COVER PAGE

CS323 Programming Assignments

Andres Imperial

Assignment 1: Syntax Analyzer/Parser

Due: November 6

Turned in: November 6

Executable FileName [Parsley]

Lab Room: Titan online server

OS: Linux Mint 18 ‘Sarah’ 64bit

GRADE:

COMMENTS:

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.

```cpp
// ----------------------------------------------------------------------------
// Andres Imperial
// CSCP 323 mw 11:30am
// Assignment 2: Parser
//
// file: parser.h
// ----------------------------------------------------------------------------

#ifndef PARSER_H
#define PARSER_H
#include <iostream>
#include <stdlib.h>
#include <cstring>
#include <fstream>
#include "parser_functs.h"
using namespace std;
const int token = 0;
const int lexeme = 1;
// Global variables
string** lexerArr;
int arrIndex = 1;
ofstream parserFile;

bool Parser(string fileName, string sourceName, int lineCount)
{

  ifstream lexerFile;

  lexerArr = new string*[lineCount + 1];
  bool compile = true;

  // Open file for reading
  lexerFile.open(fileName.c_str());
  parserFile.open((string("parser_") + sourceName).c_str());

  // If file opened properly
  if(!lexerFile.is_open()){
    cout << "Error file -- " << fileName << " -- could not be opened!\n";
    return 0;
  }
  if(!parserFile.is_open()){
    cout << "Error -- parser.txt could not be opened.\n";
    return 0;
  }

  for(int i = 0; !lexerFile.eof(); ++i){
    lexerArr[i] = new string[2];
    // Load array
    lexerFile >> lexerArr[i][token]; // Token
    lexerFile >> lexerArr[i][lexeme]; // Lexeme
  }

  // Start the Parsing at the root
  if(!Rat16F()){
    // Failed Parsing
    cout << "Error unable to parse file!\n";
    compile = false;
  }

  // Delete dynamic memory
  for(int i = 0; i < lineCount + 1; ++i){
    delete[] lexerArr[i];
  }
  delete[] lexerArr;

  lexerFile.close();
  parserFile.close();

  return compile;

} // End of Parser()


// ----- ErrorMsg =------------------------------------------------------------
// ----------------------------------------------------------------------------
void ErrorMsg(string msg)
{
  // Print out error message with line number, given token and lexeme, and
  // expected lexeme or token

  cout << "Error on line: " << arrIndex << " -- expected " << msg
    << " instead received lexeme: " << lexerArr[arrIndex][lexeme]
    << " token type: " << lexerArr[arrIndex][token] << endl;

  // Exit program
  exit(0);

} // End of ErrorMsg()


// ----- lexeme_is ------------------------------------------------------------
// ----------------------------------------------------------------------------
bool lexeme_is(string target)
{

  // Create compile flag
  bool compile = true;

  if(lexerArr[arrIndex][lexeme] == target){
    ++arrIndex;

    parserFile << "\nToken: " << lexerArr[arrIndex][token] << "\tLexeme: "
    << lexerArr[arrIndex][lexeme] << endl;
  }
  else{
    // Lexemes did not match
    compile = false;
  }

  return compile;

} // End of lexeme_is()


// ----- token_is ---------------------------------------------------------------
// ----------------------------------------------------------------------------
bool token_is(string target)
{

  // Create compile flag
  bool compile = true;

  if(lexerArr[arrIndex][token] == target){
    ++arrIndex;

    parserFile << "\nToken: " << lexerArr[arrIndex][token] << "\tLexeme: "
    << lexerArr[arrIndex][lexeme] << endl;
  }
  else{
    // Tokens did not match
    compile = false;
  }

  return compile;
```

```

} // End of token_is()


// ----- Rat16F ---------------------------------------------------------------
// ----------------------------------------------------------------------------
bool Rat16F(void)
{

  // First Production Rule
  parserFile << "Token: " << lexerArr[arrIndex][token] << "\tLexeme: "
    << lexerArr[arrIndex][lexeme] << endl;
  parserFile << "<Rat16F> -> $$ <Opt Function Definitions>\n"
        "$$ <Opt Declaration List> <Statement List> $$\n";

  // Create compile flag
  bool compile = false;
  // Program must start with $$ marker
  if(lexeme_is("$$")){
    if(Opt_Funct_Def()){
      if(lexeme_is("$$")){
        if(Opt_Declar_List()){
          if(Statement_List()){
            if(lexeme_is("$$")){
              // File was syntactically correct.
              compile = true;
            }
            else{
              ErrorMsg("$$");
            }
          }
          else{
            ErrorMsg("<Statement_List>");
          }
        }
      }
      else{
        ErrorMsg("$$");
      }
    }
  }
  else{
    ErrorMsg("$$");
  }

  return compile;

} // End of Rat16F()


// ----- Opt_Funct_Def --------------------------------------------------------
// ----------------------------------------------------------------------------
bool Opt_Funct_Def(void)
{

  // Production Rule
  parserFile << "<Opt Function Definitions> -> <Function Definitions> | <Empty>\n";

  // Create compile flag
  bool compile = true;

  if (Funct_Def()){
    return compile;
  }

  // It was empty, but acceptable
  return compile;
```

```cpp

} // End of Opt_Funct_Def()


// ----- Opt_Declar_List ---------------------------------------------------
