COVER PAGE

CS323 Programming Assignments

1. Your Name:	**PHUC LE**
2. Assignment Number:	2
3. Due Date	**Sunday, November, 6, 2016**
4. Turn-In Date	**Sunday, November, 6, 2016**
5. Executable FileName:	**Syntax Analyzer.exe**
6. LabRoom	**CS-200, PC: 1a**
7. OS	**Windows 10**

GRADE:

COMMENTS:

1. Problem Statement:

This project is to use RDP method to implement a top-down parser as a syntax analyzer which input is a file containing Rat16F source code and output is a file containing series of records. Each record will be represented by its token, lexeme, and its production rules.

2. How to Use My Program:

Simply click on "**Syntax Analyzer.exe**" in **Executable** folder, then click on 

to select the source code file; the program will display the result and write down the result to an output file.

In case the program doesn't start, there are three alternative methods to run my program:

- Method #1: copy folder **Executable** to desktop or anywhere else. Next, get in to Windows Command line, use cd command to change to the directory to **Executable**, and finally type "**java –jar app.jar**"
- Method #2: Open Netbeans IDE, open project "**SyntaxAnalyzer_PhucLe**," open file "**src\Program.java**" and click on green button  or press **Shift-F6** to run my program
- Method #3: double click on **app.jar** file.

3. Designing of My Program:

After removing left recursions and back-tracking, the productions rules are the followings:

<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> <If Prime>

<If Prime> → endif | else <Statement> endif

<Return> → return ; | return <Expression> ;

<Write> → print (<Expression>);

<Read> → read (<IDs>);

<While> → while (<Condition>) <Statement>

<Condition> → <Expression> <Relop> <Expression>

<Relop> → = | /= | > | < | => | <=

<Expression> → <Term> <Expression Prime>

<Expression Prime> → + <Term><Expression Prime> | - <Term><Expression Prime> | <empty>

<Term> → <Factor> <Term Prime>

<Term Prime> → * <Factor> <Term Prime> | / <Factor> <Term Prime> | <empty>

<Factor> → <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false | - <Factor>

22<Empty> → ε

4. **Limitation:** The error messages needs to be more detailed
5. **Shortcoming:** None

