```
/*
     Jonathan Ishii
     Matthew Mikulka

     CPSC 323 - Section 1

     Assignment 2
 */


//Rewrite using try and catch blocks. if done correctly this should solve all
 the problems.
// not sure if this is the most elegand way of writing it.


#include <iostream>
#include "LexicalAnalyzer.h"
#include <string>
#include <fstream>
#include <iomanip>
#include <tuple>
#include "SyntaxAnalyzer.h"

using namespace std;

bool printSwitch = true;

//R1. <Rat18F>  ::=   <Opt Function Definitions>   $$  <Opt Declaration List>
 <Statement List>  $$
void Rat18F(ifstream &infile, ostream &outfile)
{
    outfile << endl;
    tuple<string, string> token = lexer(infile, outfile);
    if (printSwitch)
    {
        outfile << "\tR1. <Rat18F>  ::=   <Opt Function Definitions>   $$  <Opt
         Declaration List>  <Statement List>  $$" << endl;
    }

    OptFunctionDefinitions(infile, outfile, token);


    if(get<1>(token) != "$$")
    {
        errorReporting(outfile, "$$", get<1>(token));
    }
    token = lexer(infile, outfile);

    OptDeclarationList(infile, outfile, token);

    StatementList(infile, outfile, token);
```

```
    if(get<1>(token) != "$$")
    {
        errorReporting(outfile, "$$", get<1>(token));
    }

    char c;
    infile.get(c);
    while (isspace(c) && infile)
    {
        infile.get(c);
    }

    if(infile)
    {
        token = lexer(infile, outfile);
        errorReporting(outfile, "End of File", get<1>(token));
    }

    outfile << "\nSyntax Analyzer is completes.\n";
    cout << "\nSyntax Analyzer is complete.\n";

}

//R2. <Opt Function Definitions> ::= <Function Definitions>  |  <Empty>
void OptFunctionDefinitions(ifstream &infile, ostream &outfile, tuple<string,
 string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR2. <Opt Function Definitions> ::= <Function Definitions>
         |  <Empty>" << endl;
    }
    FunctionDefinitions(infile, outfile, token);
}

//R3. <Function Definitions>  ::= <Function> <Function Definitions End>
void FunctionDefinitions(ifstream &infile, ostream &outfile, tuple<string,
 string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR3. <Function Definitions>  ::= <Function> <Function
         Definitions End>" << endl;
    }
    if(Function(infile, outfile, token))
    {
        FunctionDefinitionsEnd(infile, outfile, token);
    }
}
```

```
//R4.<Function Definitions End> ::= <Function Definitions>  | ε
void FunctionDefinitionsEnd(ifstream &infile, ostream &outfile, tuple<string,
 string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR4.<Function Definitions End> ::= <Function Definitions>
         | ε" << endl;
    }
    FunctionDefinitions(infile, outfile, token);
}

//R5. <Function> ::= function  <Identifier>   ( <Opt Parameter List> )  <Opt
 Declaration List>  <Body>
bool Function(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR5. <Function> ::= function  <Identifier>   ( <Opt
         Parameter List> )  <Opt Declaration List>  <Body>" << endl;
    }

    if(get<1>(token) == "function")
    {
        token = lexer(infile, outfile);
        if(!Identifier(outfile, token))
        {
            errorReporting(outfile, "Identifier", get<0>(token));
        }

        token = lexer(infile, outfile);
        if(get<1>(token) != "(")
        {
            errorReporting(outfile, "(", get<1>(token));
        }

        token = lexer(infile, outfile);
        OptParameterList(infile, outfile, token);

        if(get<1>(token) != ")")
        {
            errorReporting(outfile, ")", get<1>(token));
        }

        token = lexer(infile, outfile);
        OptDeclarationList(infile, outfile, token);

        Body(infile, outfile, token);

        return true;
    }
```

```
    return false;
}

//identifier
bool Identifier(ostream &outfile, const tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR Checker. <Identifier>" << endl;
    }

    if(get<0>(token) != "Identifier")
    {
        return false;
    }
    outfile << "Identifier found." << endl;
    return true;
}

//R6. <Opt Parameter List> ::=  <Parameter List>    |     <Empty>
bool OptParameterList(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR6. <Opt Parameter List> ::=  <Parameter List>    |
         <Empty>" << endl;
    }
    if(ParameterList(infile, outfile, token))
    {
        return true;
    }
    return false;
}

//R7. <Parameter List>  ::=  <Parameter>  <Parameter List End>
bool ParameterList(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR7. <Parameter List>  ::=  <Parameter>  <Parameter List
         End>" << endl;
    }
    if(Parameter(infile, outfile, token))
    {
        ParameterListEnd(infile, outfile, token);