// -------------------------------------------------------------------------
bool Opt_Declar_List(void)
{

  // Production Rule
  parserFile << "<Opt Declaration List> -> <Declaration List> | <Empty>\n";

  // Create compile flag
  bool compile = true;

  if (Declar_List()){
    return compile;
  }

  // It was empty, but acceptable
  return compile;

} // End of Opt_Declar_List()


// ----- Statement_List ----------------------------------------------------
// -------------------------------------------------------------------------
bool Statement_List(void)
{

  // Production Rule
  parserFile << "<Statement List> -> <Statement> | <Statement> <Statement List>\n";

  // Create compile flag
  bool compile = true;

  if(Statement()){
    while(Statement());
  }
  else{
    // Fail on Statement()
    compile = false;
  }

  return compile;

} // End of Statement_List()


// ----- Funct_Def ---------------------------------------------------------
// -------------------------------------------------------------------------
bool Funct_Def(void)
{

  // Production Rule
  parserFile << "<Function Definitions> -> <Function> | "
      "<Function> <Function Definitions>\n";

  // Create compile flag
  bool compile = true;

  if (Function()){
    if (Funct_Def()){
    }
  }
  else{
    // Fail on Function()
    compile = false;
  }

  return compile;

} // End of Funct_Def()


// ----- Function ------------------------------------------------------------
// ----------------------------------------------------------------------------
bool Function(void)
{

  // Production Rule
  parserFile << "<Function> -> function  <Identifier> [ <Opt Parameter List> ] "
      "<Opt Declaration List> <Body>\n";

  // Create compile flag
  bool compile = true;

  // Function must start with keyword function
  if (lexeme_is("function")){
    if (Identifier()){
      if (lexeme_is("[")){
        if (Opt_Param_List()){
          if (lexeme_is("]")){
            if (Opt_Declar_List()){
              if (Body()){
              }
              else{
                // Failed on Body()
                compile = false;
              }
            }
            else{
              // Failed on Opt_Declar_List()
              compile = false;
            }
          }
          else{
            // Failed on lexeme_is("]")
            ErrorMsg("]");
            compile = false;
          }
        }
        else{
          // Failed on Opt_Param_List()
          compile = false;
        }
      }
      else{
        // Failed on lexeme_is("[")
        ErrorMsg("[");
        compile = false;
      }
    }
    else{
      // Failed on Identifier()
      compile = false;
    }
  }
  else{
    // Failed on lexeme_is("function")
    compile = false;
  }

  return compile;
```

```

} // End of Function()


// ----- Identifier -----------------------------------------------------------
// ----------------------------------------------------------------------------
bool Identifier(void)
{

  // Create compile flag
  bool compile = true;

  if(!token_is("identifier")){
    // Failed on token_is identifer
    compile = false;
  }

  return compile;

} // End of Identifier()


// ----- Opt_Param_List -------------------------------------------------------
// ----------------------------------------------------------------------------
bool Opt_Param_List(void)
{

  // Production Rule
  parserFile << "<Opt Parameter List> ->  <Parameter List> | <Empty>\n";

  // Create compile flag
  bool compile = true;

  if(Param_List()){
    return compile;
  }

  // It was empty but acceptable.
  return compile;

} // End of Opt_Param_List()


// ----- Body -----------------------------------------------------------------
// ----------------------------------------------------------------------------
bool Body(void)
{

  // Production Rule
  parserFile << "<Body> -> { <Statement List> }\n";

  // Create compile flag
  bool compile = true;

  // Body must start with {
  if(lexeme_is("{")){
    if(Statement_List()){
      if(lexeme_is("}")){
        // used <Body>  ::=  {  < Statement List>  }
      }
      else{
        // failed on lexeme_is("}")
        ErrorMsg("}");
        compile = false;
      }
    }
    else{
      // failed on Statement_List()
      compile = false;
    }
  }
  else{
    // failed on lexeme_is("{")
    ErrorMsg("{");
    compile = false;
  }

  return compile;

} // End of Body()


// ----- Param_List -----------------------------------------------------------
// ----------------------------------------------------------------------------
bool Param_List(void)
{

  // Production Rule
  parserFile << "<Parameter List> -> <Parameter> | <Parameter> , "
      "<Parameter List>\n";

  // Create compile flag
  bool compile = true;

    if(Parameter()){
    while(lexeme_is(",")){
      Parameter();
    }
    }
    else{
        // Failed on Parameter()
        compile = false;
    }

    return compile;

} // End of Param_List()


// ----- Parameter ------------------------------------------------------------
// ----------------------------------------------------------------------------
bool Parameter(void)
{

  // Production Rule
  parserFile << "<Parameter> -> <IDs> : <Qualifier>\n";