*Source code of my program is mainly found in file **SyntaxAnalyzer_PhucLe\src\SyntaxAnalyser.java***

```
import java.util.ArrayList;
import java.util.List;
public class SyntaxAnalyser {
    LexicalAnalyser pro = new LexicalAnalyser();
    int nextTokenPosition = 0;
    Token currentToken = null;
    List<Token> tokens = new ArrayList<>();
    String output = "";
    String errorMessage = "";
    boolean successful = false;
    int errorLoc = -1;
    Token errorToken = new Token(LexicalAnalyser.Type.IDENTIFIER, "error", 0);

    public SyntaxAnalyser() {
    }

    public void input(String source) {
        tokens.clear();
        tokens.addAll(pro.tokenize(source, 1));
    }

    public void reset() {
        nextTokenPosition = 0;
        currentToken = null;
        output = "";
        successful = false;
        errorLoc = -1;
        errorMessage = "";
        errorToken = new Token(LexicalAnalyser.Type.IDENTIFIER, "error", 0);
    }
```

```
public void clearTokens() {
    tokens.clear();
}

public void input(String source, int lineNumber) {
    tokens.addAll(pro.tokenize(source, lineNumber));
}

public void lexer() {
    if (tokens.size() > 0 && nextTokenPosition < tokens.size()) {
        currentToken = tokens.get(nextTokenPosition);
        if (currentToken.tokenType == LexicalAnalyser.Type.ERROR) {
            errorLoc = nextTokenPosition - 1;
            errorToken = currentToken;
            errorMessage = "UNVALID TOKEN " + currentToken.content;
        }
        nextTokenPosition++;
    } else {
        currentToken = null;
    }
}

public String getResult() {
    String result = "";
    if (errorLoc >= 0) {
        for (int i = 0; i <= errorLoc; i++) {
            result += tokens.get(i).toString2() + "\n";
        }
        result += "ERROR AT LINE " + errorToken.lineNumber + ": " + errorMessage;
        if (errorToken.tokenType == LexicalAnalyser.Type.ERROR) {
            result += ", UNVALID TOKEN: " + errorToken.content;
        }
```

```
        } else {
            for (Token token : tokens) {
                result += token.toString2() + "\n";
            }
            if (this.successful) {
                result += "THE SYNTAX IS CORRECT";

            }
        }

        return result;
    }

    private boolean isToken(String token) {
        if (currentToken != null) {
            return currentToken.content.trim().equals(token);
        }
        return false;
    }

   boolean func_rat16f() {
        lexer();
        if (isToken("$$")) {
            currentToken.addRule("<Rat16F> -> $$ <Opt Function Definitions> $$ <Opt Declaration
List> <Statement List> $$ ");
            lexer();
            if (isToken("$$")) {
                currentToken.addRule("<Opt Function Definitions> -> <Empty>");

                lexer();
                if (currentToken.content.equals("integer") ||
currentToken.content.equals("real") || currentToken.content.equals("boolean")) {
                    currentToken.addRule("<Opt Declaration List> -> <Declaration List>");
```

```
            } else {
                currentToken.addRule("<Opt Declaration List> -> <Empty>");
            }
            nextTokenPosition--;

            if (func_opt_declaration_list()) {
                if (func_statementList()) {
                    lexer();
                    if (isToken("$$")) {
                        lexer();
                        successful = true;
                        return true;
                    } else {

                        return false;
                    }
                } else {
                    return false;
                }
            } else {
                return false;
            }

        } else {
            currentToken.addRule("<Opt Function Definitions> -> <Function Definitions>");
            nextTokenPosition--;
            if (func_opt_function_definitions()) {
                lexer();
                if (isToken("$$")) {
                    lexer();
                    if (!(currentToken.content.equals("integer") ||
currentToken.content.equals("real") || currentToken.content.equals("boolean"))) {
                        currentToken.addRule("<Opt Declaration List> -> <Empty>");
```

```
                    }
                    nextTokenPosition--;

                    if (func_opt_declaration_list()) {
                        if (func_statementList()) {
                            lexer();
                            if (isToken("$$")) {
                                successful = true;
                                return true;
                            } else {
                                errorToken = currentToken;
                                return false;
                            }
                        } else {
                            return false;
                        }
                    } else {
                        return false;
                    }
                }
            }
        } else {
            errorLoc = 0;
            errorToken = currentToken;
            errorMessage = "$$ is expected at the begining of the source code";
            return false;
        }
        return false;
    }

boolean func_opt_function_definitions() {
```

```
    lexer();
    if (isToken("$$")) {
        currentToken.addRule("<Opt Function Definitions> -><Empty>");
        nextTokenPosition--;
        return true;
    } else {
        nextTokenPosition--;
        return func_function_definitions();
    }
}

boolean func_function_definitions() {
    int pointer = nextTokenPosition;
    if (func_function()) {

        lexer();
        if (isToken("$$")) {
            nextTokenPosition--;
            return true;
        } else {
