```
#include "parse.h"
using namespace std;

map<string, bool> defVar;
map<string, Token> SymTable;

int error_count = 0;

namespace Parser{
    bool pushed_back = false;
    LexItem pushed_token;

    static LexItem GetNextToken(istream&in, int& line){
        if (pushed_back){
            pushed_back = false;
            return pushed_token;
        }
        return getNextToken(in, line);
    }

    static void PushBackToken(LexItem& t){
        if (pushed_back){
            abort();
        }
        pushed_back = true;
        pushed_token = t;
    }
}


int ErrCount(){
    return error_count;
}

void ParseError(int line, string msg){
    error_count++;
    cout << line << ": " << msg << endl;
}

//Prog ::= PROGRAM IDENT StmtList END PROGRAM
bool Prog(istream& in, int& line){
    bool status = true;
    LexItem t = Parser::GetNextToken(in, line);

    //check if begins with program and ident
    if (t.GetToken() != PROGRAM){
        //Parser::PushBackToken(t);
        if (t.GetToken() == DONE){
            ParseError(line, "Empty File");
            return false;
        }
        Parser::PushBackToken(t);
        ParseError(line, "Missing PROGRAM.");
        return false;
    }
    t = Parser::GetNextToken(in, line);
    if (t.GetToken() != IDENT){
        Parser::PushBackToken(t);
        ParseError(line, "Missing Program Name.");
```

```
        return false;
    }

    status = StmtList(in, line);
    if (!status){
        ParseError(line, "Incorrect Syntax in the Program.");
        return false;
    }
    t = Parser::GetNextToken(in, line);
    if (t.GetToken() != END){
        Parser::PushBackToken(t);
        ParseError(line, "Missing END at end of program.");
        return false;
    }
    t = Parser::GetNextToken(in, line);
    if (t.GetToken() != PROGRAM){
        Parser::PushBackToken(t);
        ParseError (line, "Missing PROGRAM at the End");
        return false;
    }
    return status;
}


//DeclStmt ::= (INT | FLOAT) IdentList
bool DeclStmt(istream& in, int& line) {
      bool status = false;
      LexItem tok;
      //cout << "in Decl" << endl;
      LexItem t = Parser::GetNextToken(in, line);

      if(t == INT || t == FLOAT ) {
            status = IdentList(in, line, t);
            //cout<< "returning from IdentList" << " " << (status? 1: 0) << endl;
            if (!status){
                  ParseError(line, "Incorrect variable in Declaration Statement.");
                  return status;
            }
      }
      else{
            Parser::PushBackToken(t);
            ParseError(line, "Incorrect Type.");
            return false;
      }
      return true;
}

//IdentList ::= IDENT, {,IDENT}
bool IdentList(istream& in, int& line, LexItem tok){
    bool status = true;
    tok = Parser::GetNextToken(in, line);

    if (tok.GetToken() != IDENT){
        ParseError(line, "Invalid Identifier List");
        return false;
    }


    if (defVar.find(tok.GetLexeme()) != defVar.end()){ //if variable already in map
```

```
        ParseError(line, "Variable Redefinition");
        return false;
    }
    defVar[tok.GetLexeme()] = true; //add to map
    SymTable[tok.GetLexeme()] = tok.GetToken();

    tok = Parser::GetNextToken(in, line);

    if (tok.GetToken() == COMMA){
        status = IdentList(in, line, tok);
    }
    else{
        Parser::PushBackToken(tok);
    }
    return status;

}

//StmtList ::= Stmt; {Stmt;}
bool StmtList(istream& in, int& line){

    bool status = Stmt(in, line);

    if (!status){
        ParseError(line, "Invalid Statement List");
        return false;
    }

    LexItem t = Parser::GetNextToken(in, line);

    if (t.GetToken() == END){
        Parser::PushBackToken(t);
        return status;
    }

    if (t.GetToken() != SEMICOL){
        if (t.GetToken() == PROGRAM){
            ParseError(line, "Missing END at end of program.");
            return false;
        }
        ParseError(line, "Missing a semicolon.");
        return false;
    }

    status = StmtList(in, line);
    return status;
}

//Stmt ::= DeclStmt | ControlStmt
bool Stmt(istream& in, int& line){
    bool status=true;
    //cout << "in Stmt" << endl;
    LexItem t = Parser::GetNextToken(in, line);

    switch( t.GetToken() ) {
    case INT: case FLOAT:
        Parser::PushBackToken(t);
        status = DeclStmt(in, line);
        if(!status)
```

```
        {
            ParseError(line, "Incorrect Declaration Statement.");
            return status;
        }
        break;
    case IF: case WRITE: case IDENT:
        Parser::PushBackToken(t);
        status = ControlStmt(in, line);
        if(!status)
        {
            ParseError(line, "Incorrect control Statement.");
            return status;
        }
        break;
    default:
        Parser::PushBackToken(t);
    }
    return status;
}

//ControlStmt ::= AssignStmt | IfStmt | WriteStmt
bool ControlStmt(istream& in, int& line){
    bool status = true;
    LexItem t = Parser::GetNextToken(in, line);


    if (t.GetToken() == IDENT){
        Parser::PushBackToken(t);
        status = AssignStmt(in, line);
        if(!status){
            ParseError(line, "Incorrect Assignment Statement.");
            return false;
        }
    }
    else if (t.GetToken() == IF){
        Parser::PushBackToken(t);
        status = IfStmt(in, line);
        if (!status){
            ParseError(line, "Incorrect If Statement.");
            return false;
        }
    }
    else if (t.GetToken() == WRITE){
        status = WriteStmt(in, line);
        if(!status){