  // Create compile flag
  bool compile = true;

    if(IDs()){
        if(lexeme_is(":")){
            if(Qualifier()){
                // <Parameter> ::=  <IDs > : <Qualifier>
            }
            else{
                // Failed on Qualifier()
                compile = false;
            }
        }
        else{
            // Failed on lexeme_is(":")
      ErrorMsg(":");
            compile = false;
        }
    }
    else{
        // Failed on IDs()
        compile = false;
    }

    return compile;

} // End of Parameter()

// ----- IDs ---------------------------------------------------------------
// -------------------------------------------------------------------------
bool IDs(void)
{

  // Production Rule
  parserFile << "<IDs> -> <Identifier> | <Identifier>, <IDs>\n";

  // Create compile flag
  bool compile = true;

    if(Identifier()){
    while(lexeme_is(",")){
      if(Identifier()){
      }
    }
    }
    else{
        // Failed on Identifier()
    ErrorMsg("<identifier>");
        compile = false;
    }

    return compile;

} // End of IDs()


// ----- Qualifier ---------------------------------------------------------
// -------------------------------------------------------------------------
bool Qualifier(void)
{

  // Production Rule
  parserFile << "<Qualifier> -> integer | boolean | real\n";

  // Create compile flag
  bool compile = true;

    if(lexeme_is("integer") | lexeme_is("boolean") | lexeme_is("real")){
        // <Qualifier> ::= integer   |  boolean  |  real
    }
    else{
        // Failed on lexeme_is() expected.....
        compile = false;
    }

    return compile;

} // End of Qualifier()


// ----- Declar_List -------------------------------------------------------
// -------------------------------------------------------------------------
bool Declar_List(void)
{

  // Production Rule
  parserFile << "<Declaration List> -> <Declaration> ; | <Declaration> ; "
      "<Declaration List>\n";

  // Create compile flag
  bool compile = true;

    if(Declaration()){
        if(lexeme_is(";")){
            Declar_List();
        }
        else{
            // Failed on lexeme_is(";")
      ErrorMsg(";");
            compile = false;
        }
    }
    else{
        // Failed on Declaration()
        compile = false;
    }

    return compile;

} // End of Declar_List()


// ----- Declaration ----------------------------------------------------------
// ---------------------------------------------------------------------------
bool Declaration(void)
{

  // Production Rule
  parserFile << "<Declaration> -> <Qualifier> <IDs>\n";
  // Create compile flag
  bool compile = true;

    if(Qualifier()){
        if(IDs()){
            // <Declaration> ::=  <Qualifier > <IDs>
        }
        else{
            // Failed on IDs()
            compile = false;
        }
    }
    else{
        // Failed on Qualifier()
        compile = false;
    }

    return compile;

} // End of Declaration()


// ----- Statement ------------------------------------------------------------
// ---------------------------------------------------------------------------
bool Statement(void)
{

  // Production Rule
  parserFile << "<Statement> -> <Compound> | <Assign> | <If> |  <Return> | "
        "<Write> | <Read> | <While>\n";

  // Create compile flag
  bool compile = true;

    if(Compound() | Assign() | If() | Return() | Write() | Read() | While()){
        // <Statement> ::=  <Compound> | <Assign> | <If> |  <Return> | <Write>
        // | <Read> | <While>
    }
    else{
        // Failed on <Compound> | <Assign> | <If> |  <Return>
        // | <Write> | <Read> | <While>
        compile = false;
    }

    return compile;

} // End of Statement()


// ----- Compound --------------------------------------------------------------
// -----------------------------------------------------------------------------
bool Compound(void)
{

  // Production Rule
  parserFile << "<Compound> -> { <Statement List> }\n";

  // Create compile flag
  bool compile = true;

    // Must start with "{"
    if(lexeme_is("{")){
        if(Statement_List()){
            if(lexeme_is("}")){
                // <Compound> ::= {  <Statement List>  }
            }
            else{
                // Failed on lexeme_is("}")
        ErrorMsg("}");
                compile = false;
            }
        }
        else{
            // Failed on Statement_List()
            compile = false;
        }
    }
    else{
        // Failed on lexeme_is("{")
        compile = false;
    }

    return compile;

} // End of Compound()


// ----- Assign ----------------------------------------------------------------
// -----------------------------------------------------------------------------
bool Assign(void)
{

  // Production Rule
  parserFile << "<Assign> -> <Identifier> := <Expression>;\n";

  // Create compile flag
  bool compile = true;

    if(Identifier()){
        if(lexeme_is(":=")){
            if(Expression()){
                if(lexeme_is(";")){
                    // <Assign> ::=   <Identifier> := <Expression> ;
                }
                else{
