CS323 Documentation

1. Problem Statement

This program is a simple lexer that reads a file and identifies the different tokens in the file. It uses several functions to perform different tasks such as opening and closing files, initializing and finalizing the output file, checking if a value in an array matches the value passed to the function, and defining finite state machines (FSMs) for different token types.

2. How to use Our program

Step1) Unzip file (a2.syntaxAnalyzer)
Step2) Click and run main.exe
Step3) When prompted, enter the input file (test1.rat23s)
Step4) When prompted, enter the output file (saOutput1.txt)
Step5) Results will be displayed in terminal and saOutput1.txt

3. Design of Our program

Repository: https://github.com/MichaelR13/cpsc323.Project

Functions:

- **vector<TokenType> parseTokens(ifstream &testInput)**
  - The following was used in **lexer.h** to help store the token/lexeme pairs into a vector of TokenType elements.
- **void tokenListHelper(ifstream& lexerOutput)**
  - This function helps populate tokenList with the tokens/lexemes from the ParseTokens function
- void openSyntaxFile() , void closeSyntaxFile()
  - Handle the file management of the syntax analyzer
- class SA
  - Contains the various functions for each syntax rule
- The following are the functions for the syntax rules
  - **void SA::Rat23S();**
  - **void SA::OptFunctionDefinitions();**
  - **void SA::FunctionDefinitions();**
  - **void SA::Function();**
  - **void SA::OptParameterList();**
  - **void SA::ParameterList();**
  - **void SA::Parameter();**
  - **void SA::Qualifier();**
  - **void SA::Body();**
  - **void SA::OptDeclarationList();**
  - **void SA::DeclarationList();**
  - **void SA::Declaration();**

- **void SA::IDs();**
- **void SA::StatementList();**
- **void SA::Statement();**
- **void SA::Compound();**
- **void SA::Assign();**
- **void SA::If();**
- **void SA::Return();**
- **void SA::Print();**
- **void SA::Scan();**
- **void SA::While();**
- **void SA::Condition();**
- **void SA::Relop();**
- **void SA::Expression();**
- **void SA::Term();**
- **void SA::Factor();**
- **void SA::Primary();**
- **void SA::Empty();**
- **void SA::GetNextToken();**
- **void SA::ExpressionPrime();**
  - **Extra functions due to left recursion**
- **void SA::TermPrime();**
  - **Extra functions due to left recursion**

4. **Any Limitation**
   It is only capable of analyzing some source code; Various source code files cause numerous bugs that do not properly traverse/generate the syntax trees. An example of this is in **test2.rat23s** , where the *while (true)* condition unwillingly causes a syntax error.

5. **Any shortcomings**
   Debugging was difficult, as the longer source code was, the larger the syntax trees tended to be, making it difficult to pinpoint exactly was triggering various unwanted syntax errors.

Samples:
Here is an example input and the corresponding output that can be produced using the functions in the provided code:

Source code:

```
function main()
{
   a = b + c;
   return;
} #
```

Syntax Analyzer output:

Token: identifier Lexeme: function
<Rat23S> ::= <Opt Function Definitions> # <Opt Declaration List> # <Statement List>
<Opt Function Definitions> ::= <Function Definitions> | <Empty>
<Function Definitions> ::= <Function> | <Function> <Function Definitions>
<Function> ::= function <identifier> (<Opt Parameter List>) <Opt Declaration List>
<Body>

Token: identifier Lexeme: main

Token: seperator Lexeme: (

Token: seperator Lexeme:)
<Opt Parameter List> ::= <Parameter List> | <Empty>
<Empty> ::= epsilon

Token: seperator Lexeme: {
<Opt Declaration List> ::= <Declaration List> | <Empty>
<Empty> ::= epsilon
<Body> ::= { < Statement List> }

Token: identifier Lexeme: a
<Statement List> ::= <Statement> | <Statement> <Statement List>
<Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan>
<Assign> ::= <identifier> = <Expression> ;

Token: operator Lexeme: =

Token: identifier Lexeme: b
<Expression> ::= <Term> <Expression Prime>
<Term> ::= <Factor> <Term Prime>
<Factor> ::= - <Primary> | <Primary>
<Primary> ::= <identifier> | <Integer> | <identifier> (<IDs>) | (<Expression>) |
<Real> | true | false

Token: seperator Lexeme: +
<Term Prime> ::= * <Factor> <Term Prime> | / <Factor> <Term Prime> | <Empty>
<Empty> ::= epsilon
<Expression Prime> ::= + <Term> <Expression Prime> | - <Term> <Expression Prime> |
<Empty>

Token: identifier Lexeme: c
<Term> ::= <Factor> <Term Prime>
<Factor> ::= - <Primary> | <Primary>

<Primary> ::= <identifier> | <Integer> | <identifier> (<IDs>) | (<Expression>) |
<Real> | true | false

Token: seperator Lexeme: ;
<Term Prime> ::= * <Factor> <Term Prime> | / <Factor> <Term Prime> | <Empty>
<Empty> ::= epsilon
<Expression Prime> ::= + <Term> <Expression Prime> | - <Term> <Expression Prime> |
<Empty>
<Empty> ::= epsilon

Token: identifier Lexeme: return
<Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan>
<Assign> ::= <identifier> = <Expression> ;

Token: seperator Lexeme: ;

Token: seperator Lexeme: }

Token: seperator Lexeme: #
<Opt Declaration List> ::= <Declaration List> | <Empty>
<Empty> ::= epsilon
<Statement List> ::= <Statement> | <Statement> <Statement List>

Syntax Accepted