            nextTokenPosition--;
            return func_function_definitions();
        }
    } else {
        errorLoc = pointer;
        errorToken = tokens.get(pointer);
        errorMessage = "Function Definition is incorrect";
        return false;
    }
}

boolean func_function() {
```

```
        lexer();
        if (isToken("function")) {
            currentToken.addRule("<Function> -> function <Identifier> [<Opt Parameter List>]
<Opt Declaration List> <Body>");
            lexer();
            if (currentToken.tokenType == LexicalAnalyser.Type.IDENTIFIER) {
                lexer();
                if (isToken("[")) {
                    currentToken.addRule("<Opt Parameter List> -> <Parameter List> | <Empty>");
                    lexer();
                    if (isToken("]")) {
                        currentToken.addRule("<Opt Parameter List> -> <Empty>");
                        return func_opt_declaration_list() && func_body();
                    } else {
                        currentToken.addRule("<Opt Parameter List> -> <Parameter List>");
                        nextTokenPosition--;
                        return func_parameter_list() && func_opt_declaration_list() &&
func_body();
                    }
                } else {
                    nextTokenPosition--;
                    return false;
                }
            } else {
                errorLoc = nextTokenPosition;
                errorToken = currentToken;
                errorMessage = "expecting an identifier";
                nextTokenPosition--;
                return false;
            }
        } else {
            errorLoc = nextTokenPosition;
            errorToken = currentToken;
```

```
        errorMessage = "expecting 'function'";
        nextTokenPosition--;
        return false;
    }
}

boolean func_opt_parameter_list() {
    lexer();
    if (isToken("]")) {
        nextTokenPosition--;
        return true;
    } else {
        nextTokenPosition--;
        return func_parameter_list();
    }
}

boolean func_parameter_list() {
    int pointer = nextTokenPosition;
    boolean isParameter = func_parameter();
    if (isParameter) {
        lexer();
        if (isToken("]")) {
            nextTokenPosition--;
            return true;
        } else {
            nextTokenPosition--;
            lexer();
            if (isToken(",")) {
                return func_parameter_list();
            } else {
                nextTokenPosition--;
```

```
                return false;
            }
        }
    } else {
        nextTokenPosition = pointer;
    }
    return false;
}

boolean func_parameter() {
    int pointer = nextTokenPosition;
    if (func_IDs()) {
        currentToken.addRule("<Parameter> -> <IDs> : <Qualifier>");
        lexer();
        if (currentToken == null) {
            nextTokenPosition--;
            return false;
        }
        if (isToken(":")) {
            return func_qualifier();
        } else {
            nextTokenPosition--;
            return false;
        }
    } else {
        nextTokenPosition = pointer;
    }
    return false;
}

boolean func_qualifier() {
    lexer();
```

```
        if (currentToken == null) {
            nextTokenPosition--;
            return false;
        }
        if (isToken("integer") || isToken("boolean") || isToken("real")) {
            currentToken.addRule("<Qualifier> -> " + currentToken.content);
            return true;
        } else {
            nextTokenPosition--;
            return false;
        }
    }

    boolean func_body() {
        return func_compound();
    }

    boolean func_opt_declaration_list() {
        lexer();
        int pointer = nextTokenPosition;
        if (isToken("]") || isToken("$$"))
        {
            lexer();
        }
        if (isToken("{") || currentToken.tokenType == LexicalAnalyser.Type.IDENTIFIER ||
isToken("if") || isToken("return") || isToken("print") || isToken("read") || isToken("while")) {
            if (isToken("{")) {
                addRule("<Opt Declaration List> - > <Empty>", pointer);
            }
            nextTokenPosition--;
            return true;
        } else {
            currentToken.addRule("<Opt Declaration List> - > <Declaration List>");
```

```
            nextTokenPosition--;
            return func_declaration_list();
        }
    }

    boolean func_declaration_list() {
        if (func_declaration()) {
            lexer();
            if (isToken(";")) {
                lexer();
                if (isToken("{") || currentToken.tokenType == LexicalAnalyser.Type.IDENTIFIER ||
isToken("if") || isToken("return") || isToken("print") || isToken("read") || isToken("while")) {
                    nextTokenPosition--;
                    return true;
                } else {
                    nextTokenPosition--;
                    return func_declaration_list();
                }
            } else {
                nextTokenPosition--;
                return false;
            }
        }
        return false;
    }

    boolean func_declaration() {
        currentToken.addRule("<Declaration> -> <Qualifier> <IDs>");
        return func_qualifier() && func_IDs();
    }

    boolean func_IDs() {
        int pointer = nextTokenPosition;
```

```
    lexer();
    if (currentToken == null) {
        return false;
    }
    if (currentToken.tokenType == LexicalAnalyser.Type.IDENTIFIER) {
        lexer();
        if (currentToken == null) {
            return true;
        }
        if (isToken(")") || isToken(";") || isToken(":") || isToken("]")) {
            currentToken.addRule("<IDs> -> <Identifier>");
            nextTokenPosition--;
            return true;
        } else if (isToken(",")) {