                    // Failed on lexeme_is(";")
          ErrorMsg(";");
                    compile = false;
                }
            }
            else{
                // Failed on Expression()
                compile = false;
            }
        }
        else{
            // Failed on lexeme_is(":=")
      ErrorMsg(":=");
            compile = false;
        }
    }
    else{
       // Failed on Identifier()
       compile = false;
    }

    return compile;

} // End of Assign()


// ----- If ---------------------------------------------------------------
// ------------------------------------------------------------------------
bool If(void)
{

  // Production Rule
  parserFile << "<If> -> if (<Condition>) <Statement> endif |\n"
            "if (<Condition>) <Statement> else <Statement> endif\n";

  // Create compile flag
  bool compile = true;

    // Must start with "if" keyword
    if(lexeme_is("if")){
        if(lexeme_is("(")){
            if(Condition()){
                if(lexeme_is(")")){
                    if(Statement()){
                        if(lexeme_is("endif")){
                            // <If> ::=     if  ( <Condition>  ) <Statement>
                            // endif    |
                        }
                        else if(lexeme_is("else")){
                            if(Statement()){
                                if(lexeme_is("endif")){
                                    // <If> ::= if ( <Condition>  ) <Statement>
                                    // else <Statement> endif
                                }
                                else{
                                    // Failed on lexeme_is("endif")
                  ErrorMsg("endif");
                                    compile = false;
                                }
                            }
                            else{
                                // Failed on Statement()
                                compile = false;
                            }
                        }
                        else{
                            // Failed on lexeme_is() expected endif or else
              ErrorMsg("endif | else");
                            compile = false;
                        }
                    }
                    else{
                        // Failed on Statement()
                        compile = false;
                    }
                }
                else{
                    // Failed on lexeme_is(")")
          ErrorMsg(")");
                    compile = false;
                }
            }
            else{
                // Failed on Condition()
                compile = false;
            }
        }
        else{
            // Failed on lexeme_is("(")
      ErrorMsg("(");
            compile = false;
        }
    }
    else{
        // Failed on lexeme_is("if")
        compile = false;
    }

    return compile;

} // End of If()


// ----- Return ---------------------------------------------------------------
// ---------------------------------------------------------------------------
bool Return(void)
{

  // Production Rule
  parserFile << "<Return> -> return ; | return <Expression> ;\n";

  // Create compile flag
  bool compile = true;

    if(lexeme_is("return")){
        if(lexeme_is(";")){
            // <Return> ::=  return ;
        }
        else if(Expression()){
            if(lexeme_is(";")){
                // <Return> ::= return <Expression> ;
            }
            else{
                // Failed on lexeme_is(";")
        ErrorMsg(";");
                compile = false;
            }
        }
        else{
            // Failed, expected ";" or Expression
      ErrorMsg("; | <Expression>");
            compile = false;
        }
    }
    else{
        // Failed on lexeme_is("return")
        compile = false;
    }

    return compile;

} // End of Return()


// ----- Write ----------------------------------------------------------------
// ----------------------------------------------------------------------------
bool Write(void)
{

  // Production Rule
  parserFile << "<Write> -> print (<Expression>);\n";

  // Create compile flag
  bool compile = true;

    // Must start with keyword "print"
    if(lexeme_is("print")){
        if(lexeme_is("(")){
            if(Expression()){
                if(lexeme_is(")")){
                    if(lexeme_is(";")){
                        // <Write> ::=   print ( <Expression>);
                    }
                    else{
                        // Failed on lexeme_is(";")
            ErrorMsg(";");
                        compile = false;
                    }
                }
                else{
                    // Failed on lexeme_is(")")
          ErrorMsg(")");
                    compile = false;
                }
            }
            else{
                // Failed on Expression()
                compile = false;
            }
        }
        else{
            // Failed on lexeme_is("(")
      ErrorMsg("(");
            compile = false;
        }
    }
    else{
        // Failed on lexeme_is("print")
        compile = false;
    }

    return compile;

} // End of Write()

