywords)) {
```

```
                tokenType = "Keyword";
            } else if (tokenValue.matches(identifiers)) {
                tokenType = "Identifier";
            } else if (tokenValue.matches(operators)) {
                tokenType = "Operator";
            } else if (tokenValue.matches(punctuation)) {
                tokenType = "Punctuation";
            } else if (tokenValue.matches(constants)) {
                tokenType = "Constant";
            } else if (tokenValue.matches(escapes)) {
                tokenType = "Escape Character";
            } else {
                System.out.println("Invalid token type for: " + tokenValue);
            }

            //add each token to overall token list
            tokenList.add(new Token(tokenType, tokenValue));
        }
        return tokenList;
    }

    //Main function. Reads file "example.c" and parses each line
    //through Tokenizer.    public static void main(String[] args)
    {
        String file = args[0];
        BufferedReader reader;
        try {
            String line;
            reader = new BufferedReader(new FileReader(file));
            while ((line = reader.readLine()) != null)
            {
                Tokenizer(line);
            }
            reader.close();
        }
        catch (IOException e) {
            System.out.println("File not found.");
        }
        for (Token token: tokenList)
        {
            System.out.print(token);
        }
    }
}
```

Example.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
```

```
    int a;
    printf("Enter whole number:\n");
    scanf("%d", &a);

    if (a % 2 == 0)
        printf("c is even.\n");
    else
        printf("c is odd.\n");
    return 0;
}
```

Example2.c

```
#include <stdio.h>

int main() {
    //Find sum of two numbers
    int num1, num2, sum;

    printf("Enter the first integer: ");
    scanf("%d", &num1);

    printf("Enter the second integer: ");
    scanf("%d", &num2);

    sum = num1 + num2;

    printf("The sum of %d and %d is %d\n", num1, num2, sum);

    return 0;
}
```

Parser.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class Parser extends Analyzer {
    private List<Token> tokens;
    private int position;

    public Parser(List<Token> tokens) {
        this.tokens = tokens;
        this.position = 0;
    }

    // Parse the entire program
```

```
    public void parseProgram() {
        while (!isAtEnd()) {
            parseStatement();
        }
    }

    private void parseStatement() {
        Token token = peek();
        if (token.getType().equals("Keyword") &&
token.getValue().equals("int")) {
            // Check if it's a function declaration
            if (isFunctionDeclaration()) {
                parseFunctionDeclaration();
            } else {
                parseVariableDeclaration();
            }
        } else if (token.getType().equals("Identifier") && (position + 1 <
tokens.size()) && tokens.get(position + 1).getValue().equals("(")) {
            // If the next token is an opening parenthesis, it's a function
call
            parseFunctionCall(token);
        } else if (token.getType().equals("Keyword") &&
token.getValue().equals("if")) {
            // If it's an "if" statement, parse the conditional expression
and the block
            consume("Keyword", "if");
            consume("Punctuation", "(");
            parseExpression();
            consume("Punctuation", ")"); // Ensure to consume the closing
parenthesis after the conditional expression
            parseBlock(); // Parse the main block
                // Check for an optional "else" block
            if (peek().getType().equals("Keyword") &&
peek().getValue().equals("else")) {
                consume("Keyword", "else");
                parseBlock(); // Parse the else block
            }
        } else if (token.getType().equals("Keyword") &&
token.getValue().equals("return")) {
            // If it's a "return" statement, parse the expression (if
present)
            consume("Keyword", "return");
            if (!peek().getValue().equals(";")) {
                parseExpression();
            }
            Token semicolon = consume("Punctuation", ";"); // Expect a
semicolon after the return statement
            System.out.println("Parsed return statement");
        } else {
            // If none of the above, it could be an assignment statement
            parseAssignmentStatement();
        }
    }
```

```
    // Parse an expression
    private void parseExpression() {
        Token token = peek();
        if (token.getType().equals("Identifier") && (position + 1 <
tokens.size()) && tokens.get(position + 1).getValue().equals("(")) {
            parseFunctionCall(token);
        } else if (token.getType().equals("Constant")) {
            consume("Constant", null);
            System.out.println("Parsed constant expression with value " +
token.getValue());
        } else if (token.getType().equals("Identifier")) {
            consume("Identifier", null);
            System.out.println("Parsed identifier expression with value " +
token.getValue());
        } else if (token.getType().equals("Operator")) {
            // Handle different operators
            String operatorValue = token.getValue();
            if (isBinaryOperator(operatorValue)) {
                consume("Operator", operatorValue);
            } else {
                throw new ParseException("Unexpected unary operator: " +
operatorValue);
            }
        } else {
            throw new ParseException("Expected expression, but found token:
" + token);
        }
    }

    // Parse an assignment statement
    private void parseAssignmentStatement() {
        Token identifier = consume("Identifier", null);
        consume("Operator", "="); // Consume the assignment operator
        parseExpression(); // Parse the expression on the right-hand side
                // Loop until a semicolon is encountered, consuming any
additional tokens
        while (!peek().getValue().equals(";")) {
            // Consume the next token
            consume(peek().getType(), peek().getValue());
        }

