# Semantic Analyzer Report

- **Title**
  - Semantic analyzer
- **Names**
  - Erin Sorbella
  - Alex Chen
  - Steven Bristovitski
  - Michael Rourke
  - Dana Robertson
- **Date**
  - 04/25/2024
- **Course Number and details**
  - CS 361
- **Development environment/Languages**
  - The Semantic 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 Parser.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.
    - Position: Integer that is used to track the place in the token list
    - symbolTable: A hash map that is used to link variables to their types
  - String Expressions
    - defines the type of tokens
  - parseProgram
    - Iterates through the tokens and calls on parseStatement for each token

- parseStatement
  - parses statements depending on each token found
- parseExpression
  - handles parsing expressions that are constants, string literals, operators. it recurses for compound expressions
- parseAssignment
  - parses assignment statements that are assigned and makes sure that they are declared
- isTypeMatch
  - checks if two tokens match each other
- parseBlock
  - Used to parse block statement and calls on parseStatement method until end of curly braces is reached
- isFunctionDecleration
  - Checks ahead of token to see if tokens are a function decleration
- parseVariableDecleration
  - Is a type identifier and updates symbolTable with new incoming variables
- parseFunctionDecleration
  - parses function declarations that are return types, function names, parameters. Used to encapsulate all statements within the code block
- parseParameter
  - parses function parameters that are a type identifier
- parseFunctionCall
  - parses function call to handle arguments and ensures the that there are closing parenthesis with semicolon
- parseEscapeCharacter
  - consumes escape character token
- consume
  - Consumes tokens of specified value
- peek
  - Returns current token list without continuing to next
- isAtEnd
  - Checks if end of token list has been reached
- SemanticException
  - Custom exception for handling specific errors such undeclared variable type or type mismatches
- Main

        - Reads file line by line and tokenizes lines by creating a parser object that parses the tokens
        - BufferReader method
            - Used to read the text from character input stream and buffer the characters that are read from the specified fileName (In this case: example.c)
        - 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

```
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;
    private Map<String, String> symbolTable;

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

```
        this.symbolTable = new HashMap<>();
    }

    // Parse the entire program
    public void parseProgram() {
        while (!isAtEnd()) {
            parseStatement();
        }
    }

    private void parseStatement() {
    Token token = peek();
    if (isFunctionDeclaration()) {
        parseFunctionDeclaration();
    } else if (token.getType().equals("Keyword") &&
token.getValue().equals("int") || token.getType().equals("Keyword") &&
token.getValue().equals("double")) {
        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 Token parseExpression() {
        Token token = peek();
        if (token.getType().equals("Constant")) {
            consume("Constant", null);
            System.out.println("Parsed constant expression with value " +
token.getValue());
        } else if (token.getType().equals("Identifier")) {
            String varName = token.getValue();
            if (!symbolTable.containsKey(varName)) {
                throw new SemanticException("Undeclared variable '" +
varName + "'");
            }
            consume("Identifier", null);
            System.out.println("Parsed identifier expression with value " +
varName);
        } else if (token.getType().equals("String")) {
            consume("String", null);
            System.out.println("Parsed string literal: " +
token.getValue());
        } else if (token.getType().equals("Operator") &&
token.getValue().equals("+")) {
            consume("Operator", "+");
            parseExpression();
            parseExpression();
            System.out.println("Parsed addition expression");
        } else {
            throw new SemanticException("Unsupported expression");
        }
        return token;
    }

    // Parse an assignment statement
    private void parseAssignmentStatement() {
        Token identifier = consume("Identifier", null);

        // Check if the identifier has been declared
        if (!symbolTable.containsKey(identifier.getValue())) {
            throw new SemanticException("Variable '" + identifier.getValue()
+ "' not declared");
        }

        consume("Operator", "="); // Consume the assignment operator
        Token expressionToken = parseExpression(); // Parse the expression
on the right-hand side
                // Ensure that the remaining tokens in the assignment
statement are correctly matched
        while (!peek().getValue().equals(";")) {
            // Consume the next token
            Token nextToken = peek();
            if (nextToken.getType().equals("Operator")) {
                consume("Operator", nextToken.getValue());
```

```
            } else if (nextToken.getType().equals("Identifier")) {
                consume("Identifier", nextToken.getValue());
            } else if (nextToken.getType().equals("Constant")) {
                consume("Constant", nextToken.getValue());
            } else {
                throw new SemanticException("Unexpected token in assignment
statement");
            }
        }

        // Check for type mismatch
        if (!isTypeMatch(identifier, expressionToken)) {
            throw new SemanticException("Type mismatch: " +
symbolTable.get(identifier.getValue()) + " and " +
symbolTable.get(expressionToken.getValue()));
        }

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


    // Check if the type of the identifier matches the type of the
expression
    private boolean isTypeMatch(Token identifier, Token expressionToken) {
        //System.out.println(identifier.getValue());
        if(expressionToken.getType().equals("Constant"))
            return true;
        if
(symbolTable.get(identifier.getValue()).equals(symbolTable.get(expressionTok
en.getValue())))
            return true;
        return false;
    }

    // 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() {
        String type = consume("Keyword", null).getValue(); // Accept any
type (int or double)
        Token identifier = consume("Identifier", null);

        // Check for duplicate variable declaration
        if (symbolTable.containsKey(identifier.getValue())) {
            throw new SemanticException("Variable '" + identifier.getValue()
+ "' already declared");
        }

        symbolTable.put(identifier.getValue(), type);

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

        consume("Punctuation", ";");

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

    // 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();
    }

    static class SemanticException extends RuntimeException {
        public SemanticException(String message) {
            super(message);
        }
    }

    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);
        }
    }
}
```