```

```

// ----- Read ----------------------------------------------------------------
// ---------------------------------------------------------------------------
bool Read(void)
{

  // Production Rule
  parserFile << "<Read> -> read (<IDs>);\n";

  // Create compile flag
  bool compile = true;

    // Must start with keyword "read"
    if(lexeme_is("read")){
        if(lexeme_is("(")){
            if(IDs()){
                if(lexeme_is(")")){
                    if(lexeme_is(";")){
                        // <Read> ::=    read ( <IDs> );
                    }
                    else{
                        // Failed on lexeme_is(";")
            ErrorMsg(";");
                        compile = false;
                    }
                }
                else{
                    // Failed on lexeme_is(")")
          ErrorMsg(")");
                    compile = false;
                }
            }
            else{
                // Failed on IDs()
                compile = false;
            }
        }
        else{
            // Failed on lexeme_is("(")
      ErrorMsg("(");
            compile = false;
        }
    }
    else{
        // Failed on lexeme_is("read")
        compile = false;
    }

    return compile;

} // End of Read()


// ----- While ---------------------------------------------------------------
// ---------------------------------------------------------------------------
bool While(void)
{

  // Production Rule
  parserFile << "<While> -> while (<Condition>) <Statement>\n";

  // Create compile flag
  bool compile = true;

    // Must start with keyword "while"
    if(lexeme_is("while")){
        if(lexeme_is("(")){
            if(Condition()){
                if(lexeme_is(")")){
                    if(Statement()){
                        // <While> ::= while ( <Condition>  )  <Statement>
                    }
                    else{
                        // Failed on Statement()
                        compile = false;
                    }
                }
                else{
                    // Failed on lexeme_is(")")
          ErrorMsg(")");
                    compile = false;
                }
            }
            else{
                // Failed on Condition()
                compile = false;
            }
        }
        else{
            // Failed on lexeme_is("(")
      ErrorMsg("(");
            compile = false;
        }
    }
    else{
        // Failed on lexeme_is("while")
        compile = false;
    }

    return compile;

}// End of While()


// ----- Expression -----------------------------------------------------------
// ----------------------------------------------------------------------------
bool Expression(void)
{

  // Production Rule
  parserFile << "<Expression> -> <Term> <Expression Prime>\n";

  // Create compile flag
  bool compile = true;

    if(Term()){
        if(Expression_Prime()){
            // <Expression> ::= <Term> <Expression Prime>
        }
        else{
            // Failed on Expression_Prime()
            compile = false;
        }
    }
    else{
        // Failed on Term()
        compile = false;
    }

    return compile;

} // End of Expression()
```

```
// ----- Condition ----------------------------------------------------------
// --------------------------------------------------------------------------
bool Condition(void)
{

  // Production Rule
  parserFile << "<Condition> -> <Expression> <Relop> <Expression>\n";

  // Create compile flag
  bool compile = true;

    if(Expression()){
        if(Relop()){
            if(Expression()){
                // <Condition> ::= <Expression> <Relop> <Expression>
            }
            else{
                // Failed on Expression()
                compile = false;
            }
        }
        else{
            // Failed on Relop()
            compile = false;
        }
    }
    else{
        // Failed on Expression()
        compile = false;
    }

    return compile;

} // End of Condition()


// ----- Relop --------------------------------------------------------------
// --------------------------------------------------------------------------
bool Relop(void)
{

  // Production Rule
  parserFile << "<Relop> -> = | /= | > | < | => | <=\n";

  // Create compile flag
  bool compile = true;

    if(lexeme_is("=") | lexeme_is("/=") | lexeme_is(">") | lexeme_is("<") |
        lexeme_is("=>") | lexeme_is("<=")){
        //<Relop> ::=   = |  /=  |   >   | <   |  =>   | <=
    }
    else{
        // Failed on <Relop> ::=   = |  /=  |   >   | <   |  =>   | <=
    ErrorMsg("= | /= | > | < | => | <=");
        compile = false;
    }

    return compile;

} // End of Relop()


// ----- Expression_Prime ---------------------------------------------------
// --------------------------------------------------------------------------
bool Expression_Prime(void)
{

  // Production Rule
  parserFile << "<Expression Prime> -> +<Term> <Expression Prime> | "
      "-<Term> <Expression Prime> | epsilon\n";

  // Create compile flag
  bool compile = true;

    if(lexeme_is("+") | lexeme_is("-")){
        if(Term()){
            if(Expression_Prime()){
                //<Expression Prime> ::= +<Term> <Expression Prime> | -<Term>
                //<Expression Prime> | epsilon
            }
            else{
                // Failed on Expression_Prime()
                compile = false;
            }
        }
        else{
            // Failed on Term()
            compile = false;
        }
    }
    else{
        // Was empty and moved to epsilon
        // <Expression Prime> ::= epsilon
    }

    return compile;

} // End of Expression_Prime()


// ----- Term -----------------------------------------------------------------
// ----------------------------------------------------------------------------
bool Term(void)
{

  // Production Rule
  parserFile << "<Term> -> <Factor> <Term Prime>\n";

  // Create compile flag
  bool compile = true;

    if(Factor()){
        if(Term_Prime()){
            // <Term>  ::= <Factor> <Term Prime>
        }
        else{
            // Failed on Term_Prime()
            compile = false;
        }
    }
    else{
        // Failed on Factor()
        compile = false;
    }

    return compile;

} // End of Term()


// ----- Term_Prime -----------------------------------------------------------
// ----------------------------------------------------------------------------
bool Term_Prime(void)
{

  // Production Rule
  parserFile << "<Term Prime> -> * <Factor> <Term Prime> | "
        "/ Factor <Term Prime> | epsilon\n";

  // Create compile flag
  bool compile = true;

    if(lexeme_is("*") | lexeme_is("/")){
        if(Factor()){
            if(Term_Prime()){
                // <Term Prime> ::=  * <Factor> <Term Prime> | / Factor <Term
                // Prime> | epsilon
            }
            else{
                // Failed on Term_Prime()
                compile = false;
            }
        }
        else{
            // Failed on Factor()
            compile = false;
        }
    }
    else{
        // It moved to epsilon
        // <Term Prime> ::= epsilon
    }

    return compile;

} // End Term_Prime()


// ----- Factor ---------------------------------------------------------------
// ---------------------------------------------------------------------------
bool Factor(void)
{

  // Production Rule
  parserFile << "<Factor> -> - <Primary> | <Primary>\n";

  // Create compile flag
  bool compile = true;

    if(lexeme_is("-")){
        if(Primary()){
            // <Factor> ::= - <Primary>
        }
        else{
            // Failed on Primary()
            compile = false;
        }
    }
    else if(Primary()){
        // <Factor> ::= <Primary>
    }
    else{
        // Failed expected "-" or Primary
    ErrorMsg("- | <Primary>");
        compile = false;
    }

    return compile;

} // Factor()