        Token semicolon = consume("Punctuation", ";"); // Expect a semicolon
after the assignment statement
        System.out.println("Parsed assignment statement for " +
identifier.getValue());
    }

    // Parse a block of statements
    private void parseBlock() {
        // Parse the block until a closing brace '}'
        consume("Punctuation", "{");
        while (!peek().getValue().equals("}")) {
```

```
            parseStatement();
        }
        consume("Punctuation", "}"); // End of the block
    }

    private boolean isFunctionDeclaration() {
        int currentPosition = position; // Save current position
        try {
            // Check if the next tokens match the pattern for a function
declaration
            Token currentToken = tokens.get(currentPosition);
            Token nextToken = tokens.get(currentPosition + 1);
            Token thirdToken = tokens.get(currentPosition + 2);

            return currentToken.getType().equals("Keyword") &&
                    currentToken.getValue().equals("int") &&
                    nextToken.getType().equals("Identifier") &&
                    thirdToken.getType().equals("Punctuation") &&
                    thirdToken.getValue().equals("(");
        } catch (IndexOutOfBoundsException e) {
            // If there aren't enough tokens to check, it's not a function
declaration
            return false;
        } finally {
            position = currentPosition; // Restore the position
        }
    }

    // Parse a variable declaration (e.g., "int x = 5;")
    private void parseVariableDeclaration() {
        consume("Keyword", "int");
        Token identifier = consume("Identifier", null);

        // Check for optional assignment
        if (peek().getValue().equals("=")) {
            consume("Operator", "=");
            parseExpression();
        }

        consume("Punctuation", ";");

        System.out.println("Parsed variable declaration for " +
identifier.getValue());
    }

    // Parse a function declaration
    private void parseFunctionDeclaration() {
        consume("Keyword", null); // Accept any return type
        Token identifier = consume("Identifier", null);
        consume("Punctuation", "(");

        // Parse parameters
        while (!peek().getValue().equals(")")) {
```

```
            parseParameter();
            // Check for comma to separate parameters
            if (peek().getValue().equals(",")) {
                consume("Punctuation", ",");
            }
        }

        consume("Punctuation", ")"); // End of parameter list
        consume("Punctuation", "{"); // Start of function body

        // Parse function body until closing brace '}'        while
(!peek().getValue().equals("}")) {
            parseStatement();
        }

        consume("Punctuation", "}"); // End of function body

        System.out.println("Parsed function declaration for " +
identifier.getValue());
    }


    // Parse a function parameter
    private void parseParameter() {
        // For simplicity, let's assume parameters are of the form "Type
Identifier"
        consume("Keyword", null); // Accept any type
        consume("Identifier", null);
    }

    // Check if the operator is a binary operator
    private boolean isBinaryOperator(String operator) {
        return operator.equals("+") || operator.equals("-") ||
operator.equals("*") || operator.equals("/") || operator.equals("%");
    }

    // Parse a function call
    private void parseFunctionCall(Token identifierToken) {
        consume("Identifier", identifierToken.getValue()); // Consume the
function identifier
        consume("Punctuation", "("); // Expect opening parenthesis for
function call

        // Parse arguments (if any)        while
(!peek().getValue().equals(")")) {
            if (peek().getType().equals("Escape Character")) {
                // If an escape character is encountered, consume it and
continue
                parseEscapeCharacter();
            } else {
                parseExpression();
            }
            // Check for comma to separate arguments
```

```
            if (peek().getValue().equals(",")) {
                consume("Punctuation", ",");
            }
        }

        consume("Punctuation", ")"); // Expect closing parenthesis for
function call
        Token semicolon = consume("Punctuation", ";"); // Expect a semicolon
after a function call
        System.out.println("Consumed semicolon: " + semicolon);
    }

    // Parse an escape character
    private void parseEscapeCharacter() {
        Token escapeChar = consume("Escape Character", null); // Consume the
escape character
        System.out.println("Parsed escape character with value " +
escapeChar.getValue());
    }


    // Consume a token of a specific type and value (if value is not null)
    private Token consume(String type, String value) {
        Token token = peek();
        if (token.getType().equals(type) && (value == null ||
token.getValue().equals(value))) {
            position++;
            return token;
        }
        throw new ParseException("Expected token of type " + type + " with
value " + value + " but got " + token);
    }

    // Peek at the current token without consuming it
    private Token peek() {
        if (isAtEnd()) {
            throw new ParseException("Reached end of token list");
        }
        return tokens.get(position);
    }

    // Check if we've reached the end of the token list
    private boolean isAtEnd() {
        return position >= tokens.size();
    }

    public static void main(String[] args) {
        String file = args[0];
        BufferedReader reader;
        try {
            reader = new BufferedReader(new FileReader(file));
            String line;
            while ((line = reader.readLine()) != null) {
```

```
                Tokenizer(line);
            }
            reader.close();
        } catch (IOException e) {
            System.out.println("File not found.");
        }

        Parser parser = new Parser(tokenList);
        parser.parseProgram();
    }

    // Custom exception class for parse errors
    static class ParseException extends RuntimeException {
        public ParseException(String message) {
            super(message);
        }
    }
}
```