            currentToken.addRule("<IDs> -> <Identifier> , <IDs>");
            //lexer();
            return func_IDs();
        } else {
            nextTokenPosition--;
        }
    } else {
        nextTokenPosition = pointer;
    }
    return false;
}

boolean func_statementList() {
    int pointer = nextTokenPosition;
    if (func_statement()) {
        lexer();
        if (isToken("}") && nextTokenPosition < (tokens.size())) {
            nextTokenPosition--;
```

```
                return true;
            }
            else if (isToken("$$")) {
                if (nextTokenPosition == (tokens.size())) {
                    nextTokenPosition--;
                    return true;
                } else {
                    return false;
                }
            } else {
                nextTokenPosition--;
                return func_statementList();
            }
        } else {
            nextTokenPosition = pointer;
            return false;
        }

    }

    boolean func_statement() {
        int pointer = nextTokenPosition;
        boolean isCompound = func_compound();
        boolean isIf = func_if();
        boolean isReturn = func_return();
        boolean isWrite = func_write();
        boolean isRead = func_read();
        boolean isWhile = func_while();
        boolean isAssign = func_assign();
        boolean result = isCompound || isIf || isReturn || isWrite || isRead || isWhile ||
isAssign;
        if (result) {
            return true;
```

```
    }
    errorLoc = pointer;
    errorToken = tokens.get(pointer);
    errorMessage = "INCORRECT STATEMENT";
    return false;
}

boolean func_compound() {
    int pointer = nextTokenPosition;
    lexer();
    if (isToken("{")) {
        currentToken.addRule("<Statement> -> <Compound>");
        if (func_statementList()) {
            lexer();
            if (currentToken == null) {
                nextTokenPosition--;
                return false;
            }
            if (isToken("}")) {
                return true;
            }
        }
    }
    nextTokenPosition = pointer;
    return false;
}

boolean func_assign() {
    int pointer = nextTokenPosition;
    lexer();
    if (currentToken == null) {
        return false;
```

```
    }
    if (currentToken.tokenType == LexicalAnalyser.Type.IDENTIFIER) {
        currentToken.addRule("<Statement> -> <Assign>");
        currentToken.addRule("<Assign> -> <Identifier> := <Expression> ;");
        lexer();
        if (isToken(":=")) {

            if (func_expression()) {

                lexer();
                if (isToken(";")) {
                    return true;
                }
            }
        }
    }
    nextTokenPosition = pointer;

    return false;

}

boolean func_return() {
    int pointer = nextTokenPosition;
    lexer();
    if (currentToken == null) {
        nextTokenPosition--;
        return false;
    }
    if (currentToken.tokenType == LexicalAnalyser.Type.KEYWORD && isToken("return")) {
        currentToken.addRule("<Statement> -> <Return>");
        lexer();
```

```
        if (isToken(";")) {
            currentToken.addRule("<Return> -> return;");
            return true;
        } else {
            nextTokenPosition--;
            currentToken.addRule("<Return> -> return <Expression>;");
            if (func_expression()) {
                lexer();
                if (isToken(";")) {
                    return true;
                }
            }
        }
    }
    nextTokenPosition = pointer;
    return false;
}

boolean func_write() {
    int pointer = nextTokenPosition;
    lexer();
    if (currentToken == null) {
        nextTokenPosition--;
        return false;
    }
    if (currentToken.tokenType == LexicalAnalyser.Type.KEYWORD && isToken("print")) {
        currentToken.addRule("<Statement> ->  <Write>");
        currentToken.addRule("<Write> ->  print (<Expression>);");
        lexer();
        if (isToken("(")) {
            if (func_expression()) {
```

```
                lexer();
                if (isToken(")")) {
                    lexer();
                    if (isToken(";")) {
                        return true;
                    } else {
                        nextTokenPosition--;
                        return false;
                    }
                } else {
                    nextTokenPosition--;
                    return false;
                }
            }
        } else {
            nextTokenPosition--;
            return false;
        }

    } else {
        nextTokenPosition--;
        return false;
    }
    nextTokenPosition = pointer;
    return false;
}

boolean func_read() {
    int pointer = nextTokenPosition;
    lexer();
    if (isToken("read")) {
        currentToken.addRule("<Statement> -> <Read>");
```

```
        currentToken.addRule("<Read> -> read <IDs>");
        lexer();
        if (isToken("(")) {
            if (func_IDs()) {
                lexer();
                if (isToken(")")) {
                    lexer();
                    if (isToken(";")) {
                        return true;
                    }
                }
            }
        }
    } else {
        nextTokenPosition = pointer;
        return false;
    }
    return false;
}

boolean func_while() {
    int pointer = nextTokenPosition;
    lexer();
    if (isToken("while")) {
        currentToken.addRule("<Statement> -> <While>");
        currentToken.addRule("<While> -> while (<Condition>) <Statement>");
        lexer();
        if (isToken("(")) {
            if (func_condition()) {
                lexer();
                if (isToken(")")) {
                    return func_statement();
```

```
                } else {