            ParseError(line, "Incorrect Write Statement");
            return false;
        }
    }
    else{
        Parser::PushBackToken(t);
    }

    return status;
}

//WriteStmt ::= WRITE ExprList
bool WriteStmt(istream& in, int& line){
    LexItem t;
```

```
    bool ex = ExprList(in, line);

    if (!ex){
        ParseError(line, "Missing expression after Write");
        return false;
    }
    return ex;
}

//IfStmt ::= IF(LogicExpr) ControlStmt
bool IfStmt(istream& in, int& line){
    LexItem t = Parser::GetNextToken(in, line);
    bool status = false;

    if (t.GetToken() != IF){
        ParseError(line, "missing IF");
        return false;
    }
    t = Parser::GetNextToken(in, line);
    if (t.GetToken() != LPAREN){
        ParseError(line, "Missing left paren in IF");
        return false;
    }

    status = LogicExpr(in, line);

    if (!status){
        ParseError(line, "Incorrect statement in program If Statement");
        return false;
    }
    t = Parser::GetNextToken(in, line);
    if (t.GetToken() != RPAREN){
        ParseError(line, "Missing Right Parenthesis");
        return false;
    }
    //t = Parser::GetNextToken(in, line);

    status = ControlStmt(in, line);

    if (!status){
        ParseError(line, "Incorrect Statement in program");
        return false;
    }
    return status;
}

//AssignStmt ::= Var = Expr
bool AssignStmt(istream& in, int& line){
    bool status = Var(in, line);
    LexItem t = Parser::GetNextToken(in, line);

    //t = Parser::GetNextToken(in, line);
    if (t.GetToken() != ASSOP){
        Parser::PushBackToken(t);
        ParseError(line, "Missing assignment op");
        return false;
    }
    status = Expr(in, line);
```

```
    if (!status){
        Parser::PushBackToken(t);
        ParseError(line, "Invalid assignment var");
        return false;
    }
    return true;
}

//ExprList ::= Expr {,Expr}
bool ExprList(istream& in, int& line){

    bool status = Expr(in, line);
    //cout << t.GetToken() << " " << t.GetLexeme() << " " << line << endl;

    if (!status){
        ParseError(line, "Invalid Expression List error");
        return false;
    }

    LexItem t = Parser::GetNextToken(in, line);
    if (t.GetToken() == COMMA){
        status = ExprList(in, line);
    }
    else{
        Parser::PushBackToken(t);
    }
    return status;
}

//Expr ::= Term {(+|-) Term}
bool Expr(istream& in, int& line){
    bool status = Term(in, line);

    if (!status){
        ParseError(line, "Expression error");
        return false;
    }

    LexItem t = Parser::GetNextToken(in, line);
    if (t.GetToken() == PLUS || t.GetToken() == MINUS){
        status = Term(in, line);
    }
    else{
        Parser::PushBackToken(t);
    }
    return status;
}

//Term ::= SFactor{(*|/|%) SFactor}
bool Term(istream& in, int& line){
    bool status = SFactor(in, line);

    if (!status){
        ParseError(line, "Term Error 1");
        return false;
    }
    LexItem t = Parser::GetNextToken(in, line);
```

```
    if (t.GetToken() == MULT || t.GetToken() == DIV || t.GetToken() == REM){
        status = Term(in, line);
    }
    else{
        Parser::PushBackToken(t);
    }
    return status;
}

//SFactor ::= (+|-) Factor | Factor
bool SFactor(istream& in, int& line){
    LexItem t = Parser::GetNextToken(in, line);
    bool status = true;

    if (t.GetToken() == PLUS || t.GetToken() == MINUS){
        status = Factor(in, line, 1);
        if (!status){
            ParseError(line, "SFactor Error 1");
            return false;
        }
    }
    else{
        Parser::PushBackToken(t);
        status = Factor(in, line, 1);
        if(!status){
            ParseError(line, "SFactor Error 2");
            return false;
        }
    }
    return status;
}

//LogicExpr ::= Expr (==|>) Expr
bool LogicExpr(istream& in, int& line){
    bool status = Expr(in, line);

    if (!status){
        ParseError(line, "Logic expression error");
        return false;
    }
    LexItem t = Parser::GetNextToken(in, line);

    if (t.GetToken() == EQUAL || t.GetToken() == GTHAN){
        status = Expr(in, line);
        if (!status){
            ParseError(line, "Logic Expression error 1");
        }
        return status;
    }
    else{
        ParseError(line, "invalid operator Error");
        return false;
    }
    return status;
}

//Var ::= IDENT
bool Var(istream& in, int& line){
    LexItem t = Parser::GetNextToken(in, line);
```

```
    if (t.GetToken() == IDENT){
        return true;
    }
    else{
        ParseError(line, "Incorrect Identifier Statement");
        return false;
    }
}

//Factor = IDENT | ICONST | RCONST | SCONST | (Expr)
bool Factor(istream& in, int& line, int sign){
    LexItem t = Parser::GetNextToken(in, line);
    bool status = true;

    if (t.GetToken() == IDENT){
        if (defVar.count(t.GetLexeme()) == 0){
            ParseError(line, "Undeclared variable");
        }
        return status;
    }
    else if (t.GetToken() == ICONST){
        return status;
    }
    else if (t.GetToken() == RCONST){
        return status;
    }
    else if (t.GetToken() == SCONST){
        return status;
    }

    if (t.GetToken() != LPAREN){
        ParseError(line, "No left parenthesis");
        return false;
    }
    status = Expr(in, line);
    if (!status){
        ParseError(line, "Factor error");
        return false;
    }

    t = Parser::GetNextToken(in, line);
    if(t.GetToken() != RPAREN){
        ParseError(line, "No right parenthesis");
        return false;
    }
    return status;

}
```