```

```

// ----- Primary -------------------------------------------------------------
// ---------------------------------------------------------------------------
bool Primary(void)
{

  // Production Rule
  parserFile << "<Primary> -> <Identifier> | <Integer> | <Identifier> "
        "[<IDs>] | (<Expression>) | <Real> | true | false\n";

  // Create compile flag
  bool compile = true;

    if(Identifier()){
        if(lexeme_is("[")){
            if(IDs()){
                if(lexeme_is("]")){
                    // <Primary> ::= <Identifier> [<IDs>]
                }
                else{
                    // Failed on lexeme_is("]")
          ErrorMsg("]");
                    compile = false;
                }
            }
            else{
                // Failed on IDs()
                compile = false;
            }
        }
        else{
            // <Primary> ::= <Identifier>
        }
    }
    else if(token_is("integer")){
        // <Primary> ::= <Integer>
    }
    else if(lexeme_is("(")){
        if(Expression()){
            if(lexeme_is(")")){
                // <Primary> ::= ( <Expression> )
            }
            else{
                // Failed on lexeme_is(")")
        ErrorMsg(")");
                compile = false;
            }
        }
        else{
            // Failed on Expression()
            compile = false;
        }
    }
    else if(token_is("real")){
        // <Primary> ::= <Real>
    }
    else if(lexeme_is("true") | lexeme_is("false")){
        // <Primary> ::= true | false
    }
    else{
        // Failed expected . . . .
    ErrorMsg("<Identifier> | <Integer> | <Identifier> [<IDs>] | "
        "<Expression> | <Real> | true | false");
        compile = false;
    }

    return compile;
```

```

} // End of Primary()

#endif // End of parser.h
```

```
// ----------------------------------------------------------------------------
// Andres Imperial
// CSCP 323 mw 11:30am
// Assignment 2: Parser
//
// file: parser_functs.h
// ----------------------------------------------------------------------------

#ifndef PARSER_FUNCTS_H
#define PARSER_FUNCTS_H
#include <cstring>
using namespace std;

// Function Declarations

bool Parser(string fileName, int lineCount);
bool lexeme_is(string target);
bool token_is(string target);
bool Rat16F(void);
bool Opt_Funct_Def(void);
bool Opt_Declar_List(void);
bool Statement_List(void);
bool Funct_Def(void);
bool Function(void);
bool Identifier(void);
bool Opt_Param_List(void);
bool Body(void);
bool Param_List(void);
bool Parameter(void);
bool IDs(void);
bool Qualifier(void);
bool Declar_List(void);
bool Declaration(void);
bool Statement(void);
bool Compound(void);
bool Assign(void);
bool If(void);
bool Return(void);
bool Write(void);
bool Read(void);
bool While(void);
bool Expression(void);
bool Condition(void);
bool Relop(void);
bool Expression_Prime(void);
bool Term(void);
bool Term_Prime(void);
bool Factor(void);
bool Primary(void);

#endif // End of parser_functs.h
```