                    nextTokenPosition--;
                    return false;
                }
            }
        } else {
            nextTokenPosition--;
            return false;
        }
    } else {
        nextTokenPosition--;
        return false;
    }
    nextTokenPosition = pointer;
    return false;
}

boolean func_condition() {
    currentToken.addRule("<Statement> -> <Condition>");
    currentToken.addRule("<Condition> -> <Expression> <Relop> <Expression> ");
    return func_expression() && func_relop() && func_expression();
}

boolean func_relop() {
    lexer();
    if (isToken("=") || isToken("/=") || isToken(">") || isToken("<") || isToken("=>") ||
isToken("<=")) {
        currentToken.addRule("<Relop> -> " + currentToken.content);
        return true;
    } else {
        nextTokenPosition--;
    }
    nextTokenPosition--;
```

```
    return false;
}

boolean func_if() {
    lexer();
    if (isToken("if")) {
        currentToken.addRule("<Statement> -> <If>");
        currentToken.addRule("<If> -> if (<Condition>) <Statement> <If Prime>");
        currentToken.addRule("<Condition> -> <Expression> <Relop> <Expression>");
        currentToken.addRule("<Relop> ->  = | /= | > | < | => | <=");
        lexer();
        if (isToken("(")) {
            if (func_condition()) {
                lexer();
                if (isToken(")")) {
                    return func_statement() && func_if_prime();
                } else {
                    nextTokenPosition--;
                    return false;
                }
            }
        } else {
            nextTokenPosition--;
        }
    } else {
        nextTokenPosition--;
    }
    return false;
}

boolean func_if_prime() {
    lexer();
```

```
    if (isToken("endif")) {
        currentToken.addRule("<If Prime> -> endif");
        return true;
    } else if (isToken("else")) {
        if (func_statement()) {
            lexer();
            if (isToken("endif")) {
                currentToken.addRule("<If Prime> -> else <Statement> endif");
                return true;
            } else {
                nextTokenPosition--;
                return false;
            }
        }
    } else {
        nextTokenPosition--;
        return false;
    }
    nextTokenPosition--;
    return false;
}

boolean func_expression() {
    if (func_term()) {
        return func_expression_prime();
    }
    return false;
}

boolean func_expression_prime() {
    lexer();
    if (isToken("+")) {
```

```
        currentToken.addRule("<Expression Prime> -> + <Term> <Expression Prime>");
        return func_term() && func_expression_prime();
    } else if (isToken("-")) {
        currentToken.addRule("<Expression Prime> -> - <Term> <Expression Prime>");
        return func_term() && func_expression_prime();
    } else {
        currentToken.addRule("<Expression Prime> -> <Empty>");
        nextTokenPosition--;
        return true;
    }
}

boolean func_term() {
    if (func_factor()) {
        return func_term_prime();
    } else {
        return false;
    }
}

 boolean func_term_prime() {
    lexer();
    if (isToken("*")) {
        currentToken.addRule("<Term Prime> -> * <Factor> <Term Prime>");
        if (func_factor()) {
            return func_term_prime();
        }
    } else if (isToken("/")) {
        currentToken.addRule("<Term Prime> -> / <Factor> <Term Prime>");
        if (func_factor()) {
            return func_term_prime();
        }
```

```
    } else {
        currentToken.addRule("<Term Prime> -> <Empty>");
        nextTokenPosition--;
        return true;
    }
    return false;
}

boolean func_factor() {
    int pointer = nextTokenPosition;
    lexer();
    currentToken.addRule("<Expression> -> <Term> <Expression Prime>");
    currentToken.addRule("<Term> -> <Factor> <Term Prime>");
    if (isToken("-")) {
        currentToken.addRule("<factor> -> - <factor>");
        return func_factor();
    } else if (isToken("(")) {
        currentToken.addRule("<factor> -> (<Expression>)");
        boolean isExpression = func_expression();
        if (isExpression) {
            lexer();
            if (isToken(")")) {
                return true;
            } else {
                return false;
            }
        }

    } else if (isToken("true")) {
        currentToken.addRule("<factor> -> true");
        return true;
    } else if (isToken("false")) {
```

```
        currentToken.addRule("<factor> -> false");
        return true;
    } else if (currentToken.tokenType == LexicalAnalyser.Type.INTEGER) {
        currentToken.addRule("<factor> -> <Integer>");
        return true;
    } else if (currentToken.tokenType == LexicalAnalyser.Type.REAL) {
        currentToken.addRule("<factor> -> <Real>");
        return true;
    } else if (currentToken.tokenType == LexicalAnalyser.Type.IDENTIFIER) {
        lexer();
        if (isToken("[")) {
            addRule("<factor> -> <Identifier>[<IDs>]", pointer);
            addRule("<IDs> -> <Identifier> | <Identifier>, <IDs>", pointer);
            boolean isIDs = func_IDs();
            if (isIDs) {
                lexer();
                if (isToken("]")) {
                    return true;
                } else {
                    nextTokenPosition--;
                    return false;
                }
            } else {
                return false;
            }
        } else {
            addRule("<factor> -> Identifier", pointer);
            nextTokenPosition--;
            return true;
        }
    }
    nextTokenPosition = pointer;
```

```
        return false;
    }
    private void addRule(String rule, int pointer) {
        this.tokens.get(pointer).addRule(rule);
    }
}
```