        return true;
    }
    return false;
}
```

```
//R9. <Parameter> ::=  <IDs > : <Qualifier>
bool Parameter(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR9. <Parameter> ::=  <IDs > : <Qualifier>" << endl;
    }
    if(IDs(infile, outfile, token))
    {
        if(get<1>(token) != ":")
        {
            errorReporting(outfile, ":", get<1>(token));
        }

        token = lexer(infile, outfile);
        if(!Qualifier(infile, outfile, token))
        {
            errorReporting(outfile, "int | true | false | real",
             get<1>(token));
        }
        token = lexer(infile, outfile);
        return true;
    }
    return false;
}

//R8. <Parameter List End> ::= , <Parameter List> | ε
void ParameterListEnd(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR8. <Parameter List End> ::= , <Parameter List> | ε" <<
         endl;
    }
    if(get<1>(token) == ",")
    {
        token = lexer(infile, outfile);
        if(!ParameterList(infile, outfile, token))
        {
            errorReporting(outfile, "<ParameterList>", get<1>(token));
        }
    }
}

//R10. <Qualifier> ::= int     |    boolean    |  real
bool Qualifier(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
```

```
    //token = lexer(infile, outfile);
    if (printSwitch)
    {
        outfile << "\tR10. <Qualifier> ::= int     |    boolean    |  real" <<
         endl;
    }
    if(get<1>(token) != "int" && get<1>(token) != "boolean" && get<1>(token) !=
     "real")
    {
        return false;
    }
    return true;
}

//R11. <Body>  ::= {  < Statement List>  }
void Body(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{

    if (printSwitch)
    {
        outfile << "\tR11. <Body>  ::= {  < Statement List>  }" << endl;
    }

    if(get<1>(token) != "{")
    {
        errorReporting(outfile, "{", get<1>(token));
    }

    token = lexer(infile, outfile);
    if(!StatementList(infile, outfile, token))
    {
        errorReporting(outfile, "{ | identifier | if | return | put | get |
         while", get<1>(token));
    }

    if(get<1>(token) != "}")
    {
        errorReporting(outfile, "}", get<1>(token));
    }
    token = lexer(infile, outfile);
}

//R12. <Opt Declaration List> ::= <Declaration List>   |    <Empty>
void OptDeclarationList(ifstream &infile, ostream &outfile, tuple<string,
 string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR12. <Opt Declaration List> ::= <Declaration List>   |
         <Empty>" << endl;
    }
```

```
    DeclarationList(infile, outfile, token);
}

//R13. <Declaration List>  ::= <Declaration> ;   <Declaration List End>
void DeclarationList(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR13. <Declaration List>  ::= <Declaration> ;
         <Declaration List End>" << endl;
    }

    if(Declaration(infile, outfile, token))
    {
        if(get<1>(token) != ";")
        {
            errorReporting(outfile, ";", get<1>(token));
        }
        token = lexer(infile, outfile);
        DeclarationListEnd(infile, outfile, token);
    }
}

//R14. <Declaration List End> ::= <Declaration List> | ε
void DeclarationListEnd(ifstream &infile, ostream &outfile, tuple<string,
 string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR14. <Declaration List End> ::= <Declaration List> | ε"
         << endl;
    }
    DeclarationList(infile, outfile, token);
}

//R15. <Declaration> ::=   <Qualifier > <IDs>
bool Declaration(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR15. <Declaration> ::=   <Qualifier > <IDs>" << endl;
    }

    if(Qualifier(infile, outfile, token))
    {
        token = lexer(infile, outfile);
        if(!IDs(infile, outfile, token))
        {
```

```
            errorReporting(outfile, "<IDs>/Identifier", get<1>(token));
        }
        return true;
    }
    return false;
}

//R16. <IDs> ::=     <Identifier>  <IDs End>
bool IDs(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR16. <IDs> ::=     <Identifier>  <IDs End>" << endl;
    }
    if(Identifier(outfile, token))
    {
        token = lexer(infile, outfile);
        IDsEnd(infile, outfile, token);
        return true;
    }
    return false;
}

//R17. <IDs End> ::= , <IDs> | ε
void IDsEnd(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR17. <IDs End> ::= , <IDs> | ε" << endl;
    }
    if(get<1>(token) == ",")
    {
        token = lexer(infile, outfile);
        if(!IDs(infile, outfile, token))
        {
            errorReporting(outfile, "Identifier", get<0>(token));
        }
    }
}

//R18. <Statement List> ::=   <Statement>  <Statement List End>
bool StatementList(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR18. <Statement List> ::=   <Statement>  <Statement List