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

Token: separator  Lexeme: $$
<Opt Function Definitions> -> <Function Definitions> | <Empty>
<Function Definitions> -> <Function> | <Function> <Function Definitions>
<Function> -> function  <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>

Token: separator  Lexeme: {
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }

Token: identifier Lexeme: a
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<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: operator Lexeme: +
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: identifier Lexeme: c
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

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

Token: separator  Lexeme: }
<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>
<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>

Token: separator  Lexeme: $$
<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>
<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>

Token:  Lexeme:
```

```
$$
$$
{
a := b + c;
}
$$
```

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

Token: keyword  Lexeme: function
<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: Subtract

Token: separator  Lexeme: [

Token: identifier Lexeme: imVal
<Opt Parameter List> ->  <Parameter List> | <Empty>
<Parameter List> -> <Parameter> | <Parameter> , <Parameter List>
<Parameter> -> <IDs> : <Qualifier>
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: :

Token: keyword  Lexeme: integer
<Qualifier> -> integer | boolean | real

Token: separator  Lexeme: ]

Token: keyword  Lexeme: real
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: retVal92
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ;

Token: separator  Lexeme: {
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<Body> -> { <Statement List> }

Token: identifier Lexeme: imVal
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: imVal
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: operator Lexeme: -
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: (
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: integer  Lexeme: 2
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: operator Lexeme: *
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon

Token: identifier Lexeme: imVal
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

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

Token: identifier Lexeme: retVal92
<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>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: imVal
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

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

Token: keyword  Lexeme: return
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;

Token: identifier Lexeme: retVal92
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

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

Token: separator  Lexeme: }
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
```

```
<While> -> while (<Condition>) <Statement>
<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>

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

Token: keyword  Lexeme: integer
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: low_av
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ,

Token: identifier Lexeme: high_av

Token: separator  Lexeme: ;

Token: keyword  Lexeme: read
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<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>);

Token: separator  Lexeme: (

Token: identifier Lexeme: low_av
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ,

Token: identifier Lexeme: high_av

Token: separator  Lexeme: )

Token: separator  Lexeme: ;

Token: keyword  Lexeme: while
<While> -> while (<Condition>) <Statement>

Token: separator  Lexeme: (

Token: identifier Lexeme: low_av
<Condition> -> <Expression> <Relop> <Expression>
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
```

```
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

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

Token: operator Lexeme: <

Token: identifier Lexeme: high_av
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: {
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }

Token: keyword  Lexeme: print
<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>);

Token: separator  Lexeme: (

Token: identifier Lexeme: low_av
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: ;

Token: keyword  Lexeme: print
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>
<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>);

Token: separator  Lexeme: (

Token: identifier Lexeme: Subtract
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: separator  Lexeme: [

Token: identifier Lexeme: low_av
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ]

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: ;

Token: separator  Lexeme: }
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>
<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>

Token: separator  Lexeme: $$
<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>
<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>

Token:  Lexeme:
```

```
$$
function Subtract[imVal:integer]
  real retVal92;
{
  imVal := imVal - (2 * imVal);
  retVal92:=imVal;
  return retVal92;
}
$$
     integer   low_av,high_av;
     read(low_av,high_av);
     while (low_av =< high_av)
         {
    print(low_av);
              print(Subtract[low_av]);
         }
$$
```