SAMPLE INPUT #1:

```
$$
function convert[miles: real]{
      return miles * 1.6;
}
$$
      real miles, kilometers;
      read (miles);
      kilometers := convert[miles];
      if(kilometers => 1600.00){
            print (kilometers);
      }
      endif
$$
```

SAMPLE OUTPUT #1:

Token: SEPARATOR Lexeme: $$
<Rat16F> -> $$ <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$

Token: KEYWORD Lexeme: function
<Opt Function Definitions> -> <Function Definitions>
<Function> -> function <Identifier> [<Opt Parameter List>] <Opt Declaration List> <Body>

Token: IDENTIFIER Lexeme: convert

Token: SEPARATOR Lexeme: [
<Opt Parameter List> -> <Parameter List> | <Empty>

Token: IDENTIFIER Lexeme: miles
<Opt Parameter List> -> <Parameter List>

Token: OPERATOR Lexeme: :
<IDs> -> <Identifier>
<Parameter> -> <IDs> : <Qualifier>

Token: KEYWORD Lexeme: real
<Qualifier> -> real

Token: SEPARATOR Lexeme:]

Token: SEPARATOR Lexeme: {
<Opt Declaration List> - > <Empty>
<Statement> -> <Compound>

Token: KEYWORD Lexeme: return
<Statement> -> <Return>

Token: IDENTIFIER Lexeme: miles
<Return> -> return <Expression>;
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<factor> -> Identifier

Token: OPERATOR Lexeme: *
<Term Prime> -> * <Factor> <Term Prime>

Token: REAL Lexeme: 1.6

<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<factor> -> <Real>

Token: SEPARATOR Lexeme: ;
<Term Prime> -> <Empty>
<Expression Prime> -> <Empty>

Token: SEPARATOR Lexeme: }

Token: SEPARATOR Lexeme: $$

Token: KEYWORD Lexeme: real
<Opt Declaration List> - > <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> real

Token: IDENTIFIER Lexeme: miles

Token: SEPARATOR Lexeme: ,
<IDs> -> <Identifier> , <IDs>

Token: IDENTIFIER Lexeme: kilometers

Token: SEPARATOR Lexeme: ;
<IDs> -> <Identifier>

Token: KEYWORD Lexeme: read
<Statement> -> <Read>
<Read> -> read <IDs>

Token: SEPARATOR Lexeme: (

Token: IDENTIFIER Lexeme: miles

Token: SEPARATOR Lexeme:)
<IDs> -> <Identifier>

Token: SEPARATOR Lexeme: ;

Token: IDENTIFIER Lexeme: kilometers
<Statement> -> <Assign>
<Assign> -> <Identifier> := <Expression> ;

Token: OPERATOR Lexeme: :=

Token: IDENTIFIER Lexeme: convert
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<factor> -> <Identifier>[<IDs>]
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: SEPARATOR Lexeme: [

Token: IDENTIFIER Lexeme: miles

Token: SEPARATOR Lexeme:]
<IDs> -> <Identifier>

Token: SEPARATOR Lexeme: ;