         End>" << endl;
    }
    if(Statement(infile, outfile, token))
    {
```

```
        StatementListEnd(infile, outfile, token);
        return true;
    }
    return false;
}

//R19. <Statement List End> ::= <Statement List> | ε
void StatementListEnd(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR19. <Statement List End> ::= <Statement List> | ε" <<
         endl;
    }

    StatementList(infile, outfile, token);
}

//R20. <Statement> ::=   <Compound>  |  <Assign> |   <If>  |  <Return>   |
 <Print>   |   <Scan>   |  <While>
bool Statement(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR20. <Statement> ::=   <Compound>  |  <Assign> |   <If>
         |  <Return>   | <Print>   |   <Scan>   |  <While>" << endl;
    }

    if (If(infile, outfile, token))
    {
        return true;
    }
    else if (Return(infile, outfile, token))
    {
        return true;
    }
    else if (Print(infile, outfile, token))
    {
        return true;
    }
    else if (Scan(infile, outfile, token))
    {
        return true;
    }
    else if (While(infile, outfile, token))
    {
        return true;
    }
    else if (Assign(infile, outfile, token))
```

```
    {
        return true;
    }
    else if(Compound(infile, outfile, token))
    {
        return true;
    }
    return false;
}

//R21. <Compound> ::=   { <Statement List>  }
bool Compound(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR21. <Compound> ::=   { <Statement List>  }" << endl;
    }

    if(get<1>(token) == "{")
    {
        token = lexer(infile, outfile);

        StatementList(infile, outfile, token);

        if(get<1>(token) != "}")
        {
            errorReporting(outfile, "}", get<1>(token));
        }

        token = lexer(infile, outfile);
        return true;
    }
    return false;
}

//R22. <Assign> ::=     <Identifier> = <Expression> ;
bool Assign(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR22. <Assign> ::=     <Identifier> = <Expression> ;" <<
         endl;
    }

    if(Identifier(outfile, token))
    {
        token = lexer(infile, outfile);
        if(get<1>(token) != "=")
        {
            errorReporting(outfile, "=", get<1>(token));
        }
```

```
        token = lexer(infile, outfile);
        Expression(infile, outfile, token);

        if(get<1>(token) != ";")
        {
            errorReporting(outfile, ";", get<1>(token));
        }
        token = lexer(infile, outfile);
        return true;
    }
    return false;
}

//R23. <If> ::=     if  ( <Condition>  ) <Statement>   <if End>
bool If(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR23. <If> ::=     if  ( <Condition>  ) <Statement>   <if
         End>" << endl;
    }

    if(get<1>(token) == "if")
    {
        token = lexer(infile, outfile);
        if(get<1>(token) != "(")
        {
            errorReporting(outfile, "(", get<1>(token));
        }
        outfile << "\tMatched (\n";

        token = lexer(infile, outfile);
        Condition(infile, outfile, token);

        if(get<1>(token) != ")")
        {
            errorReporting(outfile, ")", get<1>(token));
        }
        outfile << "\tMatched )\n";

        token = lexer(infile, outfile);
        if(!Statement(infile, outfile, token))
        {
            errorReporting(outfile, "<Statement>", get<1>(token));
        }
        IfEnd(infile, outfile, token);

        return true;
    }
    return false;
```

```
}

//R25. <Return> ::=  return <Return End>
bool Return(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR25. <Return> ::=  return <Return End>" << endl;
    }

    if(get<1>(token) == "return")
    {
        token = lexer(infile, outfile);
        ReturnEnd(infile, outfile, token);
        return true;
    }
    return false;
}

//R26. <Return End> ::= ; |  <Expression> ;
bool ReturnEnd(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR26. <Return End> ::= ; |  <Expression> ;" << endl;
    }
    if (get<1>(token) == ";")
    {
        outfile << "Matched ;\n";
        token = lexer(infile, outfile);
        return true;
    }

    Expression(infile, outfile, token);

    if (get<1>(token) != ";")
    {
        errorReporting(outfile, ";", get<1>(token));
    }
    outfile << "Matched ;\n";
    token = lexer(infile, outfile);
    return true;
}

//R27. <Print> ::=    put ( <Expression>);
bool Print(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR27. <Print> ::=    put ( <Expression>);" << endl;
```

```
    }
    if (get<1>(token) == "put")
    {
        outfile << "\tMatched put\n";

        token = lexer(infile, outfile);
        if (get<1>(token) != "(")
        {
            errorReporting(outfile, "(", get<1>(token));