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

Token: keyword  Lexeme: function
<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: Hello

Token: separator  Lexeme: [

Token: identifier Lexeme: me
<Opt Parameter List> ->  <Parameter List> | <Empty>
<Parameter List> -> <Parameter> | <Parameter> , <Parameter List>
<Parameter> -> <IDs> : <Qualifier>
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: :

Token: keyword  Lexeme: real
<Qualifier> -> integer | boolean | real

Token: separator  Lexeme: ,

Token: identifier Lexeme: alice
<Parameter> -> <IDs> : <Qualifier>
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: :

Token: keyword  Lexeme: integer
<Qualifier> -> integer | boolean | real

Token: separator  Lexeme: ,

Token: identifier Lexeme: bob
<Parameter> -> <IDs> : <Qualifier>
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: :

Token: keyword  Lexeme: boolean
<Qualifier> -> integer | boolean | real

Token: separator  Lexeme: ]

Token: keyword  Lexeme: boolean
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: modest
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ;

Token: keyword  Lexeme: boolean
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: mouse
<IDs> -> <Identifier> | <Identifier>, <IDs>

```

```
Token: separator  Lexeme: ;

Token: keyword  Lexeme: real
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: pink
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ;

Token: keyword  Lexeme: integer
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: floyd
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ;

Token: separator  Lexeme: {
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<Body> -> { <Statement List> }

Token: identifier Lexeme: alice
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: me
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

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

Token: keyword  Lexeme: if
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif

Token: separator  Lexeme: (

Token: identifier Lexeme: pink
<Condition> -> <Expression> <Relop> <Expression>
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: operator Lexeme: +
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: identifier Lexeme: floyd
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

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

Token: identifier Lexeme: modest
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: operator Lexeme: /
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon

Token: identifier Lexeme: mouse
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: identifier Lexeme: me
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: alice
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

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

Token: keyword  Lexeme: else
<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>

Token: identifier Lexeme: me
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: modest
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false
```

```

Token: operator Lexeme: +
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: identifier Lexeme: mouse
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

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

Token: keyword  Lexeme: endif
<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>

Token: keyword  Lexeme: return
<Return> -> return ; | return <Expression> ;

Token: identifier Lexeme: me
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: separator  Lexeme: [

Token: identifier Lexeme: modest
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ,

Token: identifier Lexeme: alice

Token: separator  Lexeme: ]

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

Token: separator  Lexeme: }
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>
<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>

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

Token: keyword  Lexeme: while
```

```
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<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>

Token: separator  Lexeme: (

Token: integer  Lexeme: 1
<Condition> -> <Expression> <Relop> <Expression>
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

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

Token: integer  Lexeme: 0
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: {
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }

Token: keyword  Lexeme: read
<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>);

Token: separator  Lexeme: (

Token: identifier Lexeme: Hello
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: )

Token: separator  Lexeme: ;

Token: separator  Lexeme: }
<While> -> while (<Condition>) <Statement>
```

```
<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>

Token: separator  Lexeme: $$
<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>
<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>

Token:  Lexeme:
```

```
$$
function Hello[me:real, alice:integer, bob:boolean]

boolean modest; boolean mouse;
real pink;
integer floyd;
{
  alice := me;
  if(pink + floyd > modest/mouse)
    me := alice;
  else
    me := modest + mouse;
  endif

  return me[modest, alice];
}
$$
  while(1 > 0)
  {
    read(Hello);
  }
$$
```