<Term Prime> -> <Empty>
<Expression Prime> -> <Empty>

Token: KEYWORD Lexeme: if
<Statement> -> <If>
<If> -> if (<Condition>) <Statement> <If Prime>
<Condition> -> <Expression> <Relop> <Expression>
<Relop> -> = | /= | > | < | => | <=

Token: SEPARATOR Lexeme: (
<Statement> -> <Condition>
<Condition> -> <Expression> <Relop> <Expression>

Token: IDENTIFIER Lexeme: kilometers
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<factor> -> Identifier

Token: OPERATOR Lexeme: =>
<Term Prime> -> <Empty>
<Expression Prime> -> <Empty>
<Relop> -> =>

Token: REAL Lexeme: 1600.00
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<factor> -> <Real>

Token: SEPARATOR Lexeme:)
<Term Prime> -> <Empty>
<Expression Prime> -> <Empty>

Token: SEPARATOR Lexeme: {
<Statement> -> <Compound>

Token: KEYWORD Lexeme: print
<Statement> -> <Write>
<Write> -> print (<Expression>);

Token: SEPARATOR Lexeme: (

Token: IDENTIFIER Lexeme: kilometers
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<factor> -> Identifier

Token: SEPARATOR Lexeme:)
<Term Prime> -> <Empty>
<Expression Prime> -> <Empty>

Token: SEPARATOR Lexeme: ;

Token: SEPARATOR Lexeme: }

Token: KEYWORD Lexeme: endif
<If Prime> -> endif

Token: SEPARATOR Lexeme: $$

THE SYNTAX IS CORRECT

SAMPLE CODE #2:

```
$$

function increase [number:integer]
{
      number := number + 2;
      return number;
}

$$
      integer start;
      start :   0;
      while (start < 20)
      {
          print (increase[start]);
      }
$$
```

OUTPUT #2:

Token: SEPARATOR Lexeme: $$
<Rat16F> -> $$ <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$

Token: KEYWORD Lexeme: function
<Opt Function Definitions> -> <Function Definitions>
<Function> -> function <Identifier> [<Opt Parameter List>] <Opt Declaration List> <Body>

Token: IDENTIFIER Lexeme: increase

Token: SEPARATOR Lexeme: [
<Opt Parameter List> -> <Parameter List> | <Empty>

Token: IDENTIFIER Lexeme: number
<Opt Parameter List> -> <Parameter List>

Token: OPERATOR Lexeme: :
<IDs> -> <Identifier>
<Parameter> -> <IDs> : <Qualifier>

Token: KEYWORD Lexeme: integer
<Qualifier> -> integer

Token: SEPARATOR Lexeme:]

Token: SEPARATOR Lexeme: {
<Opt Declaration List> - > <Empty>
<Statement> -> <Compound>

Token: IDENTIFIER Lexeme: number
<Statement> -> <Assign>
<Assign> -> <Identifier> := <Expression> ;

Token: OPERATOR Lexeme: :=

Token: IDENTIFIER Lexeme: number
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>

<factor> -> Identifier

Token: OPERATOR Lexeme: +
<Term Prime> -> <Empty>
<Expression Prime> -> + <Term> <Expression Prime>

Token: INTEGER Lexeme: 2
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<factor> -> <Integer>

Token: SEPARATOR Lexeme: ;
<Term Prime> -> <Empty>
<Expression Prime> -> <Empty>

Token: KEYWORD Lexeme: return
<Statement> -> <Return>

Token: IDENTIFIER Lexeme: number
<Return> -> return <Expression>;
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<factor> -> Identifier

Token: SEPARATOR Lexeme: ;
<Term Prime> -> <Empty>
<Expression Prime> -> <Empty>

Token: SEPARATOR Lexeme: }

Token: SEPARATOR Lexeme: $$

Token: KEYWORD Lexeme: integer
<Opt Declaration List> - > <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer

Token: IDENTIFIER Lexeme: start

Token: SEPARATOR Lexeme: ;
<IDs> -> <Identifier>

Token: IDENTIFIER Lexeme: start
<Statement> -> <Assign>
<Assign> -> <Identifier> := <Expression> ;

ERROR AT LINE 10: INCORRECT STATEMENT