        }

        token = lexer(infile, outfile);
        Expression(infile, outfile, token);

        if (get<1>(token) != ")")
        {
            errorReporting(outfile, ")", get<1>(token));
        }
        token = lexer(infile, outfile);

        if (get<1>(token) != ";")
        {
            errorReporting(outfile, ";", get<1>(token));
        }
        token = lexer(infile, outfile);
        return true;
    }
    return false;
}

//R28. <Scan> ::=    get ( <IDs> );
bool Scan(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR28. <Scan> ::=    get ( <IDs> );" << endl;
    }

    if (get<1>(token) == "get")
    {
        outfile << "\tMatched get\n";

        token = lexer(infile, outfile);
        if (get<1>(token) != "(")
        {
            errorReporting(outfile, "(", get<1>(token));
        }

        token = lexer(infile, outfile);
        IDs(infile, outfile, token);
```

```
        if (get<1>(token) != ")")
        {
            errorReporting(outfile, ")", get<1>(token));
        }

        token = lexer(infile, outfile);
        if (get<1>(token) != ";")
        {
            errorReporting(outfile, ";", get<1>(token));
        }
        token = lexer(infile, outfile);
        return true;
    }
    return false;
}

//R29. <While> ::=  while ( <Condition>  )  <Statement> whileend
bool While(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR29. <While> ::=  while ( <Condition>  )  <Statement>
         whileend" << endl;
    }

    if (get<1>(token) == "while")
    {
        outfile << "Matched while\n";

        token = lexer(infile, outfile);
        if (get<1>(token) != "(")
        {
            errorReporting(outfile, "(", get<1>(token));
        }

        token = lexer(infile, outfile);
        Condition(infile, outfile, token);

        if (get<1>(token) != ")")
        {
            errorReporting(outfile, ")", get<1>(token));
        }

        token = lexer(infile, outfile);
        if(!Statement(infile, outfile, token))
        {
            errorReporting(outfile, "{ | identifier | if | return | put | get |
             while", get<1>(token));
        }

        if (get<1>(token) != "whileend")
```

```
        {
            errorReporting(outfile, "whileend", get<1>(token));
        }
        token = lexer(infile, outfile);

        return true;
    }
    return false;
}

//R30. <Condition> ::=     <Expression>  <Relop>   <Expression>
void Condition(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR30. <Condition> ::=     <Expression>  <Relop>
         <Expression>" << endl;
    }
    Expression(infile, outfile, token);

    Relop(infile, outfile, token);

    Expression(infile, outfile, token);

}

//R24. <if End> ::= ifend | else <Statement> ifend
void IfEnd(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR24. <if End> ::= ifend | else <Statement> ifend" <<
         endl;
    }

    if(get<1>(token) == "else")
    {
        token = lexer(infile, outfile);
        if(!Statement(infile, outfile, token))
        {
            errorReporting(outfile, "{ | identifier | if | return | put | get |
             while", get<1>(token));
        }

        if(get<1>(token) != "ifend" && get<1>(token) != "ifEnd")
        {
            errorReporting(outfile, "ifEnd", get<1>(token));
        }
        token = lexer(infile, outfile);
    }
```

```
    else if(get<1>(token) == "ifend" || get<1>(token) == "ifEnd")
    {
        token  = lexer(infile, outfile);
    }
    else
    {
        errorReporting(outfile, "ifEnd | else", get<1>(token));
    }
}

//R31. <Relop> ::=  ==   |   ^=    |   >     |   <    |   =>    |   =<
void Relop(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR31. <Relop> ::=        ==   |   ^=    |   >     |   <
         |   =>    |   =<" << endl;
    }

    string temp = get<1>(token);
    if(temp != "==" && temp != "^=" && temp != ">" && temp != "<" && temp !=
     "=>" && temp != "=<")
    {
        errorReporting(outfile, "== | ^= | > | < | => | =<", get<1>(token));
    }
    token = lexer(infile, outfile);
}

//R32. <Expression>  ::=    <Term> <Expression'>
void Expression(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR32. <Expression> ::= <Term> <Expression'>" << endl;
    }

    Term(infile, outfile, token);
    ExpressionPrime(infile, outfile, token);
}

//R34. <Term>    ::=    <Factor>  <Term'>
void Term(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR34. <Term> ::= <Factor> <Term'>" << endl;
    }
    Factor(infile, outfile, token);
    TermPrime(infile, outfile, token);
}
```

```
//R36. <Factor> ::=  -  <Primary>  |  <Primary>
void Factor(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR36. <Factor> ::= - <Primary> | <Primary>" << endl;
