CS323 Assignment Documentation
<should consists of about 2-3 pages>

1. **Problem Statement**
   Construct a syntax analyzer using a Recursive-Decent Parser(RDP)

2. **How to use your program**
   Parsley is the name of the executable file. In order to run a lexical analysis and parse a file for syntax analysis according to the RAT16F language provide the file name as an argument in the command line when running *Parsley.* Proper format follows:

   ./Parsley *yourfile.txt*

   After execution *Parsley* will create a new file in the current directory labeled:

   parser*_yourfile.txt*

   This file will then contain all production rules utilized when parsing *yourfile.txt.*

3. **Design of your program**
   I choose to go with the Recursive-Decent Parser for its simplicity. I also chose to utilize a dynamic 2D array to house the tokens and lexemes.

   All on the production rules follow and have been modified for left recursion:

<Rat16F> ::= $$ <Opt Function Definitions>
$$ <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>
<Opt Parameter List> ::= <Parameter List> | <Empty>
<Parameter List> ::= <Parameter> | <Parameter> , <Parameter List>
<Parameter> ::= <IDs > : <Qualifier>
<Qualifier> ::= integer | boolean | real
<Body> ::= { < Statement List> }
<Opt Declaration List> ::= <Declaration List> | <Empty>
<Declaration List> := <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> ::= <Qualifier > <IDs>
<IDs> ::= <Identifier> | <Identifier>, <IDs>
<Statement List> ::= <Statement> | <Statement> <Statement List>
<Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> |
<While>
<Compound> ::= { <Statement List> }
<Assign> ::= <Identifier> := <Expression> ;
<If> ::= if (<Condition>) <Statement> endif |

if (<Condition>) <Statement> else <Statement> endif
<Return> ::= return ; | return <Expression> ;
<Write> ::= print (<Expression>);
<Read> ::= read (<IDs>);
<While> ::= while (<Condition>) <Statement>
<Condition> ::= <Expression> <Relop> <Expression>
<Relop> ::= = | /= | > | < | => | <=
<Factor> ::= - <Primary> | <Primary>
<Primary> ::= <Identifier> | <Integer> | <Identifier> [<IDs>] |
(<Expression>) | <Real> | true | false
<Empty> ::= epsilon

Left Recursive:
<Expression> ::= <Expression> + <Term> | <Expression> - <Term> | <Term>

<Term> ::= <Term> * <Factor> | <Term> / <Factor> | <Factor>

Rewriting it without left recursion:
<Expression> ::= <Term> <Expression Prime>
<Expression Prime> ::= +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

<Term> ::= <Factor> <Term Prime>
<Term Prime> ::= * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon

4. **Any Limitation**
The only limitations for the program are that once an error is encountered the program does not recover and instead exits.

5. **Any shortcomings**
I was not able to get the line number in the source program file, instead when an error occurs the line number refers the line number in the lexer file.