    }

    if (get<1>(token) == "-")
    {
        token = lexer(infile, outfile);
        Primary(infile, outfile, token);
    }
    else
    {
        Primary(infile, outfile, token);
    }

}

//R37. <Primary> ::= <Identifier> <Primary End> | <Integer> | ( <Expression> )
 | <Real> | true | false
bool Primary(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    if (printSwitch)
    {
        outfile << "\tR37. <Primary> ::= <Identifier> <Primary End> | <Integer>
         | (<Expression>) | <Real> | true |false" << endl;
    }
    string expected ="<Identifier> | <Integer> | ( | <Real> | true | false";
    if(get<0>(token) == "Integer")
    {
        outfile << "\tR. <Integer>\n";
        token = lexer(infile, outfile);
        return true;
    }
    else if (get<1>(token) == "(")
    {
        outfile << "Used: (" << endl;

        token = lexer(infile, outfile);
        Expression(infile, outfile, token);

        if (get<1>(token) != ")")
        {
            errorReporting(outfile, ")", get<1>(token));
        }
        token = lexer(infile, outfile);
        return true;
    }
```

```
    else if (get<0>(token) == "Real")
    {
        outfile << "used R. <Real>\n";
        token = lexer(infile, outfile);
        return true;
    }
    else if (get<0>(token) == "Keyword")
    {
        string temp;
        int length = get<1>(token).length();
        for(int i = 0; i < length; ++i)
        {
            temp += tolower(get<1>(token)[i]);
        }
        if (temp == "true")
        {
            outfile << "\tR. true\n";
            token = lexer(infile, outfile);
            return true;
        }
        else if (temp == "false")
        {
            outfile << "\tR. false\n";
            token = lexer(infile, outfile);
            return true;
        }
        errorReporting(outfile, expected, get<1>(token));
        return false;
    }
    else if (Identifier(outfile, token))
    {
        token = lexer(infile, outfile);
        PrimaryEnd(infile, outfile, token);
        return true;
    }

    errorReporting(outfile, expected, get<1>(token));

    return false;
}

//R38. <Primary End> ::= ( <IDs> ) | ε
void PrimaryEnd(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR38. <Primary End> ::= ( <IDs> ) | ε" << endl;
    }
    if (get<1>(token) == "(")
    {
```

```
        token = lexer(infile, outfile);
        if(!IDs(infile, outfile, token))
        {
            errorReporting(outfile, "(", get<1>(token));
        }


        if (get<1>(token) != ")")
        {
            errorReporting(outfile, ")", get<1>(token));
        }
        token = lexer(infile, outfile);
    }
}

//R35.<Term'> ::= * <Factor> <Term'>  | / <Factor> <Term'> | ε
bool TermPrime(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR35.<Term'> ::= * <Factor> <Term'>  | / <Factor> <Term'>
         | ε" << endl;
    }

    if (get<1>(token) == "*")
    {
        token = lexer(infile, outfile);
        Factor(infile, outfile, token);
        TermPrime(infile, outfile, token);
        return true;
    }
    else if (get<1>(token) == "/")
    {
        token = lexer(infile, outfile);
        Factor(infile, outfile, token);
        TermPrime(infile, outfile, token);
        return true;
    }
    return false;
}

//R33. <Expression'> ::= + <Term> <Expression'>   |  - <Term>  <Expression'>
 | ε
bool ExpressionPrime(ifstream &infile, ostream &outfile, tuple<string, string>
 &token)
{
    if (printSwitch)
    {
        outfile << "\tR33. <Expression'> ::= + <Term> <Expression'>   |  -
         <Term>  <Expression'>   | ε" << endl;
```

```
    }

    if (get<1>(token) == "+")
    {
        token = lexer(infile, outfile);
        Term(infile, outfile, token);
        ExpressionPrime(infile, outfile, token);
        return true;
    }
    else if (get<1>(token) == "-")
    {
        token = lexer(infile, outfile);
        Term(infile, outfile, token);
        ExpressionPrime(infile, outfile, token);
        return true;
    }
    return false;
}

//R38. <Primary End> ::= ( <IDs> ) | ε
void empty(ifstream &infile, ostream &outfile, tuple<string, string> &token)
{
    outfile << "R39. <Empty>   ::= ε";
}

void errorReporting(ostream &outfile, string expected, string received)
{
    if(printSwitch)
    {
        outfile << "\nERROR: NOT VALID SYNTAX. on line: " << getLineNumber() <<
         "\n";
        outfile << "Expected: " << expected << "\nReceived: " << received <<
         endl;
    }
    cerr << "\nERROR: NOT VALID SYNTAX. On line: " << getLineNumber() << "\n";
    cerr << "Expected: " << expected << "\nReceived: " << received << endl;
    exit